

Apache Unomi 4.x - Documentation

Apache Software Foundation

Version 4.0.0

Table of contents

1. Foreword . . . 2
2. What's new . . . 3
   2.1. What's new in Apache Unomi 4.0 . . . 3
      2.1.1. Multi-tenancy and API keys . . . 3
      2.1.2. Security hardening (credentials, profile binding, privileged APIs) . . . 3
      2.1.3. Web tracker 2.0 . . . 3
      2.1.4. Visitor cookie across websites . . . 4
      2.1.5. Cluster-aware task scheduler . . . 4
      2.1.6. OpenSearch 3 support . . . 4
      2.1.7. Tenant usage and retention . . . 4
      2.1.8. Request tracing . . . 4
      2.1.9. Condition validation . . . 5
   2.2. What's new in Apache Unomi 3.0 . . . 5
      2.2.1. Java 17 . . . 5
      2.2.2. Karaf 4.4 . . . 5
      2.2.3. Elasticsearch 9 Java API client . . . 5
      2.2.4. QueryBuilder ID renames . . . 5
      2.2.5. Persistence-based clustering . . . 5
      2.2.6. Health check extension . . . 5
3. Discover Unomi . . . 6
   3.1. Quick start with Docker . . . 6
      3.1.1. Option 1: Using Elasticsearch . . . 6
      3.1.2. Option 2: Using OpenSearch . . . 6
      3.1.3. After startup (Elasticsearch Docker path) . . . 8
   3.2. Quick Start manually . . . 8
      3.2.1. Option 1: Using Elasticsearch . . . 8
      3.2.2. Option 2: Using OpenSearch . . . 8
      3.2.3. Complete the Setup . . . 9
   3.3. Getting started with Unomi . . . 11
      3.3.1. Prerequisites . . . 11
      3.3.2. Running Unomi . . . 11
   3.4. Unomi web tracking tutorial . . . 13
      3.4.1. Installing the web tracker in a web page . . . 14
      3.4.2. Creating a scope to collect the data . . . 16
      3.4.3. Using tracker in your own JavaScript projects . . . 16
      3.4.4. Viewing collected events . . . 18
      3.4.5. Viewing the current profile . . . 18
      3.4.6. Adding a rule . . . 19
      3.4.7. Adding personalization . . . 19
      3.4.8. Conclusion . . . 21
      3.4.9. Next steps . . . 21
4. Apache Unomi Recipes and requests . . . 22
   4.1. Recipes . . . 22
      4.1.1. Introduction . . . 22
      4.1.2. Enabling debug mode . . . 22
      4.1.3. How to read a profile . . . 23
      4.1.4. How to update a profile from the public internet . . . 24
      4.1.5. How to search for profile events . . . 29
      4.1.6. How to create a new rule . . . 30

4.1.7. How to search for profiles . . . 30
4.1.8. Getting / updating consents . . . 31
4.1.9. How to send a login event to Unomi . . . 31
4.1.10. Technical Implementation Details . . . 32
4.1.11. What profile aliases are and how to use them . . . 33
4.1.12. Common Issues and Best Practices . . . 36
4.2. Request examples . . . 38
4.2.1. Prerequisites . . . 38
4.2.2. Retrieving your first context . . . 42
4.2.3. Retrieving a context as a JSON object. . . . 42
4.2.4. Accessing profile properties in a context . . . 43
4.2.5. Sending events using the context servlet . . . 43
4.2.6. Sending events using the eventcollector servlet . . . 44
4.2.7. Where to go from here . . . 45
4.3. Public API Examples . . . 45
4.3.1. Sending a context request . . . 45
4.3.2. Collecting events . . . 46
4.4. Private API Examples . . . 47
4.4.1. Setting up birthday personalization . . . 47
4.4.2. Searching profiles . . . 55
4.4.3. Creating a segment . . . 55
4.4.4. Setting up product view tracking . . . 56
4.4.5. Searching profiles with frequent product views . . . 61
4.5. Setting up Groovy Actions . . . 63
4.5.1. Deploying a Groovy Action . . . 63
4.5.2. Using the explain parameter for request tracing . . . 68
5. Configuration . . . 77
5.1. Centralized configuration . . . 77
5.2. Changing the default configuration using environment variables (i.e. Docker configuration) . . . 77
5.3. Changing the default configuration using property files . . . 77
5.4. Refresh Policy Configuration . . . 79
5.5. Secured events configuration . . . 80
5.5.1. Normal multi-tenant mode . . . 80
5.5.2. Single-tenant compatibility mode (third-party / peer servers) . . . 80
5.6. Installing the MaxMind GeoIPLite2 IP lookup database . . . 82
5.7. Installing Geonames database . . . 82
5.8. REST API Security . . . 83
5.9. Tenant Management and API Access . . . 83
5.9.1. Creating and Managing Tenants . . . 83
5.9.2. API Keys and Authentication . . . 85
5.9.3. Accessing API Endpoints . . . 86
5.9.4. Websites on other origins (visitor cookie and CORS) . . . 89
5.9.5. Admin UI origins (credentialed CORS) . . . 94
5.9.6. GraphQL query limits . . . 95
5.10. Scripting security . . . 95
5.10.1. Multi-layer scripting filtering system . . . 95
5.10.2. Scripts and expressions . . . 97
5.10.3. Scripting expression filtering configuration parameters . . . 99
5.10.4. Groovy Actions . . . 101
5.10.5. Scripting roadmap . . . 105
5.11. Automatic profile merging . . . 106

5.11.1. How profile merging works . . . 106
5.11.2. Before and after a profile merge . . . 106
5.11.3. Configuration parameters . . . 108
5.11.4. Security considerations . . . 108
5.11.5. Troubleshooting profile merges . . . 110
5.11.6. Performance considerations . . . 110
5.12. Securing a production environment . . . 110
5.13. Integrating with an Apache HTTP web server . . . 112
5.14. Changing the default tracking location . . . 114
5.15. Apache Karaf SSH Console . . . 115
5.16. Search engine authentication and security . . . 115
5.16.1. User authentication . . . 115
5.16.2. SSL communication . . . 116
5.16.3. Permissions . . . 116
5.17. Unomi Distribution (features configuration) . . . 117
5.17.1. Default Distributions . . . 117
5.17.2. Environment-Specific Distributions . . . 117
5.17.3. Distribution Feature Format . . . 120
5.17.4. Using Custom Distributions . . . 121
5.17.5. Auto-Start . . . 121
5.18. Customizing Apache Unomi Distribution . . . 121
5.18.1. Note on Custom Distributions (Advanced) . . . 121
5.18.2. Important Notes . . . 122
5.18.3. Available Features . . . 122
5.18.4. Docker Deployment with Custom Distribution . . . 122
5.18.5. Configuration Override Priority . . . 123
5.18.6. Best Practices for Custom Distributions . . . 124
5.18.7. Deployment with Custom Distribution . . . 124
5.19. Health Check Extension . . . 125
5.19.1. Health check for another persistence backend . . . 127
5.19.2. Configuration . . . 127
6. Multi-tenancy . . . 129
6.1. Multi-tenancy guide . . . 129
6.1.1. Overview . . . 129
6.1.2. Key Features . . . 129
6.1.3. Authentication Methods . . . 129
6.1.4. Getting Started . . . 130
6.1.5. Configuration . . . 132
6.1.6. Security Model . . . 133
6.1.7. Tenant Management . . . 137
6.1.8. GraphQL Support . . . 138
6.1.9. Monitoring and Management . . . 139
6.1.10. Best Practices . . . 140
6.1.11. Troubleshooting . . . 141
6.1.12. Migration Guide . . . 141
6.1.13. Working with Events and Rules . . . 142
7. JSON schemas . . . 150
7.1. Introduction . . . 150
7.1.1. What is a JSON Schema . . . 150
7.1.2. Key concepts . . . 151
7.1.3. How are JSON Schema used in Unomi . . . 154

7.2. JSON schema API . . . 155
7.2.1. List existing schemas . . . 155
7.2.2. Read a schema . . . 156
7.2.3. Create / update a JSON schema to validate an event . . . 156
7.2.4. Deleting a schema . . . 157
7.2.5. Error Management . . . 157
7.2.6. Details on invalid events . . . 158
7.3. Develop with Unomi and JSON Schemas . . . 158
7.3.1. Logs in debug mode . . . 158
7.3.2. validateEvent endpoint . . . 158
7.3.3. validateEvents endpoint . . . 159
7.4. Extend an existing schema . . . 160
7.4.1. When a extension is needed? . . . 161
7.4.2. Understanding how extensions are merged in unomi . . . 161
7.4.3. How to add an extension through the API . . . 162
8. GraphQL API . . . 163
8.1. Introduction . . . 163
8.2. Enabling the API . . . 163
8.3. Endpoints . . . 163
8.4. GraphQL Schema . . . 163
8.4.1. Authentication . . . 163
8.5. Multi-Tenancy Support . . . 164
8.5.1. Overview . . . 164
8.5.2. Architecture . . . 164
8.5.3. Schema Lifecycle . . . 165
8.5.4. Performance Considerations . . . 165
8.5.5. Benefits . . . 165
8.5.6. Troubleshooting . . . 165
8.6. Graphql request examples . . . 166
8.6.1. Retrieving your first profile . . . 166
8.6.2. Updating profile . . . 167
8.6.3. Restricted methods . . . 168
8.6.4. Deleting profile . . . 169
8.6.5. Where to go from here . . . 170
9. Migrations . . . 171
9.1. Writing migration scripts . . . 172
9.1.1. Script location and naming . . . 173
9.1.2. Use performMigrationStep for every resumable unit of work . . . 173
9.1.3. Pitfall: do not nest performMigrationStep inside another step . . . 174
9.1.4. Pitfall: make data transforms safe to re-run . . . 174
9.1.5. Pitfall: bundled resources must exist and fail clearly . . . 175
9.1.6. Configuration keys . . . 175
9.1.7. Destructive operations . . . 175
9.1.8. Testing . . . 176
9.1.9. Pre-merge checklist . . . 176
9.2. Unomi 2 / 3 to 4.0 API compatibility . . . 176
9.3. Apache Unomi 2 / 3 to 4.0 API differences . . . 177
9.3.1. Overview . . . 177
9.3.2. Key differences between Unomi 2 / 3 and Unomi 4 . . . 177
9.3.3. Authentication Flow (4.0) . . . 178
9.3.4. Code Examples . . . 178

9.3.5. Implementation Strategy . . . 179
9.3.6. Migration Guidelines . . . 180
9.3.7. Conclusion . . . 181
9.4. single-tenant compatibility mode . . . 181
9.4.1. Overview . . . 181
9.4.2. Prerequisites . . . 182
9.4.3. Configuration . . . 182
9.4.4. Migration Workflow . . . 183
9.4.5. Client Migration Examples . . . 184
9.4.6. Security Considerations . . . 185
9.4.7. Troubleshooting . . . 188
9.4.8. Disabling single-tenant compatibility mode . . . 188
9.4.9. Testing single-tenant compatibility mode . . . 189
9.4.10. Conclusion . . . 189
9.5. From version 3.0 to 4.0 . . . 189
9.6. Migration Overview . . . 189
9.7. Updating applications consuming Unomi . . . 189
9.7.1. Authentication Model Changes . . . 189
9.7.2. API Key Types (4.0 Only) . . . 190
9.7.3. Authentication Requirements by Endpoint Type . . . 190
9.7.4. Authentication Flow (4.0) . . . 190
9.7.5. Code Examples . . . 191
9.7.6. Implementation Strategy . . . 192
9.7.7. API paths stay the same; behaviour does not . . . 193
9.7.8. Client-facing hardening (4.0) . . . 193
9.7.9. A search condition states its type (4.0) . . . 195
9.7.10. An unset optional parameter no longer makes an OR match (4.0) . . . 196
9.7.11. Segment and scoring lists merge system and tenant items (4.0) . . . 196
9.8. Migrating your existing data . . . 197
9.8.1. Multi-Tenancy Impact . . . 197
9.8.2. Recurrent import and export files move under a tenant directory . . . 197
9.8.3. Background tasks and recurrent routes run for every tenant . . . 198
9.8.4. Migration Steps . . . 198
9.8.5. Client-facing hardening (profile cookie, passwords, privileged APIs) . . . 199
9.8.6. Benefits of Multi-Tenancy in 4.0 . . . 203
9.9. Breaking changes . . . 203
9.9.1. Operators . . . 203
9.9.2. Integrators / REST clients . . . 204
9.9.3. Plugin developers . . . 204
9.10. Migration Checklist . . . 205
9.11. Migration Process . . . 205
9.12. Transitional authentication options . . . 206
9.12.1. Option 1: Update clients to 4.0 authentication (recommended) . . . 206
9.12.2. Option 2: single-tenant compatibility mode (Unomi 2 and Unomi 3 clients) . . . 206
9.13. Post Migration . . . 207
9.13.1. Migrating from Elasticsearch to OpenSearch . . . 207
9.14. From version 2.x to 3.0 . . . 210
9.14.1. Migration Overview . . . 210
9.14.2. Prerequisites . . . 210
9.14.3. Elasticsearch Client Upgrade . . . 211
9.14.4. QueryBuilder ID Changes . . . 211

9.14.5. Configuration Changes . . . 215
9.14.6. Elasticsearch to OpenSearch Migration . . . 215
9.14.7. Migration Checklist . . . 216
9.14.8. Other Breaking Changes . . . 216
9.14.9. Migrate from Elasticsearch 7 to Elasticsearch 9 . . . 216
9.14.10. Execution . . . 218
9.15. From version 1.6 to 2.0 . . . 219
9.16. Migration Overview . . . 219
9.17. Updating applications consuming Unomi . . . 219
9.17.1. Data Model changes . . . 219
9.17.2. Create JSON schemas . . . 219
9.18. Migrating your existing data . . . 220
9.18.1. Elasticsearch version and capacity . . . 220
9.18.2. Migrate custom data . . . 220
9.18.3. Perform the migration . . . 221
9.19. From version 1.5 to 1.6 . . . 224
9.20. From version 1.4 to 1.5 . . . 224
9.20.1. Migration from Apache Unomi 1.4 to 1.5 . . . 224
9.21. Important changes in public servlets since version 1.5.5 and 2.0.0 . . . 227
10. Conditions . . . 228
10.1. Conditions guide . . . 228
10.1.1. Concepts . . . 228
10.1.2. Condition instance JSON shape . . . 229
10.1.3. Comparison operators . . . 230
10.1.4. Level 1 — Profile property check . . . 230
10.1.5. Level 2 — Event type in a rule . . . 230
10.1.6. Level 3 — Boolean AND / OR . . . 231
10.1.7. Level 4 — Segment membership and session properties . . . 232
10.1.8. Level 5 — Past events (behavioral) . . . 233
10.1.9. Level 6 — Scoring, geo, and nested data . . . 233
10.1.10. Using conditions in segments and rules . . . 234
10.1.11. Testing conditions . . . 235
10.1.12. Discovering available condition types . . . 235
10.1.13. Custom condition types . . . 236
10.1.14. Debugging conditions . . . 236
10.2. Condition validation . . . 237
10.2.1. When validation runs . . . 237
10.2.2. Validation metadata on condition types . . . 238
10.2.3. Parameter type checks . . . 239
10.2.4. Contextual parameters (partial validation) . . . 239
10.2.5. Error types . . . 240
10.2.6. Save-time behavior . . . 240
10.2.7. Rules and invalid conditions . . . 241
10.2.8. Validation in request tracing . . . 241
10.2.9. Adding validation to custom condition types . . . 241
10.2.10. Relation to JSON Schema validation . . . 241
10.3. Condition Evaluation System . . . 242
10.3.1. Architecture Overview . . . 242
10.4. Condition Evaluation Architecture . . . 242
10.4.1. Key Components . . . 242
10.4.2. Evaluation Strategies . . . 243

10.4.3. Query Builder Implementation . . . 243
10.4.4. Best Practices . . . 244
10.5. Past Event Conditions in Apache Unomi . . . 244
10.5.1. Introduction . . . 244
10.5.2. Mobile Application Engagement . . . 247
10.5.3. Customer Data Platform (CDP) Integration . . . 248
10.5.4. B2B Use Cases . . . 250
10.5.5. Architecture Overview . . . 252
10.5.6. Event Processing Flow . . . 254
10.5.7. Segment vs Direct Rule Comparison . . . 254
10.5.8. Cache Update Process . . . 255
10.5.9. Query Strategy Selection . . . 256
10.5.10. Usage Patterns . . . 257
10.5.11. Auto-Generated Rules . . . 259
10.5.12. Core Components . . . 259
10.5.13. Condition Parameters . . . 260
10.5.14. Event Processing System . . . 261
10.5.15. Rule Generation . . . 266
10.5.16. Best Practices . . . 267
10.5.17. Common Use Cases and Examples . . . 269
10.5.18. Troubleshooting . . . 273
10.5.19. Integration Points . . . 275
10.5.20. Security Considerations . . . 276
10.5.21. Configuration Parameters . . . 278
10.5.22. Best Practices . . . 278
10.5.23. Evaluation Strategies . . . 280
10.5.24. Performance Considerations . . . 281
10.5.25. Maintenance and Monitoring . . . 284
11. Queries and aggregations . . . 286
11.1. Query counts . . . 286
11.2. Metrics . . . 287
11.3. Aggregations . . . 288
11.3.1. Aggregation types . . . 288
11.3.2. Reading the results . . . 295
11.4. Scroll Queries . . . 296
11.4.1. Initial Scroll Query . . . 296
11.4.2. Continue Scroll Query . . . 297
11.4.3. Query Parameters . . . 297
12. Profile import & export . . . 298
12.1. Importing profiles . . . 298
12.1.1. Import API . . . 298
12.2. Exporting profiles . . . 300
12.2.1. Export API . . . 300
12.3. Configuration in details . . . 302
13. Consent management . . . 305
13.1. Consent API . . . 305
13.1.1. Profiles with consents . . . 305
13.1.2. Consent type definitions . . . 306
13.1.3. Creating / update a visitor consent . . . 306
13.1.4. How it works (internally) . . . 308
14. Privacy management . . . 310

14.1. Setting up access to the privacy endpoint . . . 310
14.2. Anonymizing a profile . . . 310
14.3. Downloading profile data . . . 310
14.4. Deleting a profile . . . 311
14.5. Related . . . 311
15. Cluster setup . . . 312
15.1. Cluster setup . . . 312
15.1.1. Required settings . . . 312
15.1.2. Node identity . . . 312
15.1.3. How it works . . . 313
15.1.4. Scheduler and executor nodes . . . 314
15.1.5. Operations . . . 314
15.1.6. Migration from Unomi 2.x / Cellar . . . 314
16. Task scheduler . . . 315
16.1. Cluster-aware task scheduler . . . 315
16.1.1. Quick example: periodic cache refresh . . . 315
16.1.2. Architecture overview . . . 315
16.1.3. Task lifecycle . . . 315
16.1.4. Execution flow in a cluster . . . 316
16.1.5. Task states in detail . . . 317
16.1.6. Exclusive vs parallel execution . . . 318
16.1.7. How distributed locks work with Elasticsearch / OpenSearch . . . 319
16.1.8. Node affinity timeline . . . 325
16.1.9. Crash recovery . . . 326
16.1.10. In-memory (non-persistent) tasks . . . 327
16.1.11. Operational checklist for clustering . . . 327
16.2. Core concepts . . . 327
16.3. Task model . . . 328
16.4. Configuration . . . 329
16.5. REST API . . . 330
16.5.1. List tasks . . . 330
16.5.2. Get one task . . . 331
16.5.3. Cancel a task . . . 331
16.5.4. Retry a failed task . . . 331
16.5.5. Resume a crashed task . . . 331
16.6. Karaf shell commands . . . 332
16.7. Using the scheduler in plugins . . . 332
16.7.1. Pattern A: one-shot async work (real example) . . . 333
16.7.2. Pattern B: simple recurring maintenance . . . 335
16.7.3. Pattern C: fluent builder (recommended) . . . 335
16.7.4. Pattern D: resumable long-running executor . . . 336
16.7.5. OSGi Declarative Services wiring . . . 337
16.8. Built-in tasks you will see in production . . . 338
16.9. Metrics . . . 338
16.10. Choosing persistence and execution settings . . . 339
16.11. Troubleshooting . . . 339
16.12. Related documentation . . . 340
17. Health check . . . 341
18. Reference . . . 342
18.1. Architecture Overview . . . 342
18.2. High-Level Architecture . . . 342

18.3. Service Architecture . . . 342
18.4. Condition Evaluation Architecture . . . 343
18.4.1. Key Components . . . 343
18.5. Data Flow . . . 344
18.5.1. Data Structures . . . 344
18.5.2. Core Data Structures Overview . . . 344
18.5.3. Base Types . . . 344
18.5.4. Core Entities and Their Relationships . . . 345
18.5.5. Key Relationships . . . 346
18.5.6. Detailed Structure Definitions . . . 347
18.5.7. Condition Structure . . . 347
18.5.8. Segment Structure . . . 348
18.5.9. Rule Structure . . . 349
18.5.10. Goal Structure . . . 350
18.5.11. Campaign Structure . . . 351
18.5.12. Integration Flow . . . 352
18.5.13. Condition Composition Structure . . . 354
18.5.14. Event Processing Architecture . . . 356
18.5.15. Event Processing Architecture . . . 356
18.5.16. Security Architecture . . . 357
18.5.17. Security Architecture . . . 357
18.5.18. Request tracing . . . 361
18.6. Useful Apache Unomi URLs . . . 361
18.7. How profile tracking works . . . 363
18.7.1. Overview . . . 363
18.7.2. Detailed Request Flow . . . 365
18.7.3. First-Time Visitor Flow . . . 384
18.7.4. Complete Example: Full Request Flow . . . 387
18.7.5. Summary . . . 390
18.7.6. Configuration . . . 390
18.8. Context Request Flow . . . 391
18.9. Data Model Overview . . . 393
18.10. Scope . . . 394
18.10.1. Example . . . 395
18.11. Item . . . 395
18.12. Tenant (Unomi 4.0+) . . . 395
18.12.1. Structure definition . . . 396
18.13. Metadata . . . 396
18.13.1. Structure definition . . . 397
18.13.2. Example . . . 398
18.14. MetadataItem . . . 398
18.14.1. Structure definition . . . 398
18.14.2. Example . . . 398
18.15. Event . . . 399
18.15.1. Fields . . . 399
18.15.2. Event types . . . 400
18.16. Profile . . . 400
18.16.1. Structure definition . . . 401
18.16.2. Example . . . 401
18.17. Profile aliases . . . 404
18.17.1. Structure definition . . . 404

18.17.2. Example . . . 404
18.18. Persona . . . 404
18.18.1. Structure definition . . . 405
18.18.2. Example . . . 405
18.19. Consent . . . 405
18.19.1. Structure definition . . . 405
18.19.2. Example . . . 406
18.20. Session . . . 406
18.20.1. Structure definition . . . 406
18.20.2. Example . . . 407
18.21. Segment . . . 409
18.21.1. Structure definition . . . 409
18.21.2. Example . . . 409
18.22. Condition . . . 411
18.22.1. Structure definition . . . 412
18.22.2. Example . . . 412
18.23. Rule . . . 413
18.23.1. Structure definition . . . 415
18.23.2. Example . . . 415
18.24. Action . . . 416
18.24.1. Structure definition . . . 416
18.24.2. Example . . . 417
18.25. List . . . 417
18.25.1. Structure definition . . . 417
18.25.2. Example . . . 417
18.26. Goal . . . 418
18.26.1. Structure definition . . . 418
18.26.2. Example . . . 418
18.27. Campaign . . . 419
18.27.1. Structure definition . . . 419
18.27.2. Example . . . 420
18.28. Scoring plan . . . 421
18.28.1. Structure definition . . . 422
18.28.2. Example . . . 422
18.29. Property Types . . . 423
18.29.1. Quick Reference: Property Types vs JSON Schemas . . . 423
18.29.2. Overview . . . 423
18.29.3. Property Type Targets . . . 424
18.29.4. Property Type Structure . . . 424
18.29.5. Value Types . . . 427
18.29.6. Examples . . . 427
18.29.7. Dynamic Creation and Management . . . 428
18.29.8. Creating Property Types . . . 429
18.29.9. Querying Property Types . . . 430
18.29.10. Property Type Features . . . 431
18.29.11. Tags and System Tags . . . 434
18.29.12. System Tags . . . 440
18.29.13. Validation and Property Types . . . 440
18.29.14. Best Practices . . . 442
18.29.15. Built-in Property Types . . . 443
18.29.16. Property Types vs JSON Schemas . . . 446

18.29.17. Property Types for Events . . . 448
18.29.18. Building UIs with Property Types . . . 448
18.29.19. Related Topics . . . 455
18.30. Built-in Event types . . . 455
18.30.1. Login event type . . . 455
18.30.2. View event type . . . 457
18.30.3. Form event type . . . 458
18.30.4. Update properties event type . . . 460
18.30.5. Identify event type . . . 463
18.30.6. Session created event type . . . 464
18.30.7. Goal event type . . . 465
18.30.8. Modify consent event type . . . 467
18.31. Built-in condition types . . . 470
18.31.1. Condition type descriptor shape . . . 470
18.31.2. Plugin registration (Unomi 3.0+) . . . 471
18.31.3. Catalog — base plugin . . . 471
18.31.4. Catalog — extensions . . . 474
18.31.5. Source locations . . . 474
18.32. Built-in action types . . . 475
18.32.1. Action type descriptor shape . . . 475
18.32.2. OSGi registration example . . . 475
18.32.3. Catalog . . . 475
18.32.4. Common action groups . . . 477
18.32.5. Source locations . . . 477
18.33. Updating Events Using the Context Servlet . . . 478
18.33.1. Authentication Requirements . . . 478
18.33.2. Solution . . . 478
18.33.3. Defining Rules . . . 479
18.34. Unomi Web Tracker reference . . . 479
18.34.1. Configuring the tracker for your Unomi server . . . 479
18.34.2. Upgrading the tracker from 1.x to 2.0 . . . 480
18.34.3. Custom events . . . 481
18.34.4. Integrating with tag managers . . . 484
18.34.5. Cookie/session handling . . . 485
18.34.6. JavaScript API . . . 486
18.35. Implementing a JavaScript Tracker . . . 486
18.35.1. Prerequisites . . . 486
18.35.2. Basic Tracker Structure . . . 487
18.35.3. Cookie Management . . . 488
18.35.4. Session Management . . . 489
18.35.5. Making Context Requests . . . 490
18.35.6. Retrieving Profile and Session Properties . . . 492
18.35.7. Sending Events . . . 493
18.35.8. Using Filters . . . 496
18.35.9. Using Personalization . . . 497
18.35.10. Complete Example: Page Load with Personalization . . . 499
18.35.11. Advanced Features . . . 502
18.35.12. Complete Tracker Implementation . . . 507
18.35.13. Best Practices . . . 511
18.35.14. UUID Generation Security . . . 512
18.35.15. Next Steps . . . 514

19. Integration samples . . . 515
19.1. Samples . . . 515
19.2. Login sample . . . 515
19.2.1. Why server-side? . . . 515
19.2.2. How it works . . . 515
19.2.3. Never forward client-supplied identifiers . . . 516
19.2.4. Build . . . 517
19.2.5. Set up . . . 517
19.2.6. Test profile merge . . . 518
19.2.7. Related . . . 519
19.3. Twitter sample . . . 519
19.3.1. Overview . . . 519
19.3.2. Interacting with the context server . . . 520
19.3.3. Retrieving context information from Unomi using the context servlet . . . 520
19.4. Example . . . 521
19.4.1. HTML page . . . 521
19.4.2. Javascript . . . 521
19.5. Conclusion . . . 533
19.6. Annex . . . 533
19.7. Weather update sample . . . 535
19.7.1. Prerequisites . . . 535
19.7.2. Install the extension . . . 535
19.7.3. Configure the OpenWeatherMap API key . . . 535
19.7.4. What it updates . . . 535
19.7.5. Related . . . 535
20. Connectors . . . 537
20.1. Connectors . . . 537
20.1.1. Call for contributors . . . 537
20.2. Salesforce Connector . . . 537
20.2.1. Getting started . . . 537
20.2.2. Properties . . . 539
20.2.3. Hot-deploying updates to the Salesforce connector (for developers) . . . 539
20.2.4. Using the Salesforce Workbench for testing REST API . . . 540
20.2.5. Setting up Streaming Push queries . . . 540
20.2.6. Executing the unit tests . . . 540
21. Developers . . . 542
21.1. Building . . . 542
21.1.1. Initial Setup . . . 542
21.1.2. Building . . . 542
21.1.3. Using the build.sh script . . . 543
21.1.4. Updating the website . . . 545
21.1.5. JGitFlow . . . 546
21.1.6. Installing a Search Engine . . . 546
21.1.7. Starting Unomi . . . 547
21.1.8. Deploying the generated binary package . . . 548
21.1.9. Deploying into an existing Karaf server . . . 549
21.1.10. Installing GraphViz for Manual Generation . . . 549
21.1.11. JDK Selection on Mac OS X . . . 550
21.1.12. Running the integration tests . . . 551
21.1.13. Testing with an example page . . . 552
21.2. Platform upgrades . . . 552

21.2.1. Overview and shared rules . . . 552
21.2.2. JIRA reference (platform upgrades) . . . 558
21.2.3. Upgrading Apache Karaf . . . 561
21.2.4. Upgrading the JDK . . . 568
21.2.5. CXF, Jackson, and Jetty (Karaf cascade libraries) . . . 570
21.2.6. Upgrading the Elasticsearch client . . . 573
21.2.7. Upgrading the OpenSearch client . . . 575
21.2.8. Upgrading Log4j . . . 577
21.2.9. Integration tests and Docker . . . 579
21.2.10. Dependency hygiene (non-platform bumps) . . . 585
21.3. SSH Shell Commands . . . 587
21.3.1. Using the shell . . . 587
21.3.2. Lifecycle commands . . . 588
21.3.3. Runtime commands . . . 589
21.3.4. Tenant commands . . . 592
21.3.5. Scheduler commands . . . 593
21.4. Writing plugins . . . 594
21.4.1. Prerequisites . . . 594
21.4.2. Types vs instances . . . 595
21.4.3. META-INF/cxs/ layout . . . 595
21.4.4. OSGi service registration (Declarative Services) . . . 596
21.4.5. Creating a Maven plugin project . . . 597
21.4.6. Deployment and updates . . . 599
21.4.7. JSON-only extensions . . . 599
21.4.8. Custom action types . . . 601
21.4.9. Custom condition types . . . 602
21.4.10. Multi-bundle layout for Elasticsearch and OpenSearch . . . 605
21.4.11. Additional bundle resources . . . 607
21.4.12. Custom value types and validation . . . 607
21.4.13. Scheduling background work . . . 607
21.4.14. GraphQL provider extensions . . . 607
21.4.15. Reference plugins in the codebase . . . 607
21.4.16. Best practices . . . 608
21.4.17. Reference implementations in the core codebase . . . 608
21.5. Migration patches . . . 609


THE
APACHE
SOFTWARE FOUNDATION
apache.org

Chapter 1. Foreword

This documentation provides a comprehensive guide to Apache Unomi 4.x, a powerful and flexible Customer Data Platform (CDP) that enables organizations to collect, manage, and leverage customer data for personalization and marketing automation.

Apache Unomi is built on open standards and provides a robust foundation for building customer-centric applications. Whether you're implementing personalization, running marketing campaigns, or building customer profiles, this documentation will help you understand and effectively use Apache Unomi's capabilities.

This guide covers everything from getting started with your first installation to advanced topics like plugin development, migration strategies, and integration patterns. We recommend starting with the "Discover Unomi" section if you're new to the platform, and then exploring the specific topics that match your needs.

The Apache Unomi community is committed to providing comprehensive, accurate, and up-to-date documentation. We welcome feedback and contributions to help improve this documentation for all users.

Happy reading!

This edition documents Apache Unomi 4.x, which requires Java 17, Elasticsearch 9.x or OpenSearch 3.x, and tenant API keys for public endpoints.

Chapter 2. What's new

2.1. What's new in Apache Unomi 4.0

Apache Unomi 4.0 builds on the 3.0 platform (Elasticsearch 9 client, Karaf 4.4, Java 17) with operator-focused features for multi-tenant production deployments.

2.1.1. Multi-tenancy and API keys

Complete tenant isolation for profiles, events, segments, rules, and schemas. Public endpoints (for example /cxs/context.json) require a tenant public API key; administrative work uses tenant private keys or system administrator credentials.

- Operator guide: Multi-tenancy
- Migration: Migrate from 3.0 to 4.0 (includes client-facing hardening)
- Migrating Unomi 2 or Unomi 3 clients: single-tenant compatibility mode (singletenantcompatibility.enabled in org.apache.unomi.rest.authentication.cfg)
- Public and GraphQL API key checks no longer read every tenant on each request, and refuse revoked and expired keys (UNOMI-1016)
- JAAS admin requests without X-Unomi-Tenant-Id use a configurable default tenant (default default), so 3.0 admin scripts keep seeing migrated data (UNOMI-1008)
- Health check environment variables UNOMI_HEALTHCHECK_ENABLED, UNOMI_HEALTHCHECK_PROVIDERS and UNOMI_HEALTHCHECK_TIMEOUT work again (UNOMI-1007)

2.1.2. Security hardening (credentials, profile binding, privileged APIs)

Unomi 4.0 requires explicit admin and health-check passwords at startup, treats the profile cookie as the only public profile bearer on /context.json and /eventcollector, defaults the profile cookie to HttpOnly, restricts Groovy Actions and Router import/export REST to system administrator, and gates cross-profile merge / updateProperties / systemProperties writes to trusted callers.

- Details and migration table: Client-facing hardening
- Profile tracking contract: How profile tracking works
- Jira: UNOMI-972

2.1.3. Web tracker 2.0

Unomi 4 serves web tracker 2.0 at /tracker/unomi-web-tracker.js. A page that loads that URL gets the new tracker with the server. A page that installed the apache-unomi-tracker package, or that uses a copied older file, stays on tracker 1.x until that copy is upgraded.

Tracker 1.x cannot send the tenant public API key. On a multi-tenant server, use tracker 2.0.0 or later and set apiKey, or keep single-tenant compatibility mode on. The same configuration still works on

Unomi 2.x, 3.x, and on Unomi 4 in that mode. After the upgrade, clicks that 1.x missed are recorded, and the page's own click handlers run once per click.

- Upgrading the tracker from 1.x to 2.0
- Configuring the tracker for your Unomi server

2.1.4. Visitor cookie across websites

The public tracker endpoints answer any origin, but only listed websites and Unomi's own origin may use the visitor cookie. Websites are listed by the operator for every tenant (UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS, empty by default) or by each tenant for itself (its publicCorsAllowedOrigins property). SameSite is configurable (UNOMI_PROFILE_COOKIE_SAMESITE), and None requires HTTPS. Websites on another subdomain that used the cookie with Unomi 2.x or 3.x must be listed; websites on another site must also use HTTPS with SameSite=None, or serve Unomi from their own origin.

- Setup guide: Websites on other origins
- Upgrading from 2.x or 3.x: compatibility table
- Each tenant can list its own websites in its publicCorsAllowedOrigins property, so tenants created at runtime need no server change: Websites listed by each tenant (UNOMI-1013)
- Jira: UNOMI-1005

2.1.5. Cluster-aware task scheduler

Built-in background job scheduler with persistence, cluster locks, recovery, REST API (/cxs/tasks), and Karaf shell commands.

- Task scheduler

2.1.6. OpenSearch 3 support

Official alternative to Elasticsearch, including secured Docker deployments.

- Configuration: Configuration
- Migration from Elasticsearch: Migrate from Elasticsearch to OpenSearch

2.1.7. Tenant usage and retention

Read-only per-tenant usage metrics and an event retention purge API for upstream control planes.

- REST: /cxs/tenants/{tenantId}/usage and /cxs/tenants/{tenantId}/purge/events (see Useful URLs)

2.1.8. Request tracing

Administrators can trace how context and event-collector requests are processed using explain=true on supported endpoints.

- Using the explain parameter for request tracing

2.1.9. Condition validation

Segments, rules, and queries are validated against condition type definitions at save time. Invalid parameters are rejected before they reach runtime evaluation.

- Condition validation
- Debugging conditions

2.2. What's new in Apache Unomi 3.0

Apache Unomi 3.0 is a major platform release. It upgrades the search-engine client, the OSGi runtime, and the minimum Java version.

2.2.1. Java 17

Unomi 3.0 requires **Java 17** or later. Java 11 is no longer supported.

2.2.2. Karaf 4.4

Karaf was upgraded from 4.2.x to **4.4.11** to support current dependencies and Java 17.

2.2.3. Elasticsearch 9 Java API client

The legacy Elasticsearch REST high-level client was replaced by the official **Elasticsearch 9** Java API client.

- Client reference: https://www.elastic.co/docs/reference/elasticsearch/clients/java
- Data migration from Elasticsearch 7: Migrate from Elasticsearch 7 to Elasticsearch 9

2.2.4. QueryBuilder ID renames

Built-in condition queryBuilder IDs were renamed to remove Elasticsearch-specific suffixes. Custom plugins and JSON definitions must be updated. See the mapping table in Migrate from 2.x to 3.0.

2.2.5. Persistence-based clustering

Cluster coordination moved from Karaf Cellar to a persistence-backed node registry (see Cluster setup).

2.2.6. Health check extension

HTTP health endpoint at /health/check with pluggable providers (Karaf, search engine, persistence, cluster, Unomi bundles). See Health check.

Chapter 3. Discover Unomi

3.1. Quick start with Docker

Begin by creating a docker-compose.yml file. You can choose between Elasticsearch or OpenSearch:

3.1.1. Option 1: Using Elasticsearch

Export the passwords first. The compose files read them from your shell, and the curl commands further down use the same variables, so setting them once here keeps both consistent. Unomi refuses to start if they are unset.

```
export UNOMI_ROOT_PASSWORD='choose-a-strong-password'
export UNOMI_HEALTHCHECK_PASSWORD='choose-a-strong-health-password'
```

```
version: '3.8'
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:9.4.3
    environment:
      - discovery.type=single-node
    ports:
      - 9200:9200
  unomi:
    # Unomi version can be updated based on your needs
    image: apache/unomi:4.0.0
    environment:
      - UNOMI_ELASTICSEARCH_ADDRESSES=elasticsearch:9200
      - UNOMI_THIRDPARTY_PROVIDER1_IPADDRESSES=0.0.0.0/0,::1,127.0.0.1
      - UNOMI_ROOT_PASSWORD=${UNOMI_ROOT_PASSWORD}
      - UNOMI_HEALTHCHECK_PASSWORD=${UNOMI_HEALTHCHECK_PASSWORD}
    ports:
      - 8181:8181
      - 9443:9443
      - 8102:8102
    links:
      - elasticsearch
    depends_on:
      - elasticsearch
```

3.1.2. Option 2: Using OpenSearch

```
version: '3.8'
services:
  opensearch-node1:
    image: opensearchproject/opensearch:3.7.0
    environment:
      - cluster.name=opensearch-cluster
      - node.name=opensearch-node1
      - discovery.type=single-node
      - bootstrap.memory_lock=true
      - "OPENSEARCH_JAVA_OPTS=-Xms512m -Xmx512m"
      - OPENSEARCH_INITIAL_ADMIN_PASSWORD=${OPENSEARCH_INITIAL_ADMIN_PASSWORD:-
MyStrongPassw0rd!}
    ulimits:
      memlock:
        soft: -1
        hard: -1
      nofile:
        soft: 65536
        hard: 65536
    volumes:
      - opensearch-data1:/usr/share/opensearch/data
    ports:
      - 9200:9200
      - 9600:9600

  unomi:
    image: apache/unomi:4.0.0
    environment:
      - UNOMI_DISTRIBUTION=unomi-distribution-opensearch
      - UNOMI_OPENSEARCH_ADDRESSES=opensearch-node1:9200
      - UNOMI_OPENSEARCH_USERNAME=admin
      - UNOMI_OPENSEARCH_PASSWORD=${OPENSEARCH_INITIAL_ADMIN_PASSWORD:-
MyStrongPassw0rd!}
      # Package default sslEnable is true; trust-all matches typical local OpenSearch Docker TLS
      - UNOMI_OPENSEARCH_SSL_ENABLE=true
      - UNOMI_OPENSEARCH_SSL_TRUST_ALL_CERTIFICATES=true
      - UNOMI_HEALTHCHECK_PROVIDERS=cluster,opensearch,unomi,persistence
      - UNOMI_ROOT_PASSWORD=${UNOMI_ROOT_PASSWORD}
      - UNOMI_HEALTHCHECK_PASSWORD=${UNOMI_HEALTHCHECK_PASSWORD}
    ports:
      - 8181:8181
      - 9443:9443
      - 8102:8102
    depends_on:
      - opensearch-node1

volumes:
  opensearch-data1:
```

From the same folder, start the environment using docker-compose up and wait for the startup to complete.

3.1.3. After startup (Elasticsearch Docker path)

Once Unomi is running, create a tenant, then **regenerate** API keys and save the plainTextKey values from those responses (tenant create only returns masked keys):

```
curl -X POST http://localhost:8181/cxs/tenants \
 --user "karaf:${UNOMI_ROOT_PASSWORD}" \
 -H "Content-Type: application/json" \
 -d '{
  "requestedId": "default",
  "properties": {
   "name": "Default Tenant",
   "description": "Default tenant for quick start"
  }
 }'

curl -X POST "http://localhost:8181/cxs/tenants/default/apikeys?type=PUBLIC" --user
"karaf:${UNOMI_ROOT_PASSWORD}"
curl -X POST "http://localhost:8181/cxs/tenants/default/apikeys?type=PRIVATE" --user
"karaf:${UNOMI_ROOT_PASSWORD}"
```

Use the public API key in X-Unomi-Api-Key for /cxs/context.json requests. See Multi-tenancy.

Try accessing https://localhost:9443/cxs/cluster with username/password: karaf / your UNOMI_ROOT_PASSWORD . You might get a certificate warning in your browser, just accept it despite the warning it is safe.

3.2. Quick Start manually

3.2.1. Option 1: Using Elasticsearch

1) Install JDK 17 and make sure you set the JAVA_HOME variable (see our Getting Started guide for more information on JDK compatibility)

2) Download Elasticsearch here : https://www.elastic.co/downloads/past-releases/elasticsearch-9-4-3 (please **make sure** you use the proper version : 9.4.3)

3) Uncompress it and change the config/elasticsearch.yml to include the following config :

```
cluster.name: contextElasticsearch
```

4) Launch Elasticsearch using : bin/elasticsearch

3.2.2. Option 2: Using OpenSearch

1) Install JDK 17 as described above

2) Download OpenSearch here: https://opensearch.org/downloads.html (please **make sure** you use

version **3.7.0**, matching opensearch.version in the Unomi root POM)

3) Uncompress it and change the config/opensearch.yml to include the following config:

```
cluster.name: opensearch-cluster
discovery.type: single-node
```

4) Launch OpenSearch using: bin/opensearch

3.2.3. Complete the Setup

5) Download Apache Unomi here : https://unomi.apache.org/download.html

5b) Before starting Karaf, export required passwords (Unomi will refuse to start without them):

```
export UNOMI_ROOT_PASSWORD='choose-a-strong-password'
export UNOMI_HEALTHCHECK_PASSWORD='choose-a-strong-health-password'
```

6) Start it using : ./bin/karaf

7) Start the Apache Unomi packages using: - unomi:start

Note that you could change or set the distribution (if not defined using environment variable) using: - unomi:setup -d=unomi-distribution-opensearch -f=true - unomi:start

The unomi:setup parameter specifies which distribution to use (unomi-distribution-elasticsearch or unomi-distribution-opensearch), which determines which set of features and bundles are installed and started. A custom distribution can also be used if available in Karaf (by adding the relevant feature repository)

8) Wait for startup to complete

9) Try accessing https://localhost:9443/cxs/cluster with username/password: karaf / your UNOMI_ROOT_PASSWORD . You might get a certificate warning in your browser, just accept it despite the warning it is safe.

10) Create a tenant that will own all your data, then regenerate keys and store plainTextKey:

```
curl -X POST http://localhost:8181/cxs/tenants \
 --user "karaf:${UNOMI_ROOT_PASSWORD}" \
 -H "Content-Type: application/json" \
 -d '{
   "requestedId": "default",
   "properties": {
    "name": "Default Tenant",
    "description": "Default tenant for quick start"
   }
 }'

curl -X POST "http://localhost:8181/cxs/tenants/default/apikeys?type=PUBLIC" --user
"karaf:${UNOMI_ROOT_PASSWORD}"
curl -X POST "http://localhost:8181/cxs/tenants/default/apikeys?type=PRIVATE" --user
"karaf:${UNOMI_ROOT_PASSWORD}"
```

Save the plainTextKey values from the key-creation responses — you'll need them for API calls.

11) Request your first context:

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
 -H "Content-Type: application/json" \
 -H "X-Unomi-Api-Key: <YOUR_PUBLIC_API_KEY>" \
 -d '{
   "source": {
    "itemId": "homepage",
    "itemType": "page",
    "scope": "default"
   }
 }'
```

Or access it directly in a browser (requires API key in header): http://localhost:8181/cxs/context.js?sessionId=1234



For the context request to work, you'll need to include the public API key from step 10 in the request header: X-Unomi-Api-Key: <YOUR_PUBLIC_API_KEY>

12) If something goes wrong, you should check the logs in ./data/log/karaf.log. If you get errors on the search engine, make sure you are using the proper version.

Next steps:

- Trying our integration samples page

Note: When using OpenSearch, make sure to: - Align SSL with the cluster (UNOMI_OPENSEARCH_SSL_ENABLE / org.apache.unomi.opensearch.sslEnable; package default is true) - For local Docker TLS, set UNOMI_OPENSEARCH_SSL_TRUST_ALL_CERTIFICATES=true or install a proper trust store - Configure a strong admin password via OPENSEARCH_INITIAL_ADMIN_PASSWORD (OpenSearch rejects weak passwords such as admin) - Select the OpenSearch distribution

(UNOMI_DISTRIBUTION=unomi-distribution-opensearch or unomi:setup -d=unomi-distribution-opensearch)

3.3. Getting started with Unomi

We will first get you up and running with an example. We will then lift the corner of the cover somewhat and explain in greater details what just happened.

3.3.1. Prerequisites

This document assumes working knowledge of git to be able to retrieve the code for Unomi and the example. Additionally, you will require a working Java 17 or above install. Refer to http://www.oracle.com/technetwork/java/javase/ for details on how to download and install Java SE 17 or greater.

JDK compatibility

Starting with Java 9, Oracle made some big changes to the Java platform releases. This is why Apache Unomi is focused on supporting the Long Term Supported versions of the JDK, currently version 17. We do not test with intermediate versions so they may or may not work properly. Currently the most tested version is version 17.

Also, as there are new licensing restrictions on JDKs provided by Oracle for production usages, Apache Unomi has also added support for OpenJDK builds. Other JDK distributions might also work but are not regularly tested so you should use them at your own risks.

Search Engine Compatibility

Apache Unomi supports two search engine backends:

- **Elasticsearch**: Version **9.x** (for example 9.4.3) is required for Unomi 3.0 and later. Version 7.17.5 applies only to Unomi 2.x.
- **OpenSearch**: Version **3.7.0** (matching opensearch.version in the Unomi root POM) is supported starting with Unomi 4.0

It is highly recommended to use the versions specified in the documentation. When in doubt, consult the Apache Unomi community for the latest compatibility information.

Note for OpenSearch users: - Security is enabled by default and requires SSL/TLS - Default admin credentials are required (username: admin) - The initial admin password can be configured via environment variable

3.3.2. Running Unomi

Set the admin and health passwords (required)

Unomi does not ship known default passwords. Set both **before** starting the server (otherwise startup fails with a clear error):

```
export UNOMI_ROOT_PASSWORD='choose-a-strong-password'
export UNOMI_HEALTHCHECK_PASSWORD='choose-a-strong-health-password'
```

Authenticate later with karaf:$UNOMI_ROOT_PASSWORD. See the configuration chapter if you prefer to set org.apache.unomi.security.root.password / org.apache.unomi.healthcheck.password in etc/custom.system.properties instead.

Start Unomi

Start Unomi according to the quick start with docker or by compiling using the building instructions. Once you have Karaf running, you should wait until you see the following messages on the Karaf console:

```
Initializing user list service endpoint...
Initializing geonames service endpoint...
Initializing segment service endpoint...
Initializing scoring service endpoint...
Initializing campaigns service endpoint...
Initializing rule service endpoint...
Initializing profile service endpoint...
Initializing cluster service endpoint...
```

This indicates that all the Unomi services are started and ready to react to requests.

Create a tenant (using the password you set before startup):

```
curl -X POST http://localhost:8181/cxs/tenants \
 --user "karaf:${UNOMI_ROOT_PASSWORD}" \
 -H "Content-Type: application/json" \
 -d '{
  "requestedId": "default",
  "properties": {
   "name": "Default Tenant",
   "description": "Default tenant for getting started"
  }
 }'
```

The tenant create response includes **masked** API key metadata only. Regenerate keys to obtain one-time plainTextKey values:

```
curl -X POST "http://localhost:8181/cxs/tenants/default/apikeys?type=PUBLIC" --user
"karaf:${UNOMI_ROOT_PASSWORD}"
curl -X POST "http://localhost:8181/cxs/tenants/default/apikeys?type=PRIVATE" --user
"karaf:${UNOMI_ROOT_PASSWORD}"
```

Store plainTextKey immediately — you'll need the public key for subsequent API calls. See Multi-tenancy for authentication details.

Now you can:

1. Access the RESTful services list: http://localhost:8181/cxs
2. Retrieve an initial context:

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
 -H "Content-Type: application/json" \
 -H "X-Unomi-Api-Key: <YOUR_PUBLIC_API_KEY>" \
 -d '{
  "source": {
   "itemId": "homepage",
   "itemType": "page",
   "scope": "default"
  }
 }'
```



This request will automatically create a profile and session if they don't exist, and return a cookie with the profile ID. See How profile tracking works for details.

+ 3. View the introduction page: http://localhost:8181

Also now that your service is up and running you can go look at the request examples to learn basic requests you can do once your server is up and running.

3.4. Unomi web tracking tutorial

In this tutorial we will guide through the basic steps of getting started with a web tracking project. You will see how to integrate the built-in web tracker with an existing web site and what this enables.

If you prefer to use existing HTML and Javascript rather than building your own, all the code we feature in this tutorial is extracted from our tracker sample which is available here: https://github.com/apache/unomi/blob/master/extensions/web-tracker/wab/src/main/webapp/index.html . However you will still need to use the REST API calls to create the scope and rule to make it all work.



On Unomi **4.0**, /cxs/context.json and /cxs/eventcollector require a tenant **public** API key. Set it as apiKey in the tracker's wemInitConfig, as in the snippets below (tracker 2.0.0 or later). Remove the apiKey line for Unomi 2.x or 3.x, or if you enable single-tenant compatibility mode. See Configuring the tracker for your Unomi server. To obtain the key, create a tenant and regenerate its keys (see Multi-tenancy). If the page already runs tracker 1.x, read Upgrading the tracker from 1.x to 2.0 before you replace it.

Create a scope and rules with tenant private-key or JAAS admin authentication as shown below.

3.4.1. Installing the web tracker in a web page

Using the built-in tracker is pretty simple, simply add the following code to your HTML page :

```
<script type="text/javascript" src="/tracker/unomi-web-tracker.min.js"></script>
```

or you can also use the non-minified version that is available here:

```
<script type="text/javascript" src="/tracker/unomi-web-tracker.js"></script>
```

This will only load the tracker. To initialize it use a snippet like the following code:

```
<script type="text/javascript">
    (function () {
        const unomiTrackerTestConf = {
            "scope": "unomi-tracker-test",
            "site": {
                "siteInfo": {
                    "siteID": "unomi-tracker-test"
                }
            },
            "page": {
                "pageInfo": {
                    "pageID": "unomi-tracker-test-page",
                    "pageName": document.title,
                    "pagePath": document.location.pathname,
                    "destinationURL": document.location.origin + document.location.pathname,
                    "language": "en",
                    "categories": [],
                    "tags": []
                },
                "consentTypes": []
            },
            "events": [],
            "wemInitConfig": {
                "contextServerUrl": document.location.origin,
                "timeoutInMilliseconds": "1500",
                "contextServerCookieName": "context-profile-id",
                "activateWem": true,
                "trackerSessionIdCookieName": "unomi-tracker-test-session-id",
                "trackerProfileIdCookieName": "unomi-tracker-test-profile-id",
                // Unomi 4 multi-tenant only: the tenant's public API key
                "apiKey": "YOUR_PUBLIC_API_KEY"
            }
        }

        // generate a new session
        if
(unomiWebTracker.getCookie(unomiTrackerTestConf.wemInitConfig.trackerSessionIdCookieName)
== null) {

unomiWebTracker.setCookie(unomiTrackerTestConf.wemInitConfig.trackerSessionIdCookieName,
unomiWebTracker.generateGuid(), 1);
        }

        // init tracker with our conf
        unomiWebTracker.initTracker(unomiTrackerTestConf);

        unomiWebTracker._registerCallback(() => {
            console.log("Unomi tracker test successfully loaded context",
unomiWebTracker.getLoadedContext());
        }, 'Unomi tracker test callback example');

        // start the tracker
        unomiWebTracker.startTracker();
    })();
</script>
```

3.4.2. Creating a scope to collect the data

You might notice the scope used in the snippet. All events sent to Unomi must be associated with a scope, that must have been created before events are accepted. So in order to make sure the events are collected with the above Javascript code, we must create a scope with the following request.

```
curl --location --request POST 'http://localhost:8181/cxs/scopes' \
 --user "TENANT_ID:PRIVATE_KEY" \
 --header 'Content-Type: application/json' \
 --data-raw '{
   "itemId": "unomi-tracker-test",
   "metadata": {
     "id": "unomi-tracker-test",
     "name": "Unomi tracker Test Scope"
   }
 }'
```



Replace TENANT_ID and PRIVATE_KEY with your actual tenant ID and private API key. Only the Tenant API (/cxs/tenants) uses system administrator authentication (the karaf user and your configured UNOMI_ROOT_PASSWORD). Unomi ships no default administrator password: set UNOMI_ROOT_PASSWORD (and UNOMI_HEALTHCHECK_PASSWORD) before starting the server, or it will refuse to start.

3.4.3. Using tracker in your own JavaScript projects

The tracker also exists as an NPM library that you can integrate with your own Javascript projects. You can find the library here:

```
https://www.npmjs.com/package/apache-unomi-tracker
```

Here's an example on how to use it:

```
yarn add apache-unomi-tracker
```

You can then simply use it in your JS code using something like this:

```
import {useTracker} from "apache-unomi-tracker";

(function () {
   const unomiWebTracker = useTracker();
   const unomiTrackerTestConf = {
      "scope": "unomi-tracker-test",
      "site": {
         "siteInfo": {
            "siteID": "unomi-tracker-test"
         }
      },
      "page": {
         "pageInfo": {
            "pageID": "unomi-tracker-test-page",
            "pageName": document.title,
            "pagePath": document.location.pathname,
            "destinationURL": document.location.origin + document.location.pathname,
            "language": "en",
            "categories": [],
            "tags": []
         },
         "consentTypes": []
      },
      "events": [],
      "wemInitConfig": {
         "contextServerUrl": document.location.origin,
         "timeoutInMilliseconds": "1500",
         "contextServerCookieName": "context-profile-id",
         "activateWem": true,
         "trackerSessionIdCookieName": "unomi-tracker-test-session-id",
         "trackerProfileIdCookieName": "unomi-tracker-test-profile-id",
         // Unomi 4 multi-tenant only: the tenant's public API key
         "apiKey": "YOUR_PUBLIC_API_KEY"
      }
   }

   // generate a new session
   if
(unomiWebTracker.getCookie(unomiTrackerTestConf.wemInitConfig.trackerSessionIdCookieName)
== null) {

unomiWebTracker.setCookie(unomiTrackerTestConf.wemInitConfig.trackerSessionIdCookieName,
unomiWebTracker.generateGuid(), 1);
   }

   // init tracker with our conf
   unomiWebTracker.initTracker(unomiTrackerTestConf);

   unomiWebTracker._registerCallback(() => {
      console.log("Unomi tracker test successfully loaded context",
unomiWebTracker.getLoadedContext());
   }, 'Unomi tracker test callback example');

   // start the tracker
   unomiWebTracker.startTracker();
})();
```

3.4.4. Viewing collected events

There are multiple ways to view the events that were received. For example, you could use the following cURL request:

```
curl --location --request POST 'http://localhost:8181/cxs/events/search' \
 --user "TENANT_ID:PRIVATE_KEY" \
 --header 'Content-Type: application/json' \
 --data-raw '{
   "sortby" : "timeStamp:desc",
   "condition" : {
     "type" : "matchAllCondition"
   }
 }'
```

Another (powerful) way to look at events is to use the SSH Console. You can connect to it with the following shell command:

```
  ssh -p 8102 karaf@localhost
```

Using the same credentials (the karaf user and your configured UNOMI_ROOT_PASSWORD) and then you can use command such as :

- event-tail to view in realtime the events as they come in (CTRL+C to stop)
- tenant-set default then crud list event to view recent events for that tenant
- crud read event EVENT_ID to view the details of a specific event

3.4.5. Viewing the current profile

By default, Unomi uses a cookie called context-profile-id to keep track of the current profile. You can use this the value of this cookie which contains a UUID to lookup the details of the profile. For example with the SSH console you can simply do:

```
tenant-set default
crud read profile PROFILE_UUID
```

Which will print out the details of the profile with the associated ID. Another interesting command is crud list profile to list recently modified profiles for the current tenant

You could also retrieve the profile details using the REST API by using a request such as this one:

```
curl --location --request GET 'http://localhost:8181/cxs/profiles/PROFILE_UUID' \
--user "TENANT_ID:PRIVATE_KEY"
```

3.4.6. Adding a rule

Rules are a powerful ways to react in real-time to incoming events. For example a rule could update a profile when a certain event comes in, either copying values from the event or performing some kind of computation when the event occurs, including accessing remote systems such as a Salesforce CRM (see the Salesforce connector sample).

In this example we will simply setup a basic rule that will react to the view event and set a property in the current profile.

```
curl --location --request POST 'http://localhost:8181/cxs/rules' \
--user "TENANT_ID:PRIVATE_KEY" \
--header 'Content-Type: application/json' \
--data-raw '{
   "metadata": {
      "id": "viewEventRule",
      "name": "View event rule",
      "description": "Increments a property on a profile to indicate that this rule executed successfully
when a view event occurs"
   },
   "condition": {
      "type": "eventTypeCondition",
      "parameterValues": {
         "eventTypeId": "view"
      }
   },
   "actions": [
      {
         "type": "incrementPropertyAction",
         "parameterValues": {
            "propertyName": "pageViewCount"
         }
      }
   ]
}'
```

The above rule will execute when a view event is received (which is automatically sent by the tracker when a page is loaded) and increments a property called pageViewCount on the user's profile.

You can then reload the page and check with crud read profile PROFILE_UUID that the profile was updated with the new property and that it is incremented on each page reload.

You can also use crud list rule to display the rules for the current tenant and rule-tail to watch in real-time which rules are executed. crud read rule RULE_ID shows the contents of a rule.

3.4.7. Adding personalization

The last step is to use the newly added property to the profile to perform some page personalization. In order to do that we will use the tracker's API to register a personalization that will be using a condition that checks if the pageViewCount is higher than 5. If it has, variant1 will be displayed, otherwise the fallback variant variant2 will be used instead.

```
variants = {
   "var1" : {
      content : "variant1",
   },
   "var2" : {
      content : "variant2",
   }
}
unomiWebTracker.registerPersonalizationObject({
   "id": "testPersonalization",
   "strategy": "matching-first",
   "strategyOptions": {"fallback": "var2"},
   "contents": [{
      "id": "var1",
      "filters": [{
         "condition": {
            "type": "profilePropertyCondition",
            "parameterValues": {
               "propertyName" : "properties.pageViewCount.<scope>",
               "comparisonOperator" : "greaterThan",
               "propertyValueInteger" : 5
            }
         }
      }]
   }, {
      "id": "var2"
   }]
}, variants, false, function (successfulFilters, selectedFilter) {
   if (selectedFilter) {
      document.getElementById(selectedFilter.content).style.display = '';
   }
});
```

As you can see in the above code snippet, a variants array is created with two objects that associated personalization IDs with content IDs. Then we build the personalization object that contains the two IDs and their associated conditions (only a condition on var1 is passed in this case) as well as an option to indicate which is the fallback variant in case no conditions are matched.

The HTML part of this example looks like this:

```
<div id="variant1" style="display: none">
   You have already seen this page 5 times
</div>
<div id="variant2" style="display: none">
   Welcome. Please reload this page 5 times until it triggers the personalization change
</div>
```

As you can see we hide the variants by default so that there is no "flashing" effect and then use the callback function to display to variant resolve by Unomi's personalization engine.

3.4.8. Conclusion

What have we achieved so far ?

- Installed a tracker in a web page
- Created a scope in which to collect the data
- Learned how to use the tracker as an NPM library
- How to view the collected events
- How to view the current visitor profile
- How to add a rule to update a profile property
- How to personalize a web page's content based on the property updated by the rule

Of course this tutorial is just one example of what could be achieved, and hasn't even yet introduced more advanced notions such as profile segmentation or Groovy action scripting. The system is capable of much more, for example by directly using its actions to integrate with third-party systems (CRM, social networks, etc..)

3.4.9. Next steps

- Learn more about the web tracker, custom events, API, ...
- Learn more about segmentation
- View some more samples
- Continue reading Unomi's user manual to see all that is possible with this technology

Chapter 4. Apache Unomi Recipes and requests

4.1. Recipes

4.1.1. Introduction

In this section of the documentation we provide quick recipes focused on helping you achieve a specific result with Apache Unomi.

4.1.2. Enabling debug mode

Although the examples provided in this documentation are correct (they will work "as-is"), you might be tempted to modify them to fit your use case, which might result in errors.

The best approach during development is to enable Apache Unomi debug mode, which will provide you with more detailed logs about events processing.

The debug mode can be activated via the karaf SSH console (user karaf, with the password you set in UNOMI_ROOT_PASSWORD — no default is shipped):

```
ubuntu@ip-10-0-3-252:~/$ ssh -p 8102 karaf@localhost
Password authentication
Password:
        __ __                  ____
       / //_/____ _____ ___  _/ __/
      / ,<  / __ `/ ___/ __ `/ /_
     / /| |/ /_/ / /  / /_/ / __/
    /_/ |_|\__,_/_/   \__,_/_/

  Apache Karaf (4.2.15)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current session.

karaf@root()> log:set DEBUG org.apache.unomi.schema.impl.SchemaServiceImpl
```

You can then either watch the logs via your preferred logging mechanism (docker logs, log file, ...) or simply tail the logs to the terminal you used to enable debug mode.

```
karaf@root()> log:tail
08:55:28.128 DEBUG [qtp1422628821-128] Schema validation found 2 errors while validating against schema: https://unomi.apache.org/schemas/json/events/view/1-0-0
08:55:28.138 DEBUG [qtp1422628821-128] Validation error: There are unevaluated properties at following paths $.source.properties
08:55:28.140 DEBUG [qtp1422628821-128] Validation error: There are unevaluated properties at following paths $.source.itemId, $.source.itemType, $.source.scope, $.source.properties
08:55:28.142 ERROR [qtp1422628821-128] An event was rejected - switch to DEBUG log level for more information
```

The example above shows schema validation failure at the $.source.properties path. Note that the validation will output one log line for the exact failing path and a log line for its parent, therefore to find the source of a schema validation issue it's best to start from the top.

4.1.3. How to read a profile

The simplest way to retrieve profile data for the current profile is to simply send a request to the /cxs/context.json endpoint. However you will need to send a body along with that request. Here's an example:



The /cxs/context.json endpoint automatically creates or loads profiles and sessions as needed. See How profile tracking works for complete details about the automatic profile and session creation process.

Here is an example that will retrieve all the session and profile properties, as well as the profile's segments and scores

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
--data-raw '{
   "source": {
      "itemId":"homepage",
      "itemType":"page",
      "scope":"example"
   },
   "requiredProfileProperties":["*"],
   "requiredSessionProperties":["*"],
   "requireSegments":true,
   "requireScores":true
}'
```

The requiredProfileProperties and requiredSessionProperties are properties that take an array of property names that should be retrieved. In this case we use the wildcard character '*' to say we want to retrieve all the available properties. The structure of the JSON object that you should send is a JSON-serialized version of the ContextRequest Java class.

Note that it is also possible to access a profile's data through the /cxs/profiles/ endpoint but that really should be reserved to administrative purposes. All public accesses should always use the

/cxs/context.json endpoint for consistency and security.

4.1.4. How to update a profile from the public internet

Before we get into how to update a profile directly from a request coming from the public internet, we'll quickly talk first about how NOT to do it, because we often see users using the following anti-patterns.

How NOT to update a profile from the public internet

Please avoid using the /cxs/profile endpoint. This endpoint was initially the only way to update a profile but it has multiple issues:

- it requires authenticated access. The temptation can be great to use this endpoint because it is simple to access but the risk is that developers might include the credentials to access it in non-secure parts of code such as client-side code. Since there is no difference between this endpoint and any other administration-focused endpoints, attackers could easily re-use stolen credentials to wreak havock on the whole platform.
- No history of profile modifications is kept: this can be a problem for multiple reasons: you might want to keep an trail of profile modifications, or even a history of profile values in case you want to understand how a profile property was modified.
- Even when protected using some kind of proxy, potentially the whole profile properties might be modified, including ones that you might not want to be overriden.

Recommended ways to update a profile

Instead you can use the following solutions to update profiles:

- (Preferred) Use you own custom event(s) to send data you want to be inserted in a profile, and use rules to map the event data to the profile. This is simpler than it sounds, as usually all it requires is setting up a simple rule, defining the corresponding JSON schema and you're ready to update profiles using events.
- Use the protected built-in "updateProperties" event. This event is designed to be used for administrative purposes only. Being a protected event type, it is only accepted from an authorized source address — the tenant's authorizedIPs (it is always protected, whatever the tenant's restrictedEventTypes), or in single-tenant compatibility mode an X-Unomi-Peer key valid for that source address. Note that this is an address-based gate: it is independent of the public-versus-trusted caller distinction that governs profile and session binding (see client-facing hardening), and once the event is accepted a public-key caller may only write plain properties.* paths on the current profile; targetId, systemProperties, segments, scores and consents require a trusted caller (system administrator or tenant private key), and identity fields such as itemId are never writable. Refused names are skipped with a WARN log entry. Prefer custom events for public visitors. Again, prefer the custom events solution because as this is a protected event it will require sending trusted credentials, and as Unomi only supports a single key for the moment it could be problematic if the key is intercepted. But at least by using an event you will get the benefits of auditing and historical property modification tracing (see request tracing).

Let's go into more detail about the preferred way to update a profile. Let's consider the following

example of a rule:

```
curl -X POST http://localhost:8181/cxs/rules \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
--data-raw '{
  "metadata": {
    "id": "setContactInfo",
    "name": "Copy the received contact info to the current profile",
    "description": "Copies the contact info received in a custom event called 'contactInfoSubmitted' to
the current profile"
  },
  "raiseEventOnlyOnceForSession": false,
  "condition": {
    "type": "eventTypeCondition",
    "parameterValues": {
      "eventTypeId": "contactInfoSubmitted"
    }
  },
  "actions": [
    {
      "type": "setPropertyAction",
      "parameterValues": {
        "setPropertyName": "properties(firstName)",
        "setPropertyValue": "eventProperty::properties(firstName)",
        "setPropertyStrategy": "alwaysSet"
      }
    },
    {
      "type": "setPropertyAction",
      "parameterValues": {
        "setPropertyName": "properties(lastName)",
        "setPropertyValue": "eventProperty::properties(lastName)",
        "setPropertyStrategy": "alwaysSet"
      }
    },
    {
      "type": "setPropertyAction",
      "parameterValues": {
        "setPropertyName": "properties(email)",
        "setPropertyValue": "eventProperty::properties(email)",
        "setPropertyStrategy": "alwaysSet"
      }
    }
  ]
}'
```

What this rule does is that it listen for a custom event (events don't need any registration, you can simply start sending them to Apache Unomi whenever you like) of type 'contactInfoSubmitted' and it will search for properties called 'firstName', 'lastName' and 'email' and copy them over to the profile with corresponding property names. You could of course change any of the property names to find your needs. For example you might want to prefix the profile properties with the source of the event, such as 'mobileApp:firstName'.

Now that our rule is defined, the next step is to create a scope and a JSON Schema corresponding to the event to be submitted.

We will start by creating a scope called "example" scope:

```
curl --location --request POST 'http://localhost:8181/cxs/scopes' \
--user "TENANT_ID:PRIVATE_KEY" \
--header 'Content-Type: application/json' \
--data-raw '{
"itemId": "example",
"itemType": "scope"
}'
```



Replace TENANT_ID and PRIVATE_KEY with your actual tenant ID and private API key. Only the Tenant API (/cxs/tenants) uses system administrator authentication (the karaf user and your configured UNOMI_ROOT_PASSWORD).

The next step consist in creating a JSON Schema to validate our event.

```
curl --location --request POST 'http://localhost:8181/cxs/jsonSchema' \
--user "TENANT_ID:PRIVATE_KEY" \
--header 'Content-Type: application/json' \
--data-raw '{
   "$id": "https://unomi.apache.org/schemas/json/events/contactInfoSubmitted/1-0-0",
   "$schema": "https://json-schema.org/draft/2019-09/schema",
   "self": {
      "vendor": "org.apache.unomi",
      "name": "contactInfoSubmitted",
      "format": "jsonschema",
      "target": "events",
      "version": "1-0-0"
   },
   "title": "contactInfoSubmittedEvent",
   "type": "object",
   "allOf": [{ "$ref": "https://unomi.apache.org/schemas/json/event/1-0-0" }],
   "properties": {
      "source" : {
       "$ref" : "https://unomi.apache.org/schemas/json/item/1-0-0"
      },
      "target" : {
       "$ref" : "https://unomi.apache.org/schemas/json/item/1-0-0"
      },
      "properties": {
       "type": "object",
       "properties": {
         "firstName": {
           "type": ["null", "string"]
         },
         "lastName": {
           "type": ["null", "string"]
         },
         "email": {
           "type": ["null", "string"]
         }
       }
      }
   },
   "unevaluatedProperties": false
}'
```

You can notice the following in the above schema:

- We are creating a schema of type "events" ("self.target" equals "events")
- The name of this schema is "contactInfoSubmitted", this MUST match the value of the "eventType" field in the event itself (below)
- To simplify our schema declaration, we're referring to an already existing schema (https://unomi.apache.org/schemas/json/item/1-0-0) to validate the "source" and "target" properties. Apache Unomi ships with a set of predefined JSON Schemas, detailed here: https://github.com/apache/unomi/tree/master/extensions/json-schema/services/src/main/resources/META-INF/cxs/schemas.

- "unevaluatedProperties": false indicates that the event should be rejected if it contains any additional metadata.

Finally, send the contactInfoSubmitted event using a request similar to this one:

```
curl -X POST http://localhost:8181/cxs/eventcollector \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
--data-raw '{
   "sessionId" : "1234",
   "events":[
      {
         "eventType":"contactInfoSubmitted",
         "scope": "example",
         "source":{
            "itemType": "site",
            "scope": "example",
            "itemId": "mysite"
         },
         "target":{
            "itemType": "form",
            "scope": "example",
            "itemId": "contactForm"
         },
         "properties" : {
          "firstName": "John",
          "lastName": "Doe",
          "email": "john.doe@acme.com"
         }
      }
   ]
}'
```

The event we just submitted can be retrieved using the following request:

```
curl -X POST http://localhost:8181/cxs/events/search \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
--data-raw '{
  "offset" : 0,
  "limit" : 20,
  "condition" : {
    "type": "eventPropertyCondition",
    "parameterValues" : {
      "propertyName" : "properties.firstName",
      "comparisonOperator" : "equals",
      "propertyValue" : "John"
    }
  }
}'
```

Troubleshooting common errors

There could be two types of common errors while customizing the above requests: * The schema is invalid * The event is invalid

While first submitting the schema during its creation, Apache Unomi will validate it is syntactically correct (JSON) but will not perform any further validation. Since the schema will be processed for the first time when events are submitted, errors might be noticeable at that time.

Those errors are usually self-explanatory, such as this one pointing to an incorrect location for the "firstName" keyword:

```
09:35:56.573 WARN [qtp1421852915-83] Unknown keyword firstName - you should define your own Meta Schema. If the keyword is irrelevant for validation, just use a NonValidationKeyword
```

If an event is invalid, the logs will contain details about the part of the event that did not validate against the schema. In the example below, an extra property "abcd" was added to the event:

```
12:27:04.269 DEBUG [qtp1421852915-481] Schema validation found 1 errors while validating against schema: https://unomi.apache.org/schemas/json/events/contactInfoSubmitted/1-0-0
12:27:04.272 DEBUG [qtp1421852915-481] Validation error: There are unevaluated properties at following paths $.properties.abcd
12:27:04.273 ERROR [qtp1421852915-481] An event was rejected - switch to DEBUG log level for more information
```

4.1.5. How to search for profile events

Sometimes you want to retrieve events for a known profile. You will need to provide a query in the body of the request that looks something like this (and documentation is available in the REST API) :

```
curl -X POST http://localhost:8181/cxs/events/search \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
--data-raw '{
  "offset" : 0,
  "limit" : 20,
  "condition" : {
    "type": "eventPropertyCondition",
    "parameterValues" : {
      "propertyName" : "profileId",
      "comparisonOperator" : "equals",
      "propertyValue" : "PROFILE_ID"
    }
  }
}'
```

where PROFILE_ID is a profile identifier. This will indeed retrieve all the events for a given profile.

4.1.6. How to create a new rule

There are basically two ways to create a new rule :

- Using the REST API
- Packaging it as a predefined rule in a plugin

In both cases the JSON structure for the rule will be exactly the same, and in most scenarios it will be more interesting to use the REST API to create and manipulate rules, as they don't require any development or deployments on the Apache Unomi server.

```
curl -X POST http://localhost:8181/cxs/rules \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
--data-raw '{
 "metadata": {
  "id": "exampleEventCopy",
  "name": "Example Copy Event to Profile",
  "description": "Copy event properties to profile properties"
 },
 "condition": {
    "type": "eventTypeCondition",
    "parameterValues": {
     "eventTypeId" : "myEvent"
    }
 },
 "actions": [
  {
    "parameterValues": {
    },
    "type": "allEventToProfilePropertiesAction"
  }
 ]
}'
```

The above rule will be executed if the incoming event is of type myEvent and will simply copy all the properties contained in the event to the current profile.

4.1.7. How to search for profiles

In order to search for profiles you will have to use the /cxs/profiles/search endpoint that requires a Query JSON structure. Here's an example of a profile search with a Query object:

```
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
--data-raw '{
  "text" : "unomi",
  "offset" : 0,
  "limit" : 10,
  "sortby" : "properties.lastName:asc,properties.firstName:desc",
  "condition" : {
    "type" : "booleanCondition",
    "parameterValues" : {
      "operator" : "and",
      "subConditions" : [
        {
          "type": "profilePropertyCondition",
          "parameterValues": {
            "propertyName": "properties.leadAssignedTo",
            "comparisonOperator": "exists"
          }
        },
        {
          "type": "profilePropertyCondition",
          "parameterValues": {
            "propertyName": "properties.lastName",
            "comparisonOperator": "exists"
          }
        }
      ]
    }
  }
}'
```

In the above example, you search for all the profiles that have the leadAssignedTo and lastName properties and that have the unomi value anywhere in their profile property values. You are also specifying that you only want 10 results beginning at offset 0. The results will be also sorted in alphabetical order for the lastName property value, and then by reverse alphabetical order for the firstName property value.

As you can see, queries can be quite complex. Please remember that the more complex the more resources it will consume on the server and potentially this could affect performance.

4.1.8. Getting / updating consents

You can find information on how to retrieve or create/update consents in the Consent API section.

4.1.9. How to send a login event to Unomi

Tracking logins must be done carefully with Unomi. A login event is considered a "privileged" event and therefore for not be initiated from the public internet. Ideally user authentication should always be validated by a trusted third- party even if it is a well-known social platform such as Facebook or Twitter. Basically what should NEVER be done:

1. Login to a social platform
2. Call back to the originating page
3. Send a login event to Unomi from the page originating the login in step 1

The problem with this, is that any attacker could simply directly call step 3 without any kind of security. Instead the flow should look something like this:

1. Login to a social platform
2. Call back to a special secured system that performs an server-to-server call to send the login event to Apache Unomi using the Unomi key.

For simplicity reasons, in our login example, the first method is used, but it really should never be done like this in production because of the aforementioned security issues. The second method, although a little more involved, is much preferred.

When sending a login event, you can setup a rule that can check a profile property to see if profiles can be merged on an universal identifier such as an email address.

In our login sample we provide an example of such a rule (fired from a **server-side** trusted call — see Login sample). You can find the rule here:

https://github.com/apache/unomi/blob/master/samples/login-integration/src/main/resources/META-INF/cxs/rules/exampleLogin.json

As you can see in this rule, we call an action called :

```
mergeProfilesOnPropertyAction
```

with as a parameter value the name of the property on which to perform the merge (the email). What this means is that upon successful login using an email, Unomi will look for other profiles that have the same email and merge them into a single profile. Because of the merge, this should only be done for authenticated profiles, otherwise this could be a security issue since it could be a way to load data from other profiles by merging their data !

4.1.10. Technical Implementation Details

Profile Selection Strategy

By default, profiles are merged with the oldest profile (determined by the firstVisit property) becoming the master profile. This behavior can be overridden by setting forceEventProfileAsMaster to true in the action parameters, which forces the current event's profile to become the master.

Property Merge Strategy

When merging profile properties, Unomi follows these rules:

1. Properties from all profiles are combined into the master profile
2. When conflicting values exist for a property, the master profile's value takes precedence
3. Multi-valued properties (arrays/collections) are merged together
4. Profile segments and scores are recalculated after the merge

A custom property merge strategy can be implemented by:

1. Creating a class that implements PropertyMergeStrategyType
2. Registering it as an OSGi service
3. Configuring it in your merge action

Asynchronous Processing

To ensure performance, Unomi processes resource-intensive operations asynchronously:

1. The profile merge itself happens synchronously during event processing
2. Session and event reassignment is scheduled asynchronously via the SchedulerService
3. Property updates triggered by the merge may cause delayed rule evaluation

Result Codes

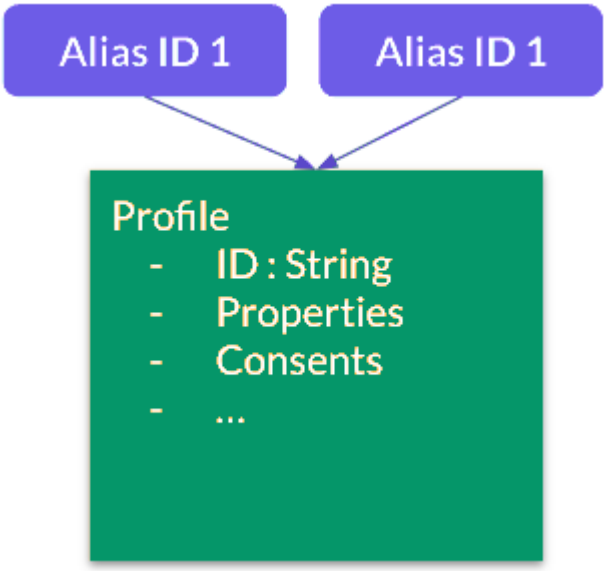
The MergeProfilesOnPropertyAction returns one of the following result codes:

- EventService.NO_CHANGE - No merge was performed
- EventService.PROFILE_UPDATED - A profile was updated during the merge
- EventService.PROFILE_UPDATED + EventService.SESSION_UPDATED - Both profile and session were updated

4.1.11. What profile aliases are and how to use them

Profile aliases make it possible to reference profiles using multiple identifiers. The profile alias object basically contains a link between the alias ID and the profile ID. The itemId of a profile alias is the actual alias ID, which the profileID field contains the reference to the aliased profile.

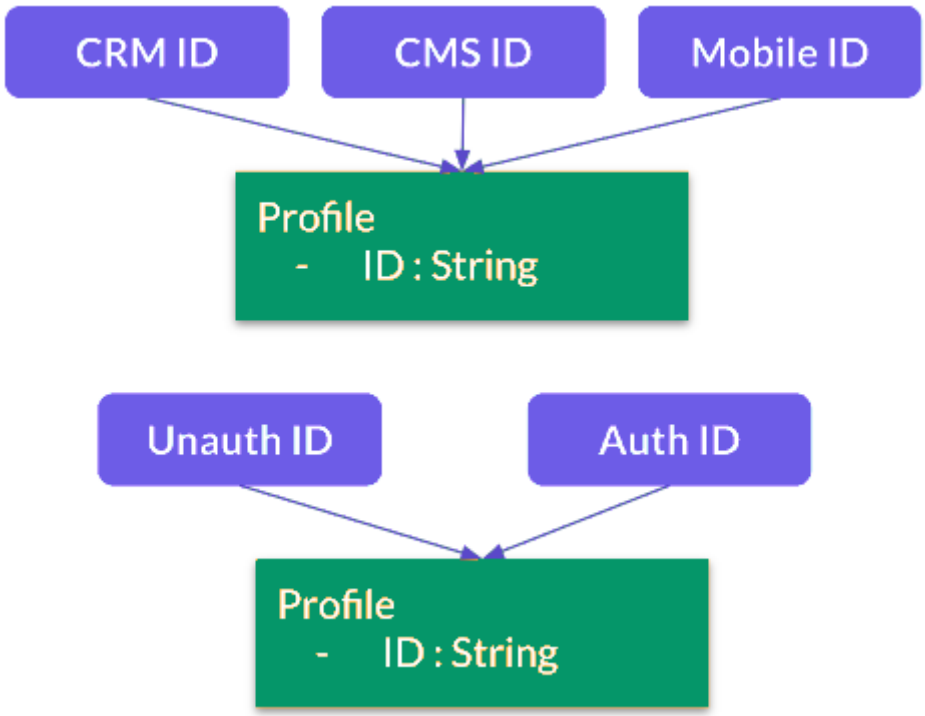
What they are



Profile aliases:

- Make it possible to lookup profiles by main (Unomi) ID or by any other alias ID
- Aliases are just IDs stored in a dedicated index
- A profile may have an unlimited number of aliases attached to it.

How to use them



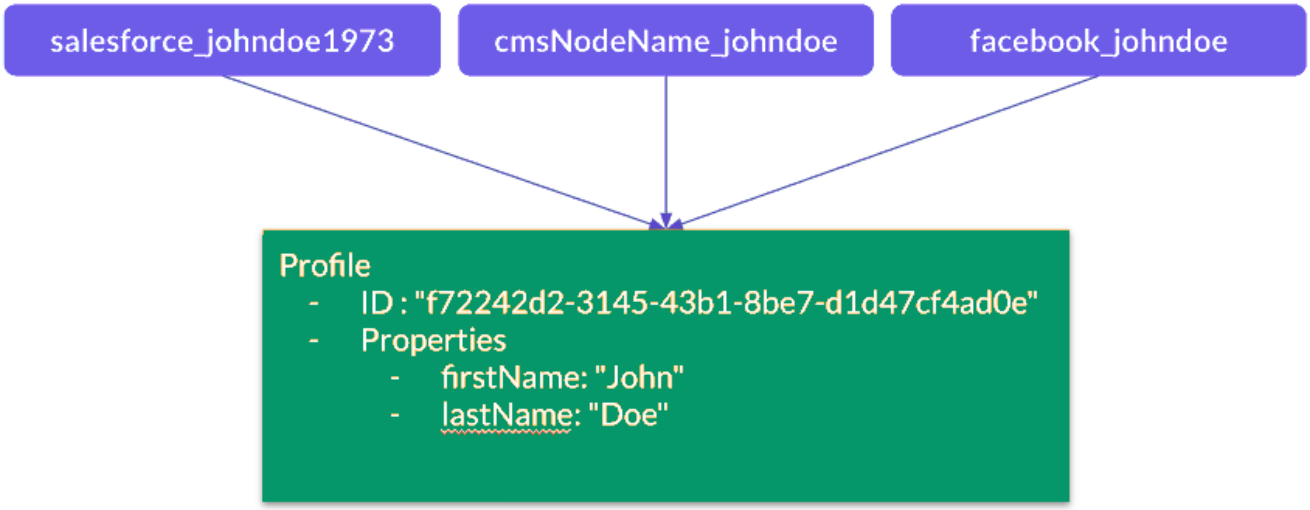
Here are different use cases for profile aliases:

- Connect different systems to Unomi such as a CRM, CMS and native mobile app that all have their own iD for a single customer

- Merging profiles when a visitor is identified
- Adding new IDs at a later time

Example

Here is an example of multiple external aliases pointing to a single Unomi profile



Interactions with merging

Profile merges have been modified to use aliases starting Unomi 2

Upon merge:

- Properties are copied to the master profile as before
- An alias is created for the "master" profile with the ID of the merged profile
- Merged profiles are now deleted
- "mergedWith" property is no longer used since we deleted the merged profiles

API

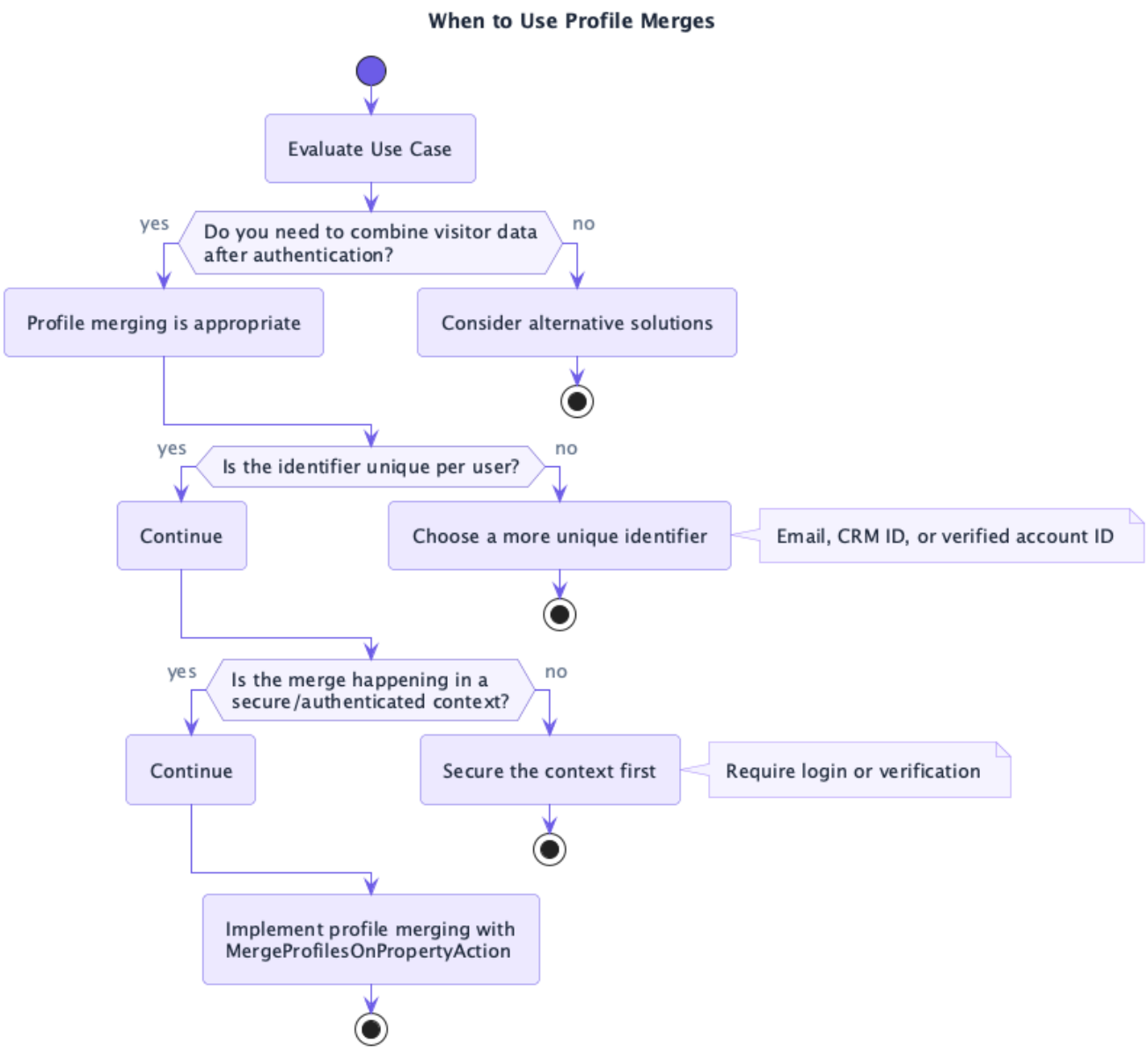
/context.json and /eventcollector look up profiles by profile ID or aliases from the profile cookie (context-profile-id by default). Public callers must present that cookie; body/query profileId is ignored for public callers. Trusted callers (tenant private key / system admin) may still pass profileId explicitly. See How profile tracking works and client-facing hardening.

Verb	Path	Description
GET	/cxs/profiles/PROFILE_ID_OR_ALIAS	Retrieves a profile by ID or Alias ID (useful if an external system wants to get a profile)
GET	/cxs/profiles/PROFILE_ID/aliases	Get all the aliases for a profile

Verb	Path	Description
POST	/cxs/profiles/PROFILE_ID/aliases/ALIAS_ID	Add an alias to a profile
DELETE	/cxs/profiles/PROFILE_ID/aliases/ALIAS_ID	Remove an alias from a profile

4.1.12. Common Issues and Best Practices

Decision Guide for Profile Merging



Troubleshooting Common Issues

Merge Not Happening

If profiles aren't merging as expected, check:

1. The merge property exists on both profiles with exactly matching values
2. The merge property is stored as a system property (systemProperties.mergeIdentifier)
3. The rule containing the merge action is correctly triggered
4. The profiles aren't personas or anonymous profiles (which are skipped)
5. Ensure the maxProfilesInOneMerge limit (default 50) is not being exceeded

Data Loss During Merge

By default, when property values conflict, the master profile's values take precedence. This can appear as data loss from secondary profiles.

Consider:

1. Implementing a custom PropertyMergeStrategy for complex merging logic
2. Pre-processing profiles to ensure critical data is preserved
3. Logging pre-merge state for audit purposes

Performance Issues

Profile merges can impact performance, especially in high-traffic sites:

1. Monitor Elasticsearch load during merge operations
2. Consider batch processing for bulk merges
3. Adjust the maxProfilesInOneMerge parameter based on your system capabilities
4. Schedule large merge operations during off-peak hours

Testing Recommendations

To properly test profile merges, consider:

1. **Test Environment**: Set up a separate testing environment that closely mirrors production
2. **Test Scenarios**:
    1. Simple merge of two profiles
    2. Merge of profiles with conflicting property values
    3. Merge with session data and event history
    4. Race condition testing (simultaneous merges)
3. **Data Verification**: Verify that all important data is preserved after merges
4. **Performance Testing**: Test merge operations under load to identify bottlenecks

Implementation Best Practices

1. **Security First**: Always implement merges in authenticated contexts only
2. **Clear Audit Trail**: Maintain logs of merge operations for troubleshooting
3. **Graceful Degradation**: Handle merge failures gracefully without disrupting the user experience
4. **Monitoring**: Set up alerts for unusual merge patterns that might indicate security issues
5. **Data Protection**: Consider regulations like GDPR when merging personal data
6. **Backup Strategy**: Ensure you have a strategy to recover from problematic merges

4.2. Request examples

4.2.1. Prerequisites

Before running any of the examples below, you need to:

1. Create a tenant

First, create a tenant that will own all the data:

```
curl -X POST http://localhost:8181/cxs/tenants \
--user "karaf:$UNOMI_ROOT_PASSWORD" \
-H "Content-Type: application/json" \
-d '{
   "requestedId": "mytenant",
   "properties": {
      "name": "My Company",
      "description": "My tenant description"
   }
}'
```

The response includes the created tenant with **masked** API key metadata only (plaintext secrets are not returned):

```
{
  "itemId": "mytenant",
  "itemType": "tenant",
  "status": "ACTIVE",
  "apiKeys": [
    {
      "maskedKey": "unomi_v1_****ab12",
      "keyType": "PUBLIC",
      "revoked": false
    },
    {
      "maskedKey": "unomi_v1_****cd34",
      "keyType": "PRIVATE",
      "revoked": false
    }
  ],
  "properties": {
    "name": "My Company",
    "description": "My tenant description"
  }
}
```

Regenerate keys to obtain one-time plaintext values (store them immediately):

```
curl -X POST "http://localhost:8181/cxs/tenants/mytenant/apikeys?type=PUBLIC" --user
"karaf:$UNOMI_ROOT_PASSWORD"
curl -X POST "http://localhost:8181/cxs/tenants/mytenant/apikeys?type=PRIVATE" --user
"karaf:$UNOMI_ROOT_PASSWORD"
```

After creating the tenant and regenerating keys, use these credentials in the examples: - Tenant ID: mytenant - Private API Key: plainTextKey from the PRIVATE key-creation response - Public API Key: plainTextKey from the PUBLIC key-creation response

2. Create a scope

Then, create a scope for your digital property (website, mobile app, etc.):

```
curl -X POST http://localhost:8181/cxs/scopes \
--user "mytenant:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "metadata": {
    "id": "mydigital",
    "name": "My Digital Property",
    "description": "My website scope"
  }
}'
```



After creating the scope, continue using the tenant **public** API key (from POST /cxs/tenants/{id}/apikeys?type=PUBLIC) for /cxs/context.json and /cxs/eventcollector. Scope creation does not return an API key.



Tenant Resolution (Version 4.0+)

Starting with Apache Unomi 4.0, tenant resolution is mandatory for all requests to /cxs/context.json and /cxs/eventcollector endpoints. The tenant must be identified before Apache Unomi can process the request. This is done automatically when you use:

- **Public API Key** (recommended for public endpoints): The X-Unomi-Api-Key header with a public API key automatically resolves the tenant
- **Private API Key**: Basic authentication with tenantId:privateApiKey resolves the tenant from the credentials
- **Tenant ID Header**: When using JAAS authentication, the X-Unomi-Tenant-Id header can be used

All examples in this document use the X-Unomi-Api-Key header with a public API key, which is the recommended approach for public-facing applications. See How profile tracking works for complete details about tenant resolution.



In all the examples below, replace: - YOUR_TENANT_ID with mytenant - YOUR_PRIVATE_API_KEY with your actual private API key - example scope with mydigital - YOUR_PUBLIC_API_KEY with the tenant public API key from key regeneration

3. Verify the scope

You can verify the scope was created correctly by retrieving it:

```
curl -X GET http://localhost:8181/cxs/scopes/mydigital \
--user "mytenant:YOUR_PRIVATE_API_KEY"
```

Or list all scopes:

```
curl -X GET http://localhost:8181/cxs/scopes \
--user "mytenant:YOUR_PRIVATE_API_KEY"
```



To get nicely formatted JSON responses, you can pipe the curl output through jq:

```
# Install jq if you don't have it:
# macOS: brew install jq
# Ubuntu/Debian: apt-get install jq
# CentOS/RHEL: yum install jq

# Then add | jq '.' to any curl command:
curl -X GET http://localhost:8181/cxs/scopes \
--user "mytenant:YOUR_PRIVATE_API_KEY" | jq '.'
```

For multi-line curl requests using heredoc syntax (<<'EOF'), you can still use jq:

```
# With heredoc
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d @- <<'EOF' | jq '.'
{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   },
   "requiredProfileProperties": ["*"]
}
EOF

# With inline JSON
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
   "offset": 0,
   "limit": 20
}' | jq '.'
```

Useful jq options and filters:

```
# Pretty print with specific indentation
curl ... | jq '.' --indent 4

# Only show specific fields
curl ... | jq '.metadata'

# Show array elements on separate lines
curl ... | jq '.[]'

# Filter and format specific data
curl ... | jq '.profiles[] | {id: .itemId, name: .properties.firstName}'

# Sort array elements
curl ... | jq '.profiles | sort_by(.properties.lastName)'

# Count array elements
curl ... | jq '.profiles | length'
```

4.2.2. Retrieving your first context

You can retrieve a context using curl like this :

```
curl http://localhost:8181/cxs/context.js?sessionId=1234 \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY"
```

This will retrieve a JavaScript script that contains a cxs object that contains the context with the current user profile, segments, scores as well as functions that makes it easier to perform further requests (such as collecting events using the cxs.collectEvents() function).



When you make a request to the context.js or context.json endpoints, Apache Unomi automatically:

- Creates or loads a visitor profile (from cookie, parameter, or creates a new one)
- Creates or loads a visitor session (if sessionId is provided)
- Processes any events provided in the request (which automatically triggers rule execution)
- Resolves personalization (if requested)
- Sets a cookie with the profile ID in the response

For detailed information about how profile tracking, session management, event processing, and rule execution work, see the How profile tracking works section.

4.2.3. Retrieving a context as a JSON object.

If you prefer to retrieve a pure JSON object, you can simply use a request formed like this:

```
curl http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY"
```

The same automatic profile and session creation behavior applies to context.json requests. See How profile tracking works for complete details.

4.2.4. Accessing profile properties in a context

By default, in order to optimize the amount of data sent over the network, Apache Unomi will not send the content of the profile or session properties. If you need this data, you must send a JSON object to configure the resulting output of the context.js(on) servlet.

Here is an example that will retrieve all the session and profile properties, as well as the profile's segments and scores

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d @- <<'EOF'
{
   "source": {
      "itemId":"homepage",
      "itemType":"page",
      "scope":"mydigital"
   },
   "requiredProfileProperties":["*"],
   "requiredSessionProperties":["*"],
   "requireSegments":true,
   "requireScores":true
}
EOF
```

The requiredProfileProperties and requiredSessionProperties are properties that take an array of property names that should be retrieved. In this case we use the wildcard character '*' to say we want to retrieve all the available properties. The structure of the JSON object that you should send is a JSON-serialized version of the ContextRequest Java class.

4.2.5. Sending events using the context servlet

At the same time as you are retrieving the context, you can also directly send events in the ContextRequest object as illustrated in the following example:

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d @- <<'EOF'
{
   "source":{
      "itemId":"homepage",
      "itemType":"page",
      "scope":"mydigital"
   },
   "events":[
      {
         "eventType":"view",
         "scope": "mydigital",
         "source":{
            "itemType": "site",
            "scope":"mydigital",
            "itemId": "mysite"
         },
         "target":{
            "itemType":"page",
            "scope":"mydigital",
            "itemId":"homepage",
            "properties":{
               "pageInfo":{
                  "referringURL":"https://apache.org/"
               }
            }
         }
      }
   ]
}
EOF
```

Upon received events, Apache Unomi will execute all the rules that match the current context, and return an updated context. This way of sending events is usually used upon first loading of a page. If you want to send events after the page has finished loading you could either do a second call and get an updating context, or if you don't need the context and want to send events in a network optimal way you can use the eventcollector servlet (see below).

4.2.6. Sending events using the eventcollector servlet

If you only need to send events without retrieving a context, you should use the eventcollector servlet that is optimized respond quickly and minimize network traffic. Here is an example of using this servlet:

```
curl -X POST http://localhost:8181/cxs/eventcollector \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d @- <<'EOF'
{
   "sessionId" : "1234",
   "events":[
      {
         "eventType":"view",
         "scope": "mydigital",
         "source":{
            "itemType": "site",
            "scope":"mydigital",
            "itemId": "mysite"
         },
         "target":{
            "itemType":"page",
            "scope":"mydigital",
            "itemId":"homepage",
            "properties":{
               "pageInfo":{
                  "referringURL":"https://apache.org/"
               }
            }
         }
      }
   ]
}
EOF
```

Note that the eventcollector executes the rules but does not return a context. If is generally used after a page is loaded to send additional events.

4.2.7. Where to go from here

- You can find more useful Apache Unomi URLs that can be used in the same way as the above examples.
- Read the Twitter sample documentation that contains a detailed example of how to integrate with Apache Unomi.

4.3. Public API Examples

4.3.1. Sending a context request

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d @- <<'EOF'
{
   "source":{
      "itemId":"homepage",
      "itemType":"page",
      "scope":"mydigital"
   },
   "events":[
      {
         "eventType":"view",
         "scope": "mydigital",
         "source":{
            "itemType": "site",
            "scope":"mydigital",
            "itemId": "mysite"
         },
         "target":{
            "itemType":"page",
            "scope":"mydigital",
            "itemId":"homepage",
            "properties":{
               "pageInfo":{
                  "referringURL":"https://apache.org/"
               }
            }
         }
      }
   ]
}
EOF
```

4.3.2. Collecting events

```
curl -X POST http://localhost:8181/cxs/eventcollector \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d '{
   "sessionId" : "1234",
   "events":[
      {
         "eventType":"contactInfoSubmitted",
         "scope": "mydigital",
         "source":{
            "itemType": "site",
            "scope": "mydigital",
            "itemId": "mysite"
         },
         "target":{
            "itemType": "form",
            "scope": "mydigital",
            "itemId": "contactForm"
         },
         "properties" : {
          "firstName": "John",
          "lastName": "Doe",
          "email": "john.doe@acme.com"
         }
      }
   ]
}'
```

4.4. Private API Examples

4.4.1. Setting up birthday personalization

This example shows how to set up and test birthday-based personalization in Unomi.

1. Creating test profiles

First, let's create two test profiles - one with today's birth date and another with a different date:

```
# Create a profile with today's birth date
curl -X POST http://localhost:8181/cxs/profiles \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
   "itemId": "profile-1",
   "itemType": "profile",
   "scope": "mydigital",
   "properties": {
      "firstName": "John",
      "lastName": "Birthday",
      "email": "john.birthday@example.com",
      "birthDate": "2000-03-24",
      "birthday": "03-24"
   }
}'

# Create a profile with a different birth date
curl -X POST http://localhost:8181/cxs/profiles \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
   "itemId": "profile-2",
   "itemType": "profile",
   "scope": "mydigital",
   "properties": {
      "firstName": "Jane",
      "lastName": "Regular",
      "email": "jane.regular@example.com",
      "birthDate": "1995-12-31",
      "birthday": "12-31"
   }
}'
```



The birthday property stores just the month and day in MM-DD format for easy matching, while birthDate stores the full date.

2. Verifying the profiles

You can verify that both profiles were created with their birth dates:

```
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "offset": 0,
  "limit": 20,
  "condition": {
    "type": "profilePropertyCondition",
    "parameterValues": {
      "propertyName": "properties.birthday",
      "comparisonOperator": "exists"
    }
  }
}'
```

3. Finding profiles with birthdays today

To find all profiles whose birthday matches today's date:

```
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "offset": 0,
  "limit": 20,
  "condition": {
    "type": "profilePropertyCondition",
    "parameterValues": {
      "propertyName": "properties.birthday",
      "comparisonOperator": "equals",
      "propertyValue": "03-24"
    }
  }
}'
```



Replace 03-24 with the current month and day in zero-padded format. For example: - March 24th: 03-24 - December 31st: 12-31

The format is always MM-DD where: - MM is the two-digit month (01-12) - DD is the two-digit day (01-31)

You can also update the personalization example to use the birthday property:



The profile is selected with the context-profile-id cookie, not with a profileId in the request body. /cxs/context.json is a public endpoint, and since Unomi 4.0 a public caller's body profileId is ignored — the cookie is the only profile identity bearer. See Client-facing hardening in the 3.0 to 4.0 migration guide.

```
curl -X POST http://localhost:8181/cxs/context.json \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-b "context-profile-id=profile-1" \
-d '{
   "sessionId": "birthday-session",
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   },
   "requiredProfileProperties": ["properties.birthday"],
   "personalizations": [
      {
         "id": "birthdayMessage",
         "strategy": "matching-first",
         "strategyOptions": {
            "fallback": "Welcome to our site!"
         },
         "contents": [
            {
               "id": "birthday-content",
               "path": "/birthday",
               "content": "🎂 Happy Birthday! Enjoy your special day!",
               "filters": [
                  {
                     "condition": {
                        "type": "profilePropertyCondition",
                        "parameterValues": {
                           "propertyName": "properties.birthday",
                           "comparisonOperator": "equals",
                           "propertyValue": "03-24"
                        }
                     }
                  }
               ]
            }
         ]
      }
   ]
}'
```

And similarly for the birthday segment:

```
curl -X POST http://localhost:8181/cxs/segments \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
 "metadata": {
  "id": "birthdaySegment",
  "name": "Users with Birthday Today",
  "scope": "mydigital"
 },
 "condition": {
  "type": "profilePropertyCondition",
  "parameterValues": {
   "propertyName": "properties.birthday",
   "comparisonOperator": "equals",
   "propertyValue": "03-24"
  }
 }
}'
```



To make this more maintainable in a production environment, you can deploy a custom condition type that handles birthday matching:

```
curl -X POST http://localhost:8181/cxs/definitions \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
 "metadata": {
  "id": "birthdayTodayCondition",
  "name": "birthdayTodayCondition",
  "description": "A condition that matches birthdays on current day",
  "systemTags": [
   "profileCondition",
   "demographic",
   "condition"
  ]
 },
 "parentCondition": {
  "type": "profilePropertyCondition",
  "parameterValues": {
   "propertyName": "properties.birthday",
   "comparisonOperator": "equals",
   "propertyValue": "parameter::monthDay"
  }
 },
 "parameters": [
  {
   "id": "monthDay",
   "type": "string",
   "multivalued": false,
   "defaultValue": "03-24"
  }
 ]
}'
```

After deploying the condition, you can use it in your searches and segments like this:

```
{
  "type": "birthdayTodayCondition",
  "parameterValues": {
    "monthDay": "03-24"
  }
}
```

For example, to create a segment using this condition:

```
curl -X POST http://localhost:8181/cxs/segments \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "metadata": {
    "id": "birthdaySegment",
    "name": "Users with Birthday Today",
    "scope": "mydigital"
  },
  "condition": {
    "type": "birthdayTodayCondition",
    "parameterValues": {
      "monthDay": "03-24"
    }
  }
}'
```

4. Testing personalization

Now we can test how personalization works for both profiles. We'll use the context.json endpoint to get personalized content:

For the birthday profile (should show birthday message):

```
curl -X POST http://localhost:8181/cxs/context.json \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-b "context-profile-id=profile-1" \
-d '{
   "sessionId": "birthday-session",
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   },
   "requiredProfileProperties": ["properties.birthday"],
   "personalizations": [
      {
         "id": "birthdayMessage",
         "strategy": "matching-first",
         "strategyOptions": {
            "fallback": "Welcome to our site!"
         },
         "contents": [
            {
               "id": "birthday-content",
               "path": "/birthday",
               "content": " Happy Birthday! Enjoy your special day!",
               "filters": [
                  {
                     "condition": {
                        "type": "profilePropertyCondition",
                        "parameterValues": {
                           "propertyName": "properties.birthday",
                           "comparisonOperator": "equals",
                           "propertyValue": "03-24"
                        }
                     }
                  }
               ]
            }
         ]
      }
   ]
}'
```

For the non-birthday profile (should show welcome message):

```
curl -X POST http://localhost:8181/cxs/context.json \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-b "context-profile-id=profile-2" \
-d '{
   "sessionId": "regular-session",
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   },
   "requiredProfileProperties": ["properties.birthday"],
   "personalizations": [
      {
         "id": "birthdayMessage",
         "strategy": "matching-first",
         "strategyOptions": {
            "fallback": "Welcome to our site!"
         },
         "contents": [
            {
               "id": "birthday-content",
               "path": "/birthday",
               "content": "🎂 Happy Birthday! Enjoy your special day!",
               "filters": [
                  {
                     "condition": {
                        "type": "profilePropertyCondition",
                        "parameterValues": {
                           "propertyName": "properties.birthday",
                           "comparisonOperator": "equals",
                           "propertyValue": "03-24"
                        }
                     }
                  }
               ]
            }
         ]
      }
   ]
}'
```

The responses will include a personalizations object that contains: - For profile-1: The birthday message "🎂 Happy Birthday! Enjoy your special day!" - For profile-2: The fallback message "Welcome to our site!"

5. Creating a birthday segment

You can also create a segment to automatically group profiles with birthdays today:

```
curl -X POST http://localhost:8181/cxs/segments \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "metadata": {
    "id": "birthdaySegment",
    "name": "Users with Birthday Today",
    "scope": "mydigital"
  },
  "condition": {
    "type": "profilePropertyCondition",
    "parameterValues": {
      "propertyName": "properties.birthday",
      "comparisonOperator": "equals",
      "propertyValue": "03-24"
    }
  }
}'
```

4.4.2. Searching profiles

```
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "offset" : 0,
  "limit" : 20,
  "condition" : {
    "type": "profilePropertyCondition",
    "parameterValues" : {
      "propertyName" : "properties.firstName",
      "comparisonOperator" : "equals",
      "propertyValue" : "John"
    }
  }
}'
```

4.4.3. Creating a segment

```
curl -X POST http://localhost:8181/cxs/segments \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "metadata": {
    "id": "newSegment",
    "name": "New Segment",
    "scope": "mydigital"
  },
  "condition": {
    "type": "profilePropertyCondition",
    "parameterValues": {
      "propertyName": "properties.age",
      "comparisonOperator": "greaterThan",
      "propertyValueInteger": 25
    }
  }
}'
```

4.4.4. Setting up product view tracking

Before using the product view search examples, you need to send product view events to Unomi. Here's how to set it up:

1. Sending a product view event

You can use the eventcollector endpoint to send product view events:

```
curl -X POST http://localhost:8181/cxs/eventcollector \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d '{
   "sessionId": "1234",
   "events": [
      {
         "eventType": "view",
         "scope": "mydigital",
         "source": {
            "itemType": "site",
            "scope": "mydigital",
            "itemId": "mysite"
         },
         "target": {
            "itemType": "product",
            "scope": "mydigital",
            "itemId": "product-123",
            "properties": {
               "pageInfo": {
                  "referringURL": "https://www.google.com"
               }
            }
         }
      }
   ]
}'
```

Key points about the event structure: 1. Use "eventType": "view" for view events 2. Set target.itemType to "product" for product views 3. Include product details in target.properties 4. Use consistent itemId values to track the same product

2. Using the context.json endpoint

For web applications, you can also send product views through the context.json endpoint:

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d '{
   "source": {
      "itemId": "product-page",
      "itemType": "page",
      "scope": "mydigital"
   },
   "events": [
      {
         "eventType": "view",
         "scope": "mydigital",
         "source": {
            "itemType": "site",
            "scope": "mydigital",
            "itemId": "mysite"
         },
         "target": {
            "itemType": "product",
            "scope": "mydigital",
            "itemId": "product-123",
            "properties": {
               "pageInfo": {
                  "referringURL": "https://www.google.com"
               }
            }
         }
      }
   ],
   "requiredProfileProperties": ["*"]
}'
```

3. Product properties schema



To simplify product view tracking, you can deploy a custom condition type that combines all the necessary event conditions:

```
curl -X POST http://localhost:8181/cxs/definitions \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "metadata": {
    "id": "productViewEventCondition",
    "name": "productViewEventCondition",
    "description": "A condition that matches product view events",
    "systemTags": [
      "eventCondition",
      "event",
      "condition"
    ]
  },
  "parentCondition": {
    "type": "booleanCondition",
    "parameterValues": {
      "operator": "and",
      "subConditions": [
        {
          "type": "eventTypeCondition",
          "parameterValues": {
            "eventTypeId": "view"
          }
        },
        {
          "type": "eventPropertyCondition",
          "parameterValues": {
            "propertyName": "target.itemType",
            "comparisonOperator": "equals",
            "propertyValue": "product"
          }
        },
        {
          "type": "eventPropertyCondition",
          "parameterValues": {
            "propertyName": "target.itemId",
            "comparisonOperator": "equals",
            "propertyValue": "parameter::productId"
          }
        }
      ]
    }
  },
  "parameters": [
    {
      "id": "productId",
      "type": "string",
      "multivalued": false
    }
  ]
}'
```

After deploying the condition, you can use it in your searches like this:

```
{
 "type": "productViewEventCondition",
 "parameterValues": {
  "productId": "product-123"
 }
}
```

This custom condition type: 1. Is properly tagged with eventCondition making it valid for use in pastEventCondition 2. Combines all the necessary conditions using booleanCondition in its definition 3. Provides a simple parameter interface (just specify the product ID)

Unomi is flexible with product properties - you don't need to declare a schema beforehand. However, for consistency, you should: 1. Use consistent property names across events 2. Use consistent value types (e.g., always use numbers for prices) 3. Use consistent categories and other enumerated values

Common product properties to consider: - name: Product name (string) - category: Product category (string) - price: Product price (number) - brand: Product brand (string) - sku: Stock keeping unit (string) - color: Product color (string) - size: Product size (string) - inStock: Stock status (boolean)

4. Testing the setup

To verify your events are being recorded, you can:

1. Send multiple view events for the same product
2. Wait a few seconds for processing
3. Use the profile search example below to check if the views were counted

```
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
 "offset": 0,
 "limit": 20,
 "condition": {
  "type": "pastEventCondition",
  "parameterValues": {
   "numberOfDays": 1,
   "minimumEventCount": 1,
   "eventCondition": {
    "type": "productViewEventCondition",
    "parameterValues": {
     "productId": "product-123"
    }
   }
  }
 }
}'
```

4.4.5. Searching profiles with frequent product views

This example shows how to find profiles that have viewed a specific product at least 3 times in the last 7 days:

```
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "offset": 0,
  "limit": 20,
  "condition": {
    "type": "pastEventCondition",
    "parameterValues": {
      "numberOfDays": 7,
      "minimumEventCount": 3,
      "eventCondition": {
        "type": "productViewEventCondition",
        "parameterValues": {
          "productId": "product-123"
        }
      }
    }
  }
}'
```

This search will: 1. Look for product view events in the past 7 days 2. Match events that: - Have type "view" - Target a product (target.itemType = "product") - Target the specific product ID (product-123) 3. Return profiles that have at least 3 such events 4. Results are paginated (20 results per page)

You can adjust: - minimumEventCount: change the minimum number of views required - maximumEventCount: optionally set a maximum number of views - numberOfDays: modify the time period to look back - operator: use "eventsOccurred" (default) or "eventsNotOccurred" - productId: change which product to track

For example, to find profiles that have viewed products in a specific category, you could create another custom condition type productCategoryViewEventCondition.json:

```
{
 "metadata": {
  "id": "productCategoryViewEventCondition",
  "name": "productCategoryViewEventCondition",
  "description": "A condition that matches product views in a category",
  "systemTags": [
   "eventCondition",
   "event",
   "condition"
  ]
 },
 "parentCondition": {
  "type": "booleanCondition",
  "parameterValues": {
   "operator": "and",
   "subConditions": [
    {
     "type": "eventTypeCondition",
     "parameterValues": {
      "eventTypeId": "view"
     }
    },
    {
     "type": "eventPropertyCondition",
     "parameterValues": {
      "propertyName": "target.itemType",
      "comparisonOperator": "equals",
      "propertyValue": "product"
     }
    },
    {
     "type": "eventPropertyCondition",
     "parameterValues": {
      "propertyName": "target.properties.category",
      "comparisonOperator": "equals",
      "propertyValue": "parameter::category"
     }
    }
   ]
  }
 },
 "parameters": [
  {
   "id": "category",
   "type": "string",
   "multivalued": false
  }
 ]
}
```

Then use it like this:

```
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "offset": 0,
  "limit": 20,
  "condition": {
    "type": "pastEventCondition",
    "parameterValues": {
      "numberOfDays": 30,
      "minimumEventCount": 5,
      "eventCondition": {
        "type": "productCategoryViewEventCondition",
        "parameterValues": {
          "category": "electronics"
        }
      }
    }
  }
}'
```

4.5. Setting up Groovy Actions

4.5.1. Deploying a Groovy Action

This example shows how to deploy and use a Groovy action to automatically extract the birthday (MM-DD) from a full birthDate.

1. Create the Groovy action file

First, create a file named ExtractBirthdayAction.groovy with this content:

```
package org.apache.unomi.groovy.actions

import org.apache.unomi.api.Event
import org.apache.unomi.api.Profile
import org.apache.unomi.api.actions.Action
import org.apache.unomi.api.actions.ActionExecutor
import org.apache.unomi.api.services.EventService

class ExtractBirthdayAction implements ActionExecutor {
    public int execute(Action action, Event event) {
        Profile profile = event.getProfile()
        def birthDate = profile.getProperty("birthDate")

        if (birthDate != null && birthDate instanceof String && birthDate.length() >= 10) {
            try {
                // Extract month-day part (e.g., "03-24" from "2000-03-24")
                def monthDay = birthDate.substring(5, 10)

                // Only update if different to avoid unnecessary saves
                if (monthDay != profile.getProperty("birthday")) {
                    profile.setProperty("birthday", monthDay)
                    return EventService.PROFILE_UPDATED
                }
            } catch (Exception e) {
                // Log error or handle invalid date format
            }
        }
        return EventService.NO_CHANGE
    }
}
```

2. Deploy the Groovy action

Use the Groovy actions endpoint to deploy the action:

```
curl -X POST http://localhost:8181/cxs/groovyActions \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: multipart/form-data" \
-F "file=@ExtractBirthdayAction.groovy"
```



The action ID will be extractBirthday (derived from the filename without the .groovy extension).

3. Create the action definition

After deploying the Groovy script, create the action definition:

```
curl -X POST http://localhost:8181/cxs/definitions \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "metadata": {
    "id": "extractBirthdayAction",
    "name": "Extract Birthday Action",
    "description": "Extracts MM-DD from birthDate and sets it as birthday property",
    "systemTags": [
      "profileTags",
      "demographic",
      "event"
    ]
  },
  "actionExecutor": "groovy:extractBirthday",
  "parameters": []
}'
```

4. Create a rule to trigger the action

Create a rule that will trigger the Groovy action whenever a profile's birthDate is set or modified:

```
curl -X POST http://localhost:8181/cxs/rules \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "metadata": {
    "id": "setBirthdayRule",
    "name": "Set Birthday Rule",
    "description": "Sets birthday property when birthDate changes",
    "scope": "mydigital"
  },
  "condition": {
    "type": "booleanCondition",
    "parameterValues": {
      "operator": "and",
      "subConditions": [
        {
          "type": "profilePropertyCondition",
          "parameterValues": {
            "propertyName": "properties.birthDate",
            "comparisonOperator": "exists"
          }
        },
        {
          "type": "profileUpdatedEventCondition",
          "parameterValues": {
            "propertyName": "properties.birthDate"
          }
        }
      ]
    }
  },
  "actions": [
    {
      "type": "extractBirthdayAction",
      "parameterValues": {}
    }
  ]
}'
```

5. Test with example profiles

Create test profiles to verify the action works:

```
# Create a profile with birthDate - should trigger the action
curl -X POST http://localhost:8181/cxs/profiles \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
   "itemId": "test-profile-1",
   "properties": {
      "firstName": "John",
      "lastName": "Doe",
      "birthDate": "1990-03-24"
   }
}'

# Verify the birthday property was set
curl -X GET http://localhost:8181/cxs/profiles/test-profile-1 \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" | jq '.'

# Update an existing profile's birthDate - should trigger the action
curl -X PATCH http://localhost:8181/cxs/profiles/test-profile-1 \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json" \
-d '{
   "properties": {
      "birthDate": "1990-12-31"
   }
}'

# Verify the birthday property was updated
curl -X GET http://localhost:8181/cxs/profiles/test-profile-1 \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" | jq '.'
```

The responses should show: 1. First profile creation: birthday property set to "03-24" 2. After update: birthday property changed to "12-31"

6. Remove the Groovy action (if needed)

To remove the Groovy action:

```
curl -X DELETE http://localhost:8181/cxs/groovyActions/extractBirthday \
--user "YOUR_TENANT_ID:YOUR_PRIVATE_API_KEY" \
-H "Content-Type: application/json"
```



This will only remove the Groovy script. You'll need to separately delete the action definition and rule if desired.



Best practices for Groovy actions: 1. Always handle potential errors gracefully 2. Check property existence and types 3. Avoid unnecessary profile updates 4. Use meaningful action and rule names 5. Test with various date formats

4.5.2. Using the explain parameter for request tracing

Apache Unomi provides a powerful request tracing feature through the explain query parameter. This feature helps administrators understand how requests are processed internally, including event processing, condition evaluations, and rule executions.

For save-time condition checks (before runtime), see Condition validation. For a conditions-focused debugging walkthrough, see Debugging conditions.

Prerequisites

To use the explain parameter, you must have one of the following roles: - ADMINISTRATOR - TENANT_ADMINISTRATOR

Request examples

Context request with explain

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234&explain=true \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
   "source": {
      "itemId":"homepage",
      "itemType":"page",
      "scope":"mydigital"
   },
   "requiredProfileProperties":["*"],
   "requireSegments":true
}
EOF
```

Event collector request with explain

```
curl -X POST http://localhost:8181/cxs/eventcollector?explain=true \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
   "sessionId": "1234",
   "events": [
      {
         "eventType":"view",
         "scope": "mydigital",
         "source":{
            "itemType": "site",
            "scope":"mydigital",
            "itemId": "mysite"
         },
         "target":{
            "itemType":"page",
            "scope":"mydigital",
            "itemId":"homepage"
         }
      }
   ]
}
EOF
```

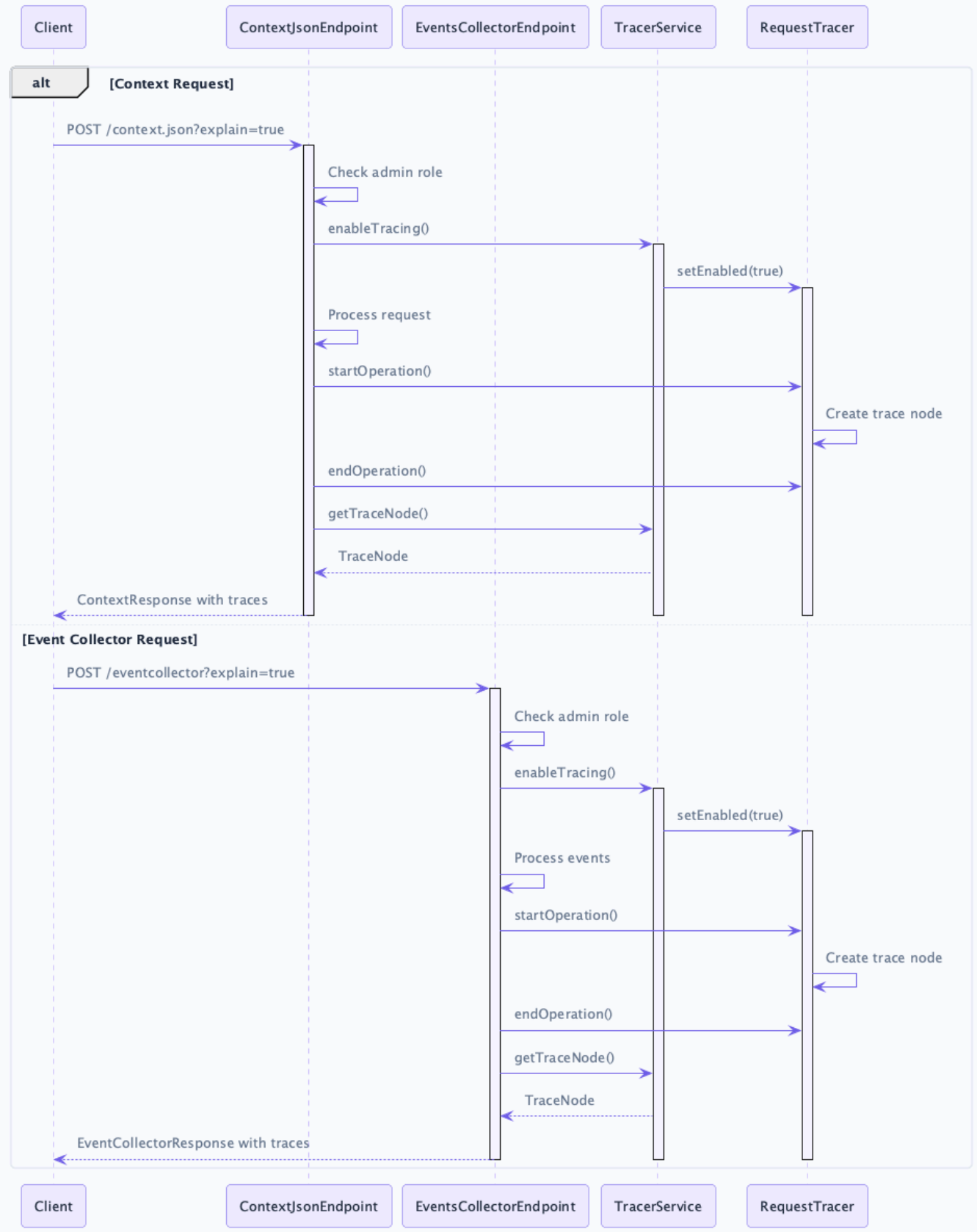
Understanding the trace output

The explain parameter adds a requestTracing field to the response that contains a tree structure of all operations performed during request processing. Here's an example trace output:

```
{
  "profileId": "12345",
  "sessionId": "1234",
  // ... other response fields ...
  "requestTracing": {
    "operationType": "request-processing",
    "description": "Processing context request",
    "startTime": 1234567890,
    "endTime": 1234567899,
    "children": [
      {
        "operationType": "event-validation",
        "description": "Validating event: view",
        "startTime": 1234567891,
        "endTime": 1234567892,
        "traces": [
          "Validating against schema event/1-0-0",
          "Event validation successful"
        ]
      },
      {
        "operationType": "rule-evaluation",
        "description": "Evaluating rules for event",
        "startTime": 1234567893,
        "endTime": 1234567895,
        "children": [
          {
            "operationType": "condition-evaluation",
            "description": "Evaluating condition: matchAll",
            "startTime": 1234567894,
            "endTime": 1234567894,
            "result": true
          }
        ]
      }
    ]
  }
}
```


Request processing flow

The following diagram shows the high-level flow of request processing when explain is enabled:



Common trace operations

The tracing system captures various types of operations:

1. Request Processing
    - Overall request handling

- Parameter validation
- Schema validation

2. Event Processing
   - Event validation
   - Event type resolution
   - Event property processing
3. Rule Evaluation
   - Condition evaluation
   - Action execution
   - Score updates
4. Profile Operations
   - Profile merging
   - Property updates
   - Segment evaluation

Each operation in the trace contains: - Operation type - Description - Start time - End time - Result (if applicable) - Child operations - Trace messages

Best practices

1. Use explain parameter selectively
   - Only enable when debugging or troubleshooting
   - Disable in production environments
   - Consider performance impact
2. Analyze trace output
   - Look for unexpected operations
   - Check operation timing
   - Review validation results
   - Monitor rule evaluations
3. Security considerations
   - Only grant admin access to trusted users
   - Monitor explain parameter usage
   - Review trace data for sensitive information

Complex personalization example with explain

This example demonstrates using the explain parameter to understand how personalization filters are evaluated:

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234&explain=true \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "source": {
    "itemId": "homepage",
    "itemType": "page",
    "scope": "mydigital"
  },
  "requiredProfileProperties": ["*"],
  "requireSegments": true,
  "personalizations": [
    {
      "id": "homepage-hero",
      "strategy": "matching-first",
      "strategyOptions": {
        "fallback": "default-content"
      },
      "contents": [
        {
          "id": "premium-user-content",
          "filters": [
            {
              "condition": {
                "type": "profileSegmentCondition",
                "parameterValues": {
                  "segments": ["premium-users"]
                }
              }
            },
            {
              "condition": {
                "type": "pastEventCondition",
                "parameterValues": {
                  "eventType": "purchase",
                  "minimumEventCount": 1,
                  "numberOfDays": 30
                }
              }
            }
          ]
        },
        {
          "id": "new-visitor-content",
          "filters": [
            {
              "condition": {
                "type": "sessionPropertyCondition",
                "parameterValues": {
                  "propertyName": "duration",
```

```
                    "comparisonOperator": "lessThan",
                    "propertyValueInteger": 300
                }
            }
        }
    ]
  }
 ]
}
EOF
```

The response will include detailed tracing of the personalization evaluation process:

```
{
  "profileId": "12345",
  "sessionId": "1234",
  "requestTracing": {
    "operationType": "request-processing",
    "description": "Processing context request with personalization",
    "startTime": 1234567890,
    "endTime": 1234567899,
    "children": [
      {
        "operationType": "personalization-evaluation",
        "description": "Evaluating personalization: homepage-hero",
        "startTime": 1234567891,
        "endTime": 1234567895,
        "children": [
          {
            "operationType": "content-filter-evaluation",
            "description": "Evaluating filters for content: premium-user-content",
            "startTime": 1234567892,
            "endTime": 1234567893,
            "children": [
              {
                "operationType": "condition-evaluation",
                "description": "Evaluating segment condition",
                "startTime": 1234567892,
                "endTime": 1234567892,
                "result": false,
                "traces": [
                  "Profile not in segment: premium-users"
                ]
              },
              {
                "operationType": "condition-evaluation",
                "description": "Evaluating past event condition",
                "startTime": 1234567892,
                "endTime": 1234567893,
                "result": false,
                "traces": [
                  "No purchase events found in last 30 days"
                ]
```

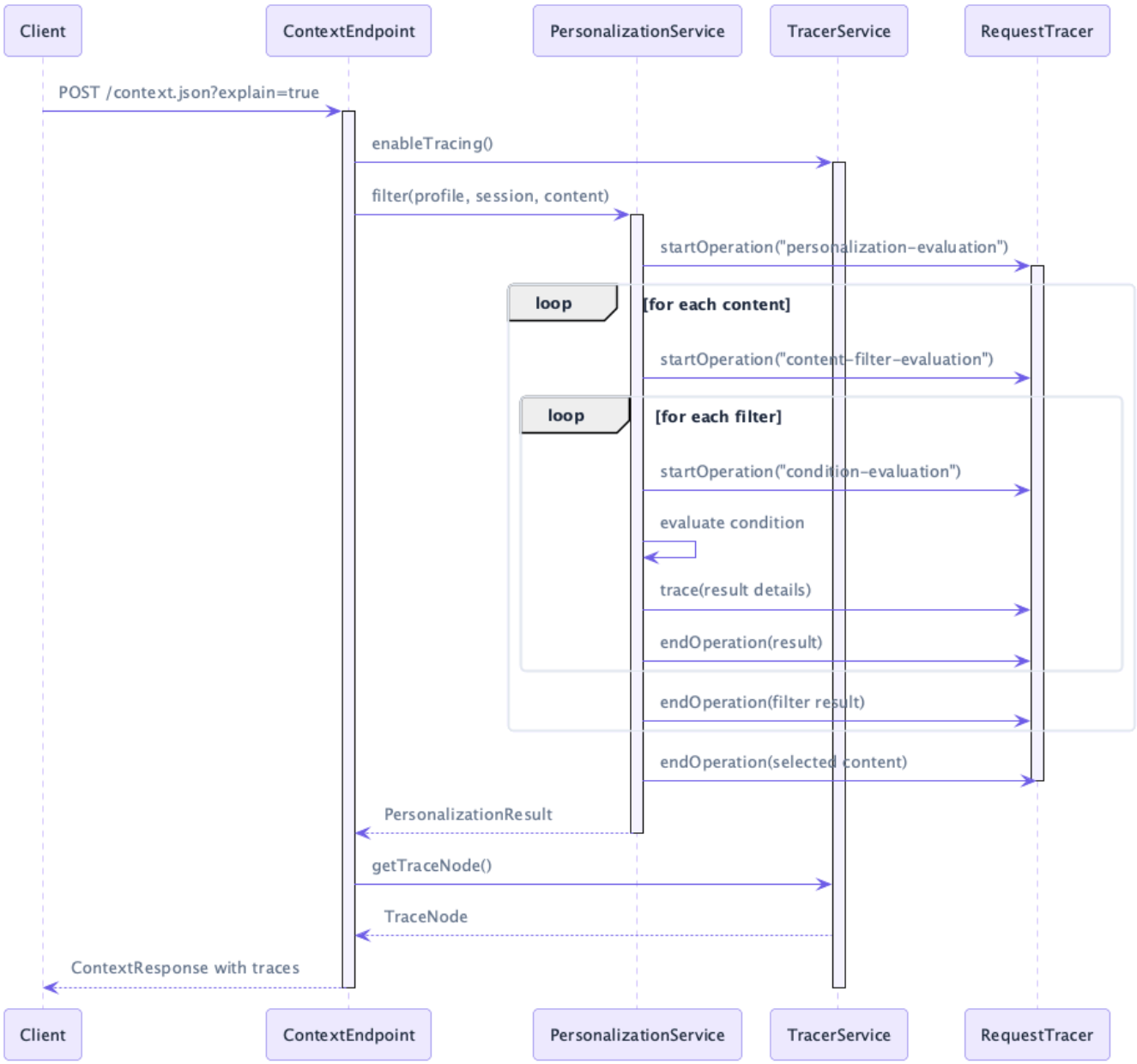
```
                }
              ],
              "result": false
            },
            {
              "operationType": "content-filter-evaluation",
              "description": "Evaluating filters for content: new-visitor-content",
              "startTime": 1234567893,
              "endTime": 1234567894,
              "children": [
                {
                  "operationType": "condition-evaluation",
                  "description": "Evaluating session duration condition",
                  "startTime": 1234567893,
                  "endTime": 1234567894,
                  "result": true,
                  "traces": [
                    "Session duration: 120 seconds, threshold: 300 seconds"
                  ]
                }
              ],
              "result": true
            }
          ]
        }
      ]
    },
    "personalizations": {
      "homepage-hero": "new-visitor-content"
    }
}
```


This example demonstrates:

1. Complex personalization setup
    - Multiple content variants
    - Different condition types
    - Fallback content
    - Strategy configuration
2. Detailed tracing of
    - Personalization evaluation flow
    - Filter condition evaluation
    - Segment membership checks
    - Past event queries
    - Session property checks
3. Trace node hierarchy showing

- Parent-child relationships
- Timing information
- Decision points
- Result propagation

The trace output helps understand: - Why specific content was selected - Which conditions failed/passed - Performance of different operations - Order of evaluation



This sequence diagram shows the detailed flow of personalization evaluation with tracing enabled, including: 1. Initial request handling 2. Personalization service interaction 3. Filter evaluation loops 4. Trace node creation and updates 5. Result aggregation and response

Chapter 5. Configuration

5.1. Centralized configuration

Apache Unomi uses a centralized configuration file that contains both system properties and configuration properties. These settings are then fed to the OSGi and other configuration files using placeholder that look something like this:

```
contextserver.publicAddress=${org.apache.unomi.cluster.public.address:-http://localhost:8181}
contextserver.internalAddress=${org.apache.unomi.cluster.internal.address:-https://localhost:9443}
```

Default values are stored in a file called $MY_KARAF_HOME/etc/custom.system.properties but you should never modify this file directly, as an override mechanism is available. Simply create a file called:

```
unomi.custom.system.properties
```

and put your own property values in their to override the defaults OR you can use environment variables to also override the values in the $MY_KARAF_HOME/etc/custom.system.properties. See the next section for more information about that.

5.2. Changing the default configuration using environment variables (i.e. Docker configuration)

You might want to use environment variables to change the default system configuration, especially if you intend to run Apache Unomi inside a Docker container. You can find the list of all the environment variable names in the following file:

https://github.com/apache/unomi/blob/master/package/src/main/resources/etc/custom.system.properties

If you are using Docker Container, simply pass the environment variables on the docker command line or if you are using Docker Compose you can put the environment variables in the docker-compose.yml file.

If you want to "save" the environment values in a file, you can use the bin/setenv(.bat) to setup the environment variables you want to use.

5.3. Changing the default configuration using property files

If you want to change the default configuration using property files instead of environment variables, you can perform any modification you want in the $MY_KARAF_HOME/etc/unomi.custom.system.properties file.

By default this file does not exist and is designed to be a file that will contain only your custom modifications to the default configuration.

For example, if you want to change the HTTP ports that the server is listening on, you will need to create the following lines in the $MY_KARAF_HOME/etc/unomi.custom.system.properties (and create it if you haven't yet) file:

```
org.osgi.service.http.port.secure=9443
org.osgi.service.http.port=8181
```

If you change these ports, also make sure you adjust the following settings in the same file :

```
org.apache.unomi.cluster.public.address=http://localhost:8181
org.apache.unomi.cluster.internal.address=https://localhost:9443
```

If you need to specify a search engine configuration that is different than the default, it is recommended to do this BEFORE you start the server for the first time, or you will lose all the data you have stored previously.

Apache Unomi supports both Elasticsearch and OpenSearch as search engine backends. Here are the configuration properties for each:

For Elasticsearch:

```
org.apache.unomi.elasticsearch.cluster.name=contextElasticsearch
# The elasticsearch.addresses may be a comma separated list of host names and ports such as
# hostA:9200,hostB:9200
# Note: the port number must be repeated for each host.
org.apache.unomi.elasticsearch.addresses=localhost:9200
# Optional Elasticsearch security (when xpack security is enabled)
org.apache.unomi.elasticsearch.username=elastic
org.apache.unomi.elasticsearch.password=${env:ELASTIC_PASSWORD}
org.apache.unomi.elasticsearch.sslEnable=true
org.apache.unomi.elasticsearch.sslTrustAllCertificates=true
```

For OpenSearch:

```
org.apache.unomi.opensearch.cluster.name=opensearch-cluster
# The opensearch.addresses may be a comma separated list of host names and ports such as
# hostA:9200,hostB:9200
# Note: the port number must be repeated for each host.
org.apache.unomi.opensearch.addresses=localhost:9200

# OpenSearch security settings (required by default since OpenSearch 3)
org.apache.unomi.opensearch.sslEnable=true
org.apache.unomi.opensearch.username=admin
org.apache.unomi.opensearch.password=${env:OPENSEARCH_INITIAL_ADMIN_PASSWORD:-
admin}
org.apache.unomi.opensearch.sslTrustAllCertificates=true
```

A query that does not give a size (for example a /cxs/segments/query body without limit) returns

defaultQueryLimit results, 10 by default. A size of -1 returns every result. The setting applies to every search, including the segment and scoring lists that merge system and tenant items:

```
# env UNOMI_ELASTICSEARCH_DEFAULTQUERYLIMIT
org.apache.unomi.elasticsearch.defaultQueryLimit=10
# env UNOMI_OPENSEARCH_DEFAULTQUERYLIMIT
org.apache.unomi.opensearch.defaultQueryLimit=10
```

To select which search engine to use, you can: 1. Use the appropriate configuration properties above 2. When building from source, use the appropriate Maven profile: * For Elasticsearch (default): no special profile needed * For OpenSearch: add -Duse.opensearch=true to your Maven command (this will only impact the integration tests if they are activated using the -Pintegration-tests profile) 3. When using Docker: * For Elasticsearch: use UNOMI_DISTRIBUTION=unomi-distribution-elasticsearch * For OpenSearch: use UNOMI_DISTRIBUTION=unomi-distribution-opensearch * For custom configurations: use UNOMI_DISTRIBUTION=your-custom-distribution-name

Note: The UNOMI_DISTRIBUTION environment variable accepts any karaf feature that is available from Karaf. Two built-in distributions (unomi-distribution-elasticsearch, unomi-distribution-opensearch) are packaged with UNOMI archive but any other can be added, either in the package or by adding a feature repository that contains the desired one in Karaf.

Note: When using OpenSearch 3.x: - Security is enabled by default and requires SSL/TLS - The default admin username is 'admin' - The initial admin password can be set via OPENSEARCH_INITIAL_ADMIN_PASSWORD environment variable

5.4. Refresh Policy Configuration

Apache Unomi provides the ability to configure refresh policies for different item types in both Elasticsearch and OpenSearch. This feature allows you to optimize when changes to documents are made visible to search.

For Elasticsearch:

```
# refresh policy per item type in Json format.
# Valid values are WAIT_UNTIL/IMMEDIATE/NONE. The default refresh policy is NONE.
# Example: {"event":"WAIT_UNTIL","rule":"NONE"}
org.apache.unomi.elasticsearch.itemTypeToRefreshPolicy={"scheduledTask":"WAIT_UNTIL"}
```

For OpenSearch:

```
# refresh policy per item type in Json format.
# Valid values are WaitFor/True/False. The default refresh policy is False.
# Example: {"event":"WaitFor","rule":"False"}
org.apache.unomi.opensearch.itemTypeToRefreshPolicy={"scheduledTask":"WaitFor"}
```

The refresh policy can be configured per item type and affects when changes to documents of that type

become visible:

- **WAIT_UNTIL/WaitFor**: Changes will be visible after the operation completes but will wait for a refresh. This ensures changes are immediately visible while being more efficient than an immediate refresh.
- **IMMEDIATE/True**: Changes will be made visible immediately (forces an index refresh). This is the most resource-intensive option but guarantees immediate visibility.
- **NONE/False**: Changes will be made visible when a refresh happens, either from the configured refresh interval or when manually triggered. This is the most efficient option but doesn't guarantee when changes will be visible.

For distributed task management, it's recommended to use **WAIT_UNTIL/WaitFor** for the scheduledTask item type to ensure lock state changes are immediately visible to all nodes without requiring explicit refresh calls in the code.

Authentication Rules:

1. If JAAS authentication is provided (username/password), it grants full access to all endpoints
2. Public paths (like /context.json) require a valid public API key
3. Private paths require both tenantId and private API key
4. All other requests are denied

5.5. Secured events configuration

Apache Unomi can restrict some event types so only callers from allowed IP addresses may send them. How that works depends on whether single-tenant compatibility mode is on.

5.5.1. Normal multi-tenant mode

In normal 4.0 mode, protected events are controlled with two fields on the tenant object, set through REST /cxs/tenants or crud update tenant (where restrictedEventTypes is named restrictedEventPermissions):

- restrictedEventTypes — extra event type ids that need an authorized source IP (for example login)
- authorizedIPs — IP addresses or CIDR ranges allowed to send protected events

Some event types are always protected, whatever the tenant lists: sessionCreated, sessionReassigned, profileUpdated, ruleFired and updateProperties. Unomi checks the caller IP of a protected event against authorizedIPs. See Multi-tenancy for tenant updates.

The global third-party provider files below are **not** used in this mode.

5.5.2. Single-tenant compatibility mode (third-party / peer servers)

When single-tenant compatibility mode is enabled, Unomi keeps the Unomi 2 / Unomi 3 third-party

provider model. Peer servers send the secret in the X-Unomi-Peer HTTP header. You can adjust the defaults with the centralized configuration file override in $MY_KARAF_HOME/etc/unomi.custom.system.properties.

You can find the default configuration in the following file:

```
$MY_KARAF_HOME/etc/custom.system.properties
```

The properties start with the prefix : org.apache.unomi.thirdparty.* and here are the default values :

```
org.apache.unomi.thirdparty.provider1.key=${env:UNOMI_THIRDPARTY_PROVIDER1_KEY:-
670c26d1cc413346c3b2fd9ce65dab41}
org.apache.unomi.thirdparty.provider1.ipAddresses=${env:UNOMI_THIRDPARTY_PROVIDER1_IPAD
DRESSES:-127.0.0.1,::1}
org.apache.unomi.thirdparty.provider1.allowedEvents=${env:UNOMI_THIRDPARTY_PROVIDER1_A
LLOWEDEVENTS:-login,updateProperties}
```

The events set in allowedEvents will be secured and will only be accepted if the call comes from the specified IP address, and if the secret is passed in the X-Unomi-Peer HTTP request header. The "env:" part means that it will attempt to read an environment variable by that name, and if it's not found it will default to the value after the ":-" marker.

It is now also possible to use IP address ranges instead of having to list all valid IP addresses for event sources. This is very useful when working in cluster deployments where servers may be added or removed dynamically. In order to support this Apache Unomi uses a library called IPAddress that supports IP ranges and subnets. Here is an example of how to setup a range:

```
org.apache.unomi.thirdparty.provider1.ipAddresses=${env:UNOMI_THIRDPARTY_PROVIDER1_IPAD
DRESSES:-192.168.1.1-100,::1}
```

The above configuration will allow a range of IP addresses between 192.168.1.1 and 192.168.1.100 as well as the IPv6 loopback.

Here's another example using the subnet format:

```
org.apache.unomi.thirdparty.provider1.ipAddresses=${env:UNOMI_THIRDPARTY_PROVIDER1_IPAD
DRESSES:-1.2.0.0/16,::1}
```

The above configuration will allow all addresses starting with 1.2 as well as the IPv6 loopback address.

Wildcards may also be used:

```
org.apache.unomi.thirdparty.provider1.ipAddresses=${env:UNOMI_THIRDPARTY_PROVIDER1_IPAD
DRESSES:-1.2.*.*,::1}
```

The above configuration is exactly the same as the previous one.

More advanced ranges and subnets can be used as well, please refer to the IPAddress library documentation for details on how to format them.

If you want to add another provider you will need to add them manually in the following file (and make sure you maintain the changes when upgrading) :

```
$MY_KARAF_HOME/etc/org.apache.unomi.thirdparty.cfg
```

These provider settings apply only while single-tenant compatibility mode is enabled. Usually, login events, which operate on profiles and do merge on protected properties, must be secured. For each trusted third party server, you need to add these 3 lines :

```
thirdparty.provider1.key=replace-with-a-long-unique-secret
thirdparty.provider1.ipAddresses=127.0.0.1,::1
thirdparty.provider1.allowedEvents=login,updateProperties
```

Full peer abilities and troubleshooting for this mode are in single-tenant compatibility mode.

5.6. Installing the MaxMind GeoIPLite2 IP lookup database

Apache Unomi requires an IP database in order to resolve IP addresses to user location. The GeoLite2 database can be downloaded from MaxMind here : http://dev.maxmind.com/geoip/geoip2/geolite2/

Simply download the GeoLite2-City.mmdb file into the "etc" directory.

X-Forwarded-For and the remoteAddr query parameter are used only when the direct peer is in org.apache.unomi.ip.trustedProxies (loopback and private ranges by default).

5.7. Installing Geonames database

Apache Unomi includes a geocoding service based on the geonames database (http://www.geonames.org/). It can be used to create conditions on countries or cities.

In order to use it, you need to install the Geonames database into . Get the "allCountries.zip" database from here : http://download.geonames.org/export/dump/

Download it and put it in the "etc" directory, without unzipping it. Edit $MY_KARAF_HOME/etc/unomi.custom.system.properties and set org.apache.unomi.geonames.forceImport to true, import should start right away. Otherwise, import should start at the next startup. Import runs in background, but can take about 15 minutes. At the end, you should have about 4 million entries in the geonames index.

5.8. REST API Security

The Apache Unomi Context Server REST API is protected using JAAS authentication and using Basic or Digest HTTP auth. You must set an admin password via UNOMI_ROOT_PASSWORD (or org.apache.unomi.security.root.password) and a health-check password via UNOMI_HEALTHCHECK_PASSWORD (or org.apache.unomi.healthcheck.password); Unomi does not ship known defaults. The default JAAS user name is karaf.

The generated package is also configured with a default SSL certificate. You can change it by following these steps :

Replace the existing keystore in $MY_KARAF_HOME/etc/keystore by your own certificate :

http://wiki.eclipse.org/Jetty/Howto/Configure_SSL

Update the keystore and certificate password in $MY_KARAF_HOME/etc/unomi.custom.system.properties file :

```
org.ops4j.pax.web.ssl.keystore=${env:UNOMI_SSL_KEYSTORE:-${karaf.etc}/keystore}
org.ops4j.pax.web.ssl.password=${env:UNOMI_SSL_PASSWORD:-changeme}
org.ops4j.pax.web.ssl.keypassword=${env:UNOMI_SSL_KEYPASSWORD:-changeme}
```

You should now have SSL setup on Karaf with your certificate, and you can test it by trying to access it on port 9443.

Changing the Karaf admin password is done by setting UNOMI_ROOT_PASSWORD or by modifying org.apache.unomi.security.root.password in the $MY_KARAF_HOME/etc/unomi.custom.system.properties file. Unomi does not ship a known default password.

5.9. Tenant Management and API Access

Apache Unomi supports multi-tenancy, allowing multiple organizations to use the same Unomi instance while keeping their data completely isolated. Each tenant has its own set of API keys for authentication.

5.9.1. Creating and Managing Tenants



All tenant management operations (create, list, update, delete, API key management) are restricted to administrators only and require JAAS authentication. These endpoints cannot be accessed using tenant API keys.

To manage tenants, you need administrator access to Unomi (karaf / your UNOMI_ROOT_PASSWORD). You can manage tenants using either the REST API or the Karaf shell commands:

Using REST API (requires admin credentials):

```
# Create a new tenant (JAAS auth required)
curl -X POST "http://localhost:8181/cxs/tenants" \
  -u "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Content-Type: application/json" \
  -d '{
    "requestedId": "mytenant",
    "properties": {
      "name": "My Company",
      "description": "My Company tenant",
      "address": "123 Main St",
      "country": "USA",
      "publicCorsAllowedOrigins": ["https://www.mycompany.com", "https://shop.mycompany.com"]
    }
  }'

# Response (HTTP 201 Created) — apiKeys are masked only; regenerate to obtain plaintext:
{
    "itemId": "mytenant",
    "itemType": "tenant",
    "status": "ACTIVE",
    "apiKeys": [
        {"maskedKey": "unomi_v1_****ab12", "keyType": "PUBLIC", "revoked": false},
        {"maskedKey": "unomi_v1_****cd34", "keyType": "PRIVATE", "revoked": false}
    ],
    "properties": {
        "name": "My Company",
        "description": "My Company tenant",
        "address": "123 Main St",
        "country": "USA",
        "publicCorsAllowedOrigins": ["https://www.mycompany.com", "https://shop.mycompany.com"]
    },
    "creationDate": "2024-03-14T10:30:00Z",
    "lastModificationDate": "2024-03-14T10:30:00Z"
}

# List all tenants (JAAS auth required)
curl -X GET "http://localhost:8181/cxs/tenants" \
  -u "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Accept: application/json"

# Get tenant details (JAAS auth required)
curl -X GET "http://localhost:8181/cxs/tenants/mytenant" \
  -u "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Accept: application/json"

# Delete a tenant (JAAS auth required)
curl -X DELETE "http://localhost:8181/cxs/tenants/mytenant" \
  -u "karaf:$UNOMI_ROOT_PASSWORD"
```

Using Karaf shell (requires admin access to Karaf console). See Shell commands for full syntax:

```
# Create a tenant (JSON properties)
unomi:crud create tenant '{"itemId":"mytenant","name":"My Company","description":"My Company
tenant"}'

# List tenants
unomi:crud list tenant

# View tenant details
unomi:crud read tenant mytenant

# Delete a tenant
unomi:crud delete tenant mytenant
```

5.9.2. API Keys and Authentication

Each tenant has two types of API keys: * Public API Key: Used for client-side operations and public endpoints * Private API Key: Used for secure operations and administrative tasks

The API keys are generated when creating a tenant, but the tenant resource only returns **masked** key metadata. To obtain the one-time plaintext value, regenerate keys (this replaces any existing key of that type):

```
# Using Karaf shell (requires admin access) — metadata only
unomi:crud read tenant mytenant

# Obtain plaintext (store immediately; it is not persisted)
curl -X POST "http://localhost:8181/cxs/tenants/mytenant/apikeys?type=PUBLIC" \
  -u "karaf:$UNOMI_ROOT_PASSWORD"
curl -X POST "http://localhost:8181/cxs/tenants/mytenant/apikeys?type=PRIVATE" \
  -u "karaf:$UNOMI_ROOT_PASSWORD"

# Response example:
{
   "apiKey": {
      "maskedKey": "unomi_v1_****ab12",
      "keyType": "PUBLIC",
      "revoked": false
   },
   "plainTextKey":
"unomi_v1_0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
}
```

To generate new API keys (requires admin access):

```
# Using REST API (JAAS auth required). Replaces any existing key of the same type.
curl -X POST "http://localhost:8181/cxs/tenants/mytenant/apikeys?type=PUBLIC&validityDays=30" \
 -u "karaf:$UNOMI_ROOT_PASSWORD"

# Response (HTTP 200 OK) — store plainTextKey immediately:
{
   "apiKey": {
      "maskedKey": "unomi_v1_****ab12",
      "keyType": "PUBLIC",
      "creationDate": "2024-03-14T10:30:00Z",
      "expirationDate": "2024-04-13T10:30:00Z",
      "revoked": false
   },
   "plainTextKey":
"unomi_v1_0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
}

# Using Karaf shell (requires admin access)
unomi:crud create apikey '{"tenantId":"mytenant","keyType":"PUBLIC","validityPeriod":30}'
```

5.9.3. Accessing API Endpoints

There are three ways to authenticate with the Unomi API:

1. Public API access (client-side / public endpoints):

```
# Get context data
curl -X POST "http://localhost:8181/cxs/context.json" \
 -H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
 -H "Content-Type: application/json" \
 -d '{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "example"
   },
   "requiredProfileProperties": ["firstName", "lastName"]
 }'
```

+ 2. Tenant private-key authentication (most private REST endpoints):

```
# Search profiles (there is no GET /cxs/profiles collection endpoint)
curl -X POST "http://localhost:8181/cxs/profiles/search" \
  --user "TENANT_ID:PRIVATE_KEY" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -d '{
    "condition": {
      "type": "matchAllCondition"
    },
    "offset": 0,
    "limit": 50
  }'
```

+ 3. System administrator (JAAS) for tenant management and cluster operations:

```
curl -X GET "http://localhost:8181/cxs/tenants" \
  --user "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Accept: application/json"
```

Authentication rules (Unomi 4.0):

1. Public paths (for example /cxs/context.json, /cxs/eventcollector, /cxs/client/*) require a valid **public** API key via X-Unomi-Api-Key (unless single-tenant compatibility mode is enabled). Browser clients may call them from any origin, but only listed origins and Unomi's own origin may use the visitor cookie; see Websites on other origins.
2. Private paths accept tenantId:privateApiKey Basic auth, or JAAS admin credentials. With JAAS, X-Unomi-Tenant-Id selects the tenant; when it is omitted, Unomi uses org.apache.unomi.rest.authentication.defaultTenantId (env UNOMI_REST_AUTHENTICATION_DEFAULTTENANTID, default default; not used yet by single-tenant compatibility mode, whose REST requests always run on default). Send X-Unomi-Tenant-Id: system only when a full administrator must work in the system context.
3. Tenant administration (/cxs/tenants) requires system administrator JAAS credentials. Without X-Unomi-Tenant-Id it runs in the system context, so it works on a fresh install before any tenant exists.
4. JAAS admin credentials are also accepted on public endpoints in place of the public API key; without X-Unomi-Tenant-Id they run in the default admin tenant (rule 2). ==== Public vs Private Endpoints

Public endpoints (requiring only public API key):

1. GET/POST /context.json

```
# Example request
curl -X GET "http://localhost:8181/cxs/context.json?sessionId=abc123" \
  -H "X-Unomi-Api-Key: 8f7d9a2c-5e4b-3f1a-9b8c-7d6e5f4a3b2c"
```

+ 2. GET/POST /eventcollector

```
# Example request
curl -X POST "http://localhost:8181/cxs/eventcollector" \
  -H "X-Unomi-Api-Key: 8f7d9a2c-5e4b-3f1a-9b8c-7d6e5f4a3b2c" \
  -H "Content-Type: application/json" \
  -d '{
    "events": [{
       "eventType": "view",
       "scope": "example",
       "source": {
          "itemId": "page1",
          "itemType": "page",
          "scope": "example"
       },
       "target": {
          "itemId": "product1",
          "itemType": "product",
          "scope": "example"
       }
    }]
  }'
```

+ 3. GET /client/*

```
# Example request
curl -X GET "http://localhost:8181/cxs/client/myapp/status" \
  -H "X-Unomi-Api-Key: 8f7d9a2c-5e4b-3f1a-9b8c-7d6e5f4a3b2c"
```

All other endpoints are considered private and require either: * JAAS authentication with admin credentials, or * Private API key authentication with tenant credentials

Example private endpoint access:

```
# Get segment details
curl -X GET "http://localhost:8181/cxs/segments/important-customers" \
  --user "mytenant:1a2b3c4d-5e6f-7g8h-9i0j-k1l2m3n4o5p6" \
  -H "Accept: application/json"

# Create a new segment
curl -X POST "http://localhost:8181/cxs/segments" \
  --user "mytenant:1a2b3c4d-5e6f-7g8h-9i0j-k1l2m3n4o5p6" \
  -H "Content-Type: application/json" \
  -d '{
    "itemId": "high-value-customers",
    "name": "High Value Customers",
    "description": "Customers with high purchase value",
    "condition": {
      "type": "profilePropertyCondition",
      "parameterValues": {
        "propertyName": "totalPurchases",
        "comparisonOperator": "greaterThan",
        "propertyValue": 1000
      }
    }
  }'
```

5.9.4. Websites on other origins (visitor cookie and CORS)

The public tracker endpoints (/cxs/context.js, /cxs/context.json, /cxs/eventcollector and /cxs/client/*) accept requests from any origin. The visitor cookie (context-profile-id by default), which identifies the visitor, is only used and set for:

- pages served from Unomi's own origin;
- origins the operator lists in org.apache.unomi.public.cors.allowedOrigins, or that the tenant of the request lists in its own settings;
- requests that carry no sign of coming from a page on another site, such as server-to-server calls.

For any other page, Unomi answers without credentialed CORS headers and ignores the cookie: the request gets a new anonymous profile, and the visitor's cookie is not replaced.

Choosing a setup

Website and Unomi	List needed	Notes
Same origin: a reverse proxy serves the website and /cxs on one host (https://www.example.com/cxs/)	No	Recommended. The cookie is a first-party cookie and needs no CORS at all.

Website and Unomi	List needed	Notes
Another subdomain of the same site (https://www.example.com and https://cdp.example.com)	Yes	List https://www.example.com. SameSite=Lax is enough. Set org.apache.unomi.profile.cookie.domain=example.com if the cookie must also be visible to other subdomains.
Another site (https://www.shop.com and https://cdp.example.com)	Yes	List https://www.shop.com, serve Unomi over HTTPS and set SameSite=None. The cookie is then a third-party cookie: it does not work in Safari, which blocks it; Firefox partitions it per top-level site; and Chrome users may block it. Prefer one of the setups above.

A response can only set the cookie on Unomi's own host, or on a parent domain set with org.apache.unomi.profile.cookie.domain. It cannot place a cookie on the website's domain. To get a first-party cookie for the website, serve Unomi from that website's origin or from one of its subdomains.

Settings

```
# etc/custom.system.properties (or the environment variables)
org.apache.unomi.public.cors.allowedOrigins=${env:UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS:-}
org.apache.unomi.profile.cookie.sameSite=${env:UNOMI_PROFILE_COOKIE_SAMESITE:-Lax}
org.apache.unomi.public.cors.matchReferer=${env:UNOMI_PUBLIC_CORS_MATCH_REFERER:-false}
org.apache.unomi.public.cors.tenantOrigins=${env:UNOMI_PUBLIC_CORS_TENANT_ORIGINS:-true}
```

Example:

```
export
UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS='https://www.example.com,https://shop.example.com:84
43'
```

Another site only (Unomi served over HTTPS):

```
export UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS='https://www.shop.com'
export UNOMI_PROFILE_COOKIE_SAMESITE=None
```

- Entries are comma-separated and give a scheme (http or https), a host and an optional port. They are compared the way browsers send the Origin header: case does not matter, a default port (:443, :80) may be left out, and a trailing / is ignored.
- Wildcards, paths and null are not accepted. A rejected entry is logged as a warning when the setting is applied, and so is a list in which no entry is valid.
- The list is empty by default: no other origin may use the cookie.
- A listed origin is named exactly in the response, with Access-Control-Allow-Credentials: true and

Vary: Origin, preflight requests included. An unlisted origin gets Access-Control-Allow-Origin: * without credentials.

- SameSite accepts Lax (default), Strict and None. None is used only when the cookie is Secure and origins can be listed: the server list is not empty, or tenant lists are on (the default). Otherwise Unomi uses Lax and logs a warning. Either way, only listed websites can use the cookie.
- tenantOrigins (default true) lets each tenant list its own websites; see Websites listed by each tenant. Only false turns it off.
- matchReferer (default false) lets listed websites that embed the tracker with a <script> tag keep their visitors; see Embedding context.js with a script tag. Only true turns it on; any other value is logged as a warning, and so is true when nothing can be listed (an empty server list with tenant lists off).
- Changes are applied without a restart and are parsed once, not on every request.

Do not:

- list origins you do not control, or every subdomain "just in case": a listed page can act as the visitor;
- use * (it is refused);
- list the http:// origin of a website that also runs over HTTPS: list the https:// one, which is what its pages send;
- turn on matchReferer "to make it work": it trusts everything the listed websites frame (see Embedding context.js with a script tag);
- set SameSite=None for a same-site setup: Lax already works there.

Websites listed by each tenant

On a server with many tenants, each tenant lists its own websites, so tenants can be created and deleted without changing the server configuration. The list is the tenant's publicCorsAllowedOrigins property, a JSON array of origins or a comma-separated string, set by a system administrator through /cxs/tenants when the tenant is created or updated (see Creating and Managing Tenants):

```
# PUT replaces the tenant's settings (but never its API keys): read it, change the property, send it back.
curl -s -u "karaf:$UNOMI_ROOT_PASSWORD" "http://localhost:8181/cxs/tenants/mytenant" \
  | jq '.properties.publicCorsAllowedOrigins = ["https://www.mycompany.com"]' \
  | curl -X PUT "http://localhost:8181/cxs/tenants/mytenant" \
      -u "karaf:$UNOMI_ROOT_PASSWORD" -H "Content-Type: application/json" --data-binary @-
```

- The origins follow the same rules as the server list. An invalid entry is refused with 400 when the tenant is saved, so a provisioning script fails instead of a website silently losing its visitors.
- A tenant's origins only apply to requests made with that tenant's public API key. The same website calling with another tenant's key is served without the visitor cookie.
- The server list still applies to every tenant. Tenants can only add origins for themselves; they cannot change the server settings.

- A browser's preflight request carries no API key, so it is answered for any origin the server or a tenant lists. That only lets the browser send the real request, which is then checked against its own tenant.
- A tenant created, updated or deleted through /cxs/tenants applies at once on the node that handled it. Other nodes pick up a new tenant on its first request, and any other change, such as an origin added to or removed from an existing tenant, within 30 seconds.
- PUT /cxs/tenants/{tenantId} replaces the tenant's settings but never its API keys, which are managed through /cxs/tenants/{tenantId}/apikeys, nor its creation date; those fields in the request body are ignored. A tenant read with GET can therefore be sent back as is. The write fails with 409 instead of undoing a concurrent API key change; retry it.
- Tenants saved without going through /cxs/tenants, for example with the developer shell commands, are not validated: invalid entries are left out and logged as a warning.
- SameSite, HttpOnly and the cookie domain stay server settings: all tenants share one visitor cookie on a Unomi host.
- org.apache.unomi.public.cors.tenantOrigins=false (UNOMI_PUBLIC_CORS_TENANT_ORIGINS) turns tenant lists off; only the server list then counts.

How Unomi recognises the page

Current browsers send Origin on every cross-origin request and on every POST, same-origin ones included, so its presence alone does not mean another site. Unomi decides as follows:

1. An Origin on the list is allowed.
2. Without Origin, as for a <script src=".../context.js"> tag, and only when matchReferer is on, the browser-set Referer is matched against the list instead.
3. Otherwise the browser's Sec-Fetch-Site header decides: only same-origin and none (a page typed in the address bar) are allowed. It takes precedence over the host comparison below.
4. A request with neither Origin nor Sec-Fetch-Site is allowed. That is a server-side caller, or a GET from a browser that does not send Sec-Fetch-Site (Chrome before 76, Firefox before 90, Safari before 16.4), which cannot be told apart from it. In such a browser, another site's <script> or <img> can therefore use the visitor cookie.

   This only matters with SameSite=None: with the default Lax, a browser does not send the cookie to another site's embeds at all, and Safari blocks third-party cookies whatever their SameSite. Current versions of Chrome, Edge, Firefox and Safari all send Sec-Fetch-Site.

5. For older browsers that send Origin but not Sec-Fetch-Site (for example Safari before 16.4), a page on the same host as Unomi counts as Unomi's own origin. A proxy that rewrites Host must then send X-Forwarded-Host.

Behind a reverse proxy

The cookie is marked Secure when the request reached Unomi over HTTPS. Behind a proxy that terminates TLS, that is only the case when the proxy's address is in org.apache.unomi.ip.trustedProxies

(the default already covers private networks) and every value it sends in X-Forwarded-Proto is https. Without it, a configured SameSite=None falls back to Lax, and a warning is logged once per configuration change.

Embedding context.js with a script tag

A page that loads the tracker with a plain <script src="https://cdp.example.com/cxs/context.js"> (no crossorigin attribute) sends no Origin header, only a Referer. Public endpoints need an API key header, which a script tag cannot send, so this only applies in single-tenant compatibility mode, where clients written for 2.x often embed context.js this way.

By default, in browsers that send Sec-Fetch-Site, such a request from another origin is served without the visitor cookie, even from a listed website: every page view starts a new anonymous profile. By default, once origins are listed, Unomi logs a warning about it once per configuration change. To accept it, set org.apache.unomi.public.cors.matchReferer=true (UNOMI_PUBLIC_CORS_MATCH_REFERER). A Referer naming a listed page is then accepted.

Before turning it on:

- It trusts everything a listed page frames. A sandboxed <iframe srcdoc> or about:blank frame, such as an ad, reports the page that created it as its Referer, so its scripts would receive the visitor's profile. Turn it on only for websites that do not frame untrusted content, or switch them to the shipped tracker, which calls context.json with credentials and needs no Referer.
- The listed pages must send their origin in Referer to other sites. The browser default (strict-origin-when-cross-origin) does. no-referrer, same-origin or a referrerpolicy attribute on the <script> tag that removes it makes every page view start a new profile. So does an HTTPS page loading Unomi over HTTP.

Compatibility with Unomi 2.x and 3.x

In 2.x and 3.x the public endpoints allowed credentialed CORS for every origin, and nothing checked where a request came from. The cookie was always SameSite=Lax, so only pages on the same site could use it. A website on another site kept the profile id itself and sent it as a body profileId, which 4.0 ignores for public callers.

Deployment in 2.x or 3.x	What it needs in 4.0
Website and Unomi on the same origin	Nothing.
Website on another subdomain using the cookie	List the website origin, in the server list or in the tenant's own list. SameSite=Lax is enough.
Website on another site sending body profileId	List the website origin, serve Unomi over HTTPS and set SameSite=None, or serve Unomi from the website's origin. Expect the cookie to be blocked in Safari.

Deployment in 2.x or 3.x	What it needs in 4.0
2.x client embedding context.js with a <script> tag (single-tenant compatibility mode)	List the website origin and set matchReferer=true, after reading the trade-offs; or move to the shipped tracker. A website on another site also needs HTTPS with SameSite=None: in 2.x only same-site embeds kept their profile.
Server-side caller sending no Origin	Nothing for the origin check. A body profileId needs a trusted caller, as described in the migration guide.

Other behaviour that changed:

- A refused request receives no Set-Cookie, so the visitor's cookie is left untouched.
- A malformed profile cookie still answers 400, but only for requests that may use the cookie; a refused request ignores it.
- The cookie is marked Secure behind a trusted TLS proxy, so a website that mixes HTTP and HTTPS pages loses the cookie on its HTTP pages.
- org.apache.unomi.profile.cookie.httpOnly now defaults to true (it was false in 2.x and 3.x) and is turned off only by false. In 4.0.0-rc1 a mistyped value turned HttpOnly off; it now keeps it on and is logged as a warning.

"Download my data" links

/cxs/client/myprofile/{format} returns the visitor's own profile and changes nothing. In single-tenant compatibility mode, where it needs no API key, a plain link to it on the website is accepted as a top-level navigation (Sec-Fetch-Mode: navigate, Sec-Fetch-Dest: document) even from a page Unomi does not otherwise recognise: the response opens in the visitor's own window, and no other page can read it. A link from another site only carries the cookie with SameSite=Lax (the default) or None; with Strict it still answers 404. The tracker endpoints never accept a navigation this way.

Tracker without cookies

A deployment that does not need the visitor cookie can set sendCredentials: false in the tracker's wemInitConfig (tracker 2.0.0 or later, see the web tracker). Requests are then sent without cookies and work with the default settings, but each request starts a new anonymous profile.

5.9.5. Admin UI origins (credentialed CORS)

The admin REST API under /cxs (other than the public tracker endpoints above) and GraphQL under /graphql do not advertise credentialed CORS by default. A browser on another origin therefore cannot read admin responses using the operator's HTTP Basic credentials.

If an admin UI is served from a different origin than Unomi, list that origin:

```
org.apache.unomi.admin.cors.allowedOrigins=${env:UNOMI_ADMIN_CORS_ALLOWED_ORIGINS:-}
```

Example: https://admin.example.com. Same-origin deployments (a reverse proxy serving the UI and Unomi together) need no entry. This list is separate from org.apache.unomi.public.cors.allowedOrigins, which only covers the visitor cookie on public tracker endpoints.

This list covers HTTP requests only. GraphQL subscriptions over WebSocket accept only handshakes from the same host as Unomi (or with no Origin), whether or not the origin is listed here.

5.9.6. GraphQL query limits

GraphQL under /graphql bounds document size and analysis cost. Configure them in etc/custom.system.properties (or the matching environment variables); they are applied through etc/org.apache.unomi.graphql.cfg. A value of 0 turns that check off.

```
org.apache.unomi.graphql.max.query.length.chars=${env:UNOMI_GRAPHQL_MAX_QUERY_LENGTH_CHARS:-262144}
org.apache.unomi.graphql.max.request.body.bytes=${env:UNOMI_GRAPHQL_MAX_REQUEST_BODY_BYTES:-524288}
org.apache.unomi.graphql.max.query.depth=${env:UNOMI_GRAPHQL_MAX_QUERY_DEPTH:-24}
org.apache.unomi.graphql.max.query.complexity=${env:UNOMI_GRAPHQL_MAX_QUERY_COMPLEXITY:-3000}
```

The defaults leave GraphiQL introspection and the public getProfile / processEvents fields usable. Oversized requests are refused with HTTP 413 before the document is parsed. Changes to these settings apply to HTTP requests as soon as the configuration is updated.

GraphQL subscriptions over WebSocket are handled differently:

- They keep the depth and complexity limits in force when the GraphQL servlet started. Restart Unomi (or the GraphQL bundle) to apply new values to them.
- The length and body size limits do not apply to them. A subscription message is limited to 1 MB by the WebSocket buffer, and a socket must authenticate before it can run anything.

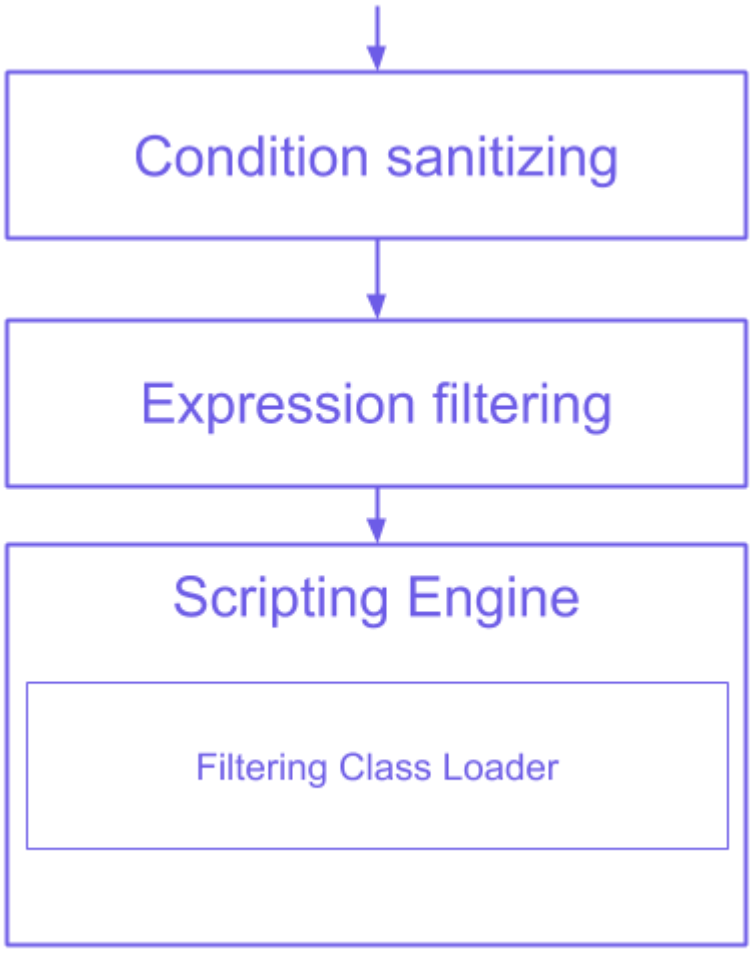
HTTP GET may run query operations only. Mutations, subscriptions, and queries that set createIfMissing: true on getProfile (which saves a profile) must use HTTP POST with Content-Type: application/json (a charset parameter is fine).

5.10. Scripting security

5.10.1. Multi-layer scripting filtering system

The scripting security system is multi-layered.

For requests coming in through the /cxs/context.json endpoint, the following flow is used to secure incoming requests:



Conditions submitted through the context.json public endpoint are first sanitized, meaning that any scripting directly injected is removed. However, as conditions can use sub conditions that include scripting, only the first directly injected layer of scripts are removed.

The second layer is the expression filtering system, that uses an allow-listing mechanism to only accept pre-vetted expressions (through configuration and deployment on the server side). Any unrecognized expression will not be accepted.

Finally, once the script starts executing in the scripting engine, a filtering class loader restricts which classes a script may load **by name**. No org.mvel2 class is on the allow list, so an expression can never name an MVEL class (its eval engines, its JSR-223 script engine, its shell or its compiler) through new, a static reference or an import; MVEL's own internals are loaded from the MVEL bundle and are unaffected. As a safety net for a loosened configuration, the public MVEL eval APIs are also always denied in code, including when they are referenced through inner classes, array types, or names with invisible characters, even if the forbid list is emptied or the allow list is set to all.

This class-loader restriction is defense in depth, not a complete sandbox. It only gates class names that the engine resolves through it; it cannot stop reflection that a running expression performs on objects it already holds, such as getClass(), getClassLoader().loadClass(…) or Class.forName(…), because those calls resolve through the JDK and MVEL bundle class loaders, not this one. The expression filtering layer above is therefore the primary, non-optional control: its allow-list of pre-vetted expressions and its forbid patterns (which reject reflection tokens such as getClass, forName, invoke and Class) are what actually stop arbitrary expressions from reaching the engine. Do not disable expression filtering or trim

its forbid list on the assumption that the class loader alone will contain a hostile expression.

This multi-layered approach makes it possible to retain a high level of security even if one layer is poorly configured or abused.

For requests coming in through the secure APIs such as rules, only the condition sanitizing step is skipped, otherwise the rest of the filtering system is the same.

5.10.2. Scripts and expressions

Apache Unomi allows using different types of expressions in the following subsystems:

- context.json filters and personalization queries
- rule conditions and actions parameters

Apache Unomi uses an integrated scripting language to provide this functionality: MVEL. MVEL is used in rule actions as in the following example:

From https://github.com/apache/unomi/blob/unomi-1.5.x/plugins/baseplugin/src/main/resources/META-INF/cxs/rules/sessionAssigned.json:

```
{
 "metadata": {
  "id": "_ajhg9u2s5_sessionAssigned",
  "name": "Session assigned to a profile",
  "description": "Update profile visit information",
  "readOnly":true
 },

 "condition": {
  "type": "booleanCondition",
  "parameterValues": {
   "subConditions":[
    {
     "type": "eventTypeCondition",
     "parameterValues": {
      "eventTypeId": "sessionCreated"
     }
    },
    {
     "type": "eventTypeCondition",
     "parameterValues": {
      "eventTypeId": "sessionReassigned"
     }
    }

    ],
   "operator":"or"

  }
 },
```

```
  "actions": [
    {
      "parameterValues": {
        "setPropertyName": "properties.previousVisit",
        "setPropertyValue": "profileProperty::lastVisit",
        "storeInSession": false
      },
      "type": "setPropertyAction"
    },
    {
      "parameterValues": {
        "setPropertyName": "properties.lastVisit",
        "setPropertyValue": "now",
        "storeInSession": false
      },
      "type": "setPropertyAction"
    },
    {
      "parameterValues": {
        "setPropertyName": "properties.nbOfVisits",
        "setPropertyValue": "script::profile.properties.?nbOfVisits != null ? (profile.properties.nbOfVisits
+ 1) : 1",
        "storeInSession": false
      },
      "type": "setPropertyAction"
    },
    {
      "parameterValues": {
        "setPropertyName": "properties.totalNbOfVisits",
        "setPropertyValue": "script::profile.properties.?totalNbOfVisits != null ?
(profile.properties.totalNbOfVisits + 1) : 1",
        "storeInSession": false
      },
      "type": "setPropertyAction"
    }
  ]

}
```

As we see in the above example, we use an MVEL script with the setPropertyAction to set a property value. Starting with version 1.5.2, any expression use in rules MUST be allow-listed.

By default, Apache Unomi comes with some built-in allowed expressions that cover all the internal uses cases.

Default allowed MVEL expressions (from https://github.com/apache/unomi/blob/unomi-1.5.x/plugins/baseplugin/src/main/resources/META-INF/cxs/expressions/mvel.json) :

```
[
  "\\Q'systemProperties.goals.'+goalId+'TargetReached'\\E",
  "\\Q'now-'+since+'d'\\E",
  "\\Q'scores.'+scoringPlanId\\E",
  "\\QminimumDuration*1000\\E",
  "\\QmaximumDuration*1000\\E",
  "\\Qprofile.properties.?nbOfVisits != null ? (profile.properties.nbOfVisits + 1) : 1\\E",
  "\\Qprofile.properties.?totalNbOfVisits != null ? (profile.properties.totalNbOfVisits + 1) : 1\\E",
  "\\Qsession != null ? session.size + 1 : 0\\E",
  "\\Q'properties.optimizationTest_'+event.target.itemId\\E",
  "\\Qevent.target.properties.variantId\\E",
  "\\Qprofile.properties.?systemProperties.goals.\\E[\\w\\_]*\\QReached != null ?
(profile.properties.systemProperties.goals.\\E[\\w\\_]*\\QReached) : 'now'\\E",
  "\\Qprofile.properties.?systemProperties.campaigns.\\E[\\w\\_]*\\QEngaged != null ?
(profile.properties.systemProperties.campaigns.\\E[\\w\\_]*\\QEngaged) : 'now'\\E"
]
```

If you require or are already using custom expressions, you should add a plugin to Apache Unomi to allow for this. The choice of a plugin was to make sure only system administrators and solution developers could provide such a list, avoiding the possibility to provide it through an API call or another security sensitive deployment mechanism.

There is another way of allow-listing expressions through configuration, see the “scripting configuration parameters” section below.

Procedure to add allowed expressions:

1. Create a new Apache Unomi plugin project.
2. Create a JSON file in src/main/resources/META-INF/cxs/expressions/mvel.json with an array of regular expressions that will contain the allowed expressions.
3. Build the project and deploy it to Apache Unomi

Warning: Do not make regular expressions too general. They should actually be as specific as possible to avoid potential injection of malicious code.

5.10.3. Scripting expression filtering configuration parameters

Alongside with the allow-listing technology, there are new configuration parameters to control the security of the scripting engines:

```
# These parameters control the list of classes that are allowed or forbidden when executing expressions.
# Forbid is applied first. No org.mvel2 class is allowed, so expressions cannot name an MVEL class; MVEL's
# own internals load from the MVEL bundle, not this filter. Public MVEL eval APIs are also always denied in
# code, even if this forbid list is emptied or the allow list is set to "all".
org.apache.unomi.scripting.allow=${env:UNOMI_ALLOW_SCRIPTING_CLASSES:-org.apache.unomi.api.Event,org.apache.unomi.api.Profile,org.apache.unomi.api.Session,org.apache.unomi.api.Item,org.apache.unomi.api.CustomItem,java.lang.Object,java.util.Map,java.util.HashMap,java.lang.Integer,java.lang.String}
org.apache.unomi.scripting.forbid=${env:UNOMI_FORBID_SCRIPTING_CLASSES:-}

# This parameter controls the whole expression filtering system. It is not recommended to turn it off. The main reason to turn it off would be to check if it is interfering with something, but it should always be active in production.
org.apache.unomi.scripting.filter.activated=${env:UNOMI_SCRIPTING_FILTER_ACTIVATED:-true}

# The following parameters control the filtering using regular expressions for each scripting sub-system.
# The "collections" parameter tells the expression filtering system which configurations to expect. By default only MVEL is accepted values, but in the future these might be replaced by new scripting sub-systems.
org.apache.unomi.scripting.filter.collections=${env:UNOMI_SCRIPTING_FILTER_COLLECTIONS:-mvel}

# For each scripting sub-system, there is an allow and a forbid property that reference a .json files,
# you can either edit this files or reference your own file directly in the following config.
# Note: You can add new expressions to the "allow" file, although it is better to add them inside any plugins you may be adding.
#       This configuration is only designed to compensate for the cases where something was not properly designed or to deal with compatibility issues.
#       Just be VERY careful to make your patterns AS SPECIFIC AS POSSIBLE in order to avoid introducing a way to abuse the expression filtering.
# Note: It is NOT recommended to change the built-in "forbid" value unless you are having issues with its value.
# Note: mvel-allow.json contains an empty array: [], this mean nothing is allowed, so far.
#       If you want to allow all expression, just remove the property org.apache.unomi.scripting.filter.mvel.allow, but this is not recommended
#       It's better to list your expressions, and provide them in the mvel-allow.json file
#       example: ["\\Qsession.size + 1\\E"]
org.apache.unomi.scripting.filter.mvel.allow=${env:UNOMI_SCRIPTING_FILTER_MVEL_ALLOW:-${karaf.etc}/mvel-allow.json}
org.apache.unomi.scripting.filter.mvel.forbid=${env:UNOMI_SCRIPTING_FILTER_MVEL_FORBID:-${karaf.etc}/mvel-forbid.json}

# This parameter controls the condition sanitizing done on the ContextServlet (/cxs/context.json). If will remove any expressions that start with "script::". It is not recommended to change this value, unless you run into compatibility issues.
org.apache.unomi.security.personalization.sanitizeConditions=${env:UNOMI_SECURITY_SANITIZEPERSONALIZATIONCONDITIONS:-true}
```

5.10.4. Groovy Actions

Groovy actions offer the ability to define a set of actions and action types (aka action descriptors) purely from Groovy scripts defined at runtime.

Initially submitted to Unomi through a purpose-built REST API endpoint, Groovy actions are then stored in the persistence layer (Elasticsearch or OpenSearch). When an event matches a rule configured to execute an action, the corresponding action is fetched from persistence and executed.

Anatomy of a Groovy Action

To be valid, a Groovy action must follow a particular convention which is divided in two parts:

- An annotation used to define the associated action type
- The function to be executed

Placed right before the function, the "@"Action annotation contains a set of parameter detailing how the action should be triggered.

Table 1. @Action annotation

Field name	Type	Required	Description
id	String	YES	Id of the action
actionExecutor	String	YES	Action executor contains the name of the script to call for the action type and must be prefixed with "**groovy:**". The prefix indicates to Unomi which dispatcher to use when processing the action. The name must be the file name of the groovy file containing the action without the extension (**groovy:<filename>**).
name	String		Action name
hidden	Boolean		Define if the action is hidden or not. It is usually used to hide objects in a UI.
parameters	List<Parameter>		The parameters of the actions, also defined by annotations

Field name	Type	Required	Description
systemTags	List<String>		A (reserved) list of tags for the associated object. This is usually populated through JSON descriptors and is not meant to be modified by end users. These tags may include values that help classify associated objects.

The function contained within the Groovy Action must be called execute() and its last instruction must be an integer.

This integer serves as an indication whether the values of the session and profile should be persisted. In general, the codes used are defined in the EventService interface.

Each groovy actions extends by default a Base script defined here

REST API

Actions can be deployed/updated/deleted via the dedicated /cxs/groovyActions rest endpoint.

Deploy/update an Action:

```
curl -X POST 'http://localhost:8181/cxs/groovyActions' \
--user "TENANT_ID:PRIVATE_KEY" \
--form 'file=@"<file location>"'
```



Replace TENANT_ID and PRIVATE_KEY with your actual tenant ID and private API key.

A Groovy Action can be updated by submitting another Action with the same id.

Delete an Action:

```
curl -X DELETE 'http://localhost:8181/cxs/groovyActions/<Action id>' \
--user "TENANT_ID:PRIVATE_KEY"
```

Note that when a groovy action is deleted by the API, the action type associated with this action will also be deleted.

Hello World!

In this short example, we're going to create a Groovy Action that will be adding "Hello world! " to the logs whenever a new view event is triggered.

The first step consists in creating the groovy script on your filesystem, start by creating the file helloWorldGroovyAction.groovy:

```
@Action(id = "helloWorldGroovyAction",
        actionExecutor = "groovy:helloWorldGroovyAction",
        parameters = [@Parameter(id = "location", type = "string", multivalued = false)])
def execute() {
   logger.info("Hello {}", action.getParameterValues().get("location"))
   EventService.NO_CHANGE
}
```

As the last instruction of the script is EventService.NO_CHANGE, data will not be persisted.

Once the action has been created you need to submit it to Unomi (from the same folder as helloWorldGroovyAction.groovy).

```
curl -X POST 'http://localhost:8181/cxs/groovyActions' \
--user "karaf:${UNOMI_ROOT_PASSWORD}" \
--form 'file=@helloWorldGroovyAction.groovy'
```



Groovy Actions REST requires the system ADMINISTRATOR role (not a tenant private key). See client-facing hardening.

Important: A bug (UNOMI-847) in Apache Unomi 2.5 and lower requires the filename of a Groovy file being submitted to be the same as the id of the Groovy action (as per the example above).

Finally, register a rule to trigger execution of the groovy action:

```
curl -X POST 'http://localhost:8181/cxs/rules' \
--user "TENANT_ID:PRIVATE_KEY" \
--header 'Content-Type: application/json' \
--data-raw '{
 "metadata": {
  "id": "scriptGroovyActionRule",
  "name": "Test Groovy Action Rule",
  "description": "A sample rule to test Groovy actions"
 },
 "condition": {
    "type": "eventTypeCondition",
    "parameterValues": {
     "eventTypeId": "view"
    }
 },
 "actions": [
  {
    "parameterValues": {
     "location": "world!"
    },
    "type": "helloWorldGroovyAction"
  }
 ]
}'
```

Note that this rule contains a "location" parameter, with the value "world!", which is then used in the log message triggered by the action.

You can now use unomi to trigger a "view" event and see the corresponding message in the Unomi logs.

Once you're done with the Hello World! action, it can be deleted using the following command:

```
curl -X DELETE 'http://localhost:8181/cxs/groovyActions/helloWorldGroovyAction' \
--user "karaf:${UNOMI_ROOT_PASSWORD}"
```

And the corresponding rule can be deleted using the following command:

```
curl -X DELETE 'http://localhost:8181/cxs/rules/scriptGroovyActionRule' \
--user "TENANT_ID:PRIVATE_KEY"
```

Inject an OSGI service in a groovy script

It's possible to use the services provided by unomi directly in the groovy actions.

In the following example, we are going to create a groovy action that displays the number of existing profiles by using the profile service provided by unomi.

```
import org.osgi.framework.Bundle
import org.osgi.framework.BundleContext
import org.osgi.framework.FrameworkUtil
import org.apache.unomi.groovy.actions.GroovyActionDispatcher
import org.osgi.framework.ServiceReference
import org.slf4j.Logger
import org.slf4j.LoggerFactory

final Logger LOGGER = LoggerFactory.getLogger(GroovyActionDispatcher.class.getName());

@Action(id = "displayNumberOfProfilesAction", actionExecutor =
"groovy:DisplayNumberOfProfilesAction", description = "Display the number of existing profiles")
def execute() {

    // Use OSGI function to get the bundleContext
    Bundle bundle = FrameworkUtil.getBundle(GroovyActionDispatcher.class);
    BundleContext context = bundle.getBundleContext();

    // Get the service reference
    ServiceReference<ProfileService> serviceReference =
context.getServiceReference(ProfileService.class);

    // Get the service you are looking for
    ProfileService profileService = context.getService(serviceReference);

    // Example of displaying the number of profile
    LOGGER.info("Display profile count")
    LOGGER.info("{}", profileService.getAllProfilesCount().toString())

    return EventService.NO_CHANGE
}
```

Known limitation

Only the services accessible by the class loader of the GroovyActionDispatcher class can be used in the groovy actions. That includes the services in the following packages:

```
org.apache.unomi.api.actions
org.apache.unomi.api.services
org.apache.unomi.api
org.apache.unomi.groovy.actions
org.apache.unomi.groovy.actions.annotations
org.apache.unomi.groovy.actions.services
org.apache.unomi.metrics
org.apache.unomi.persistence.spi
org.apache.unomi.services.actions;version
```

5.10.5. Scripting roadmap

Scripting will probably undergo major changes in future versions of Apache Unomi, with the likely retirement of MVEL in favor of Groovy Actions detailed above.

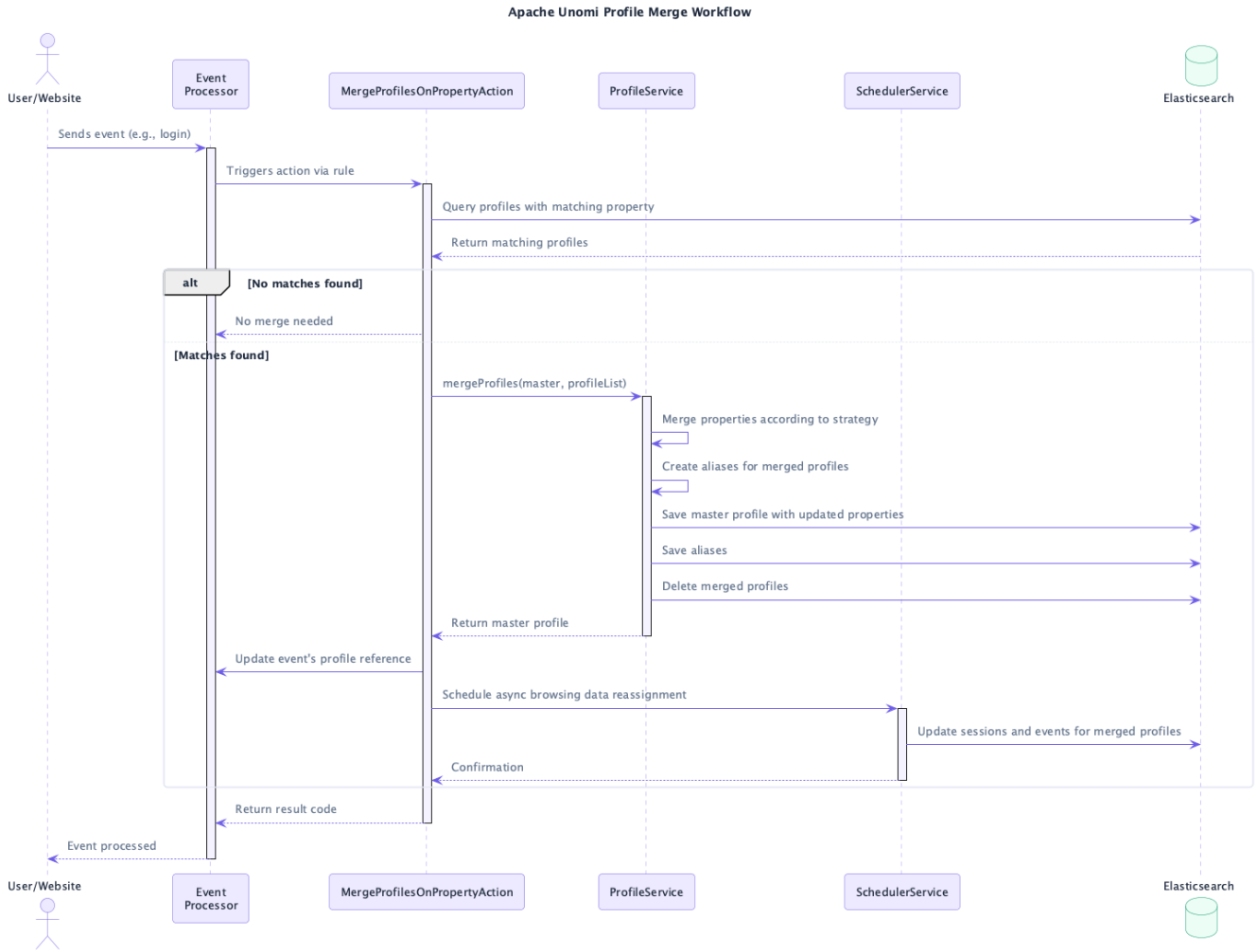
These changes will not happen on maintenance versions of Apache Unomi, only in the next major version. Maintenance versions will of course maintain compatibility with existing scripting solutions.

5.11. Automatic profile merging

Apache Unomi is capable of merging profiles based on a common property value. In order to use this, you must add the MergeProfileOnPropertyAction to a rule (such as a login rule for example), and configure it with the name of the property that will be used to identify the profiles to be merged. An example could be the "email" property, meaning that if two (or more) profiles are found to have the same value for the "email" property they will be merged by this action.

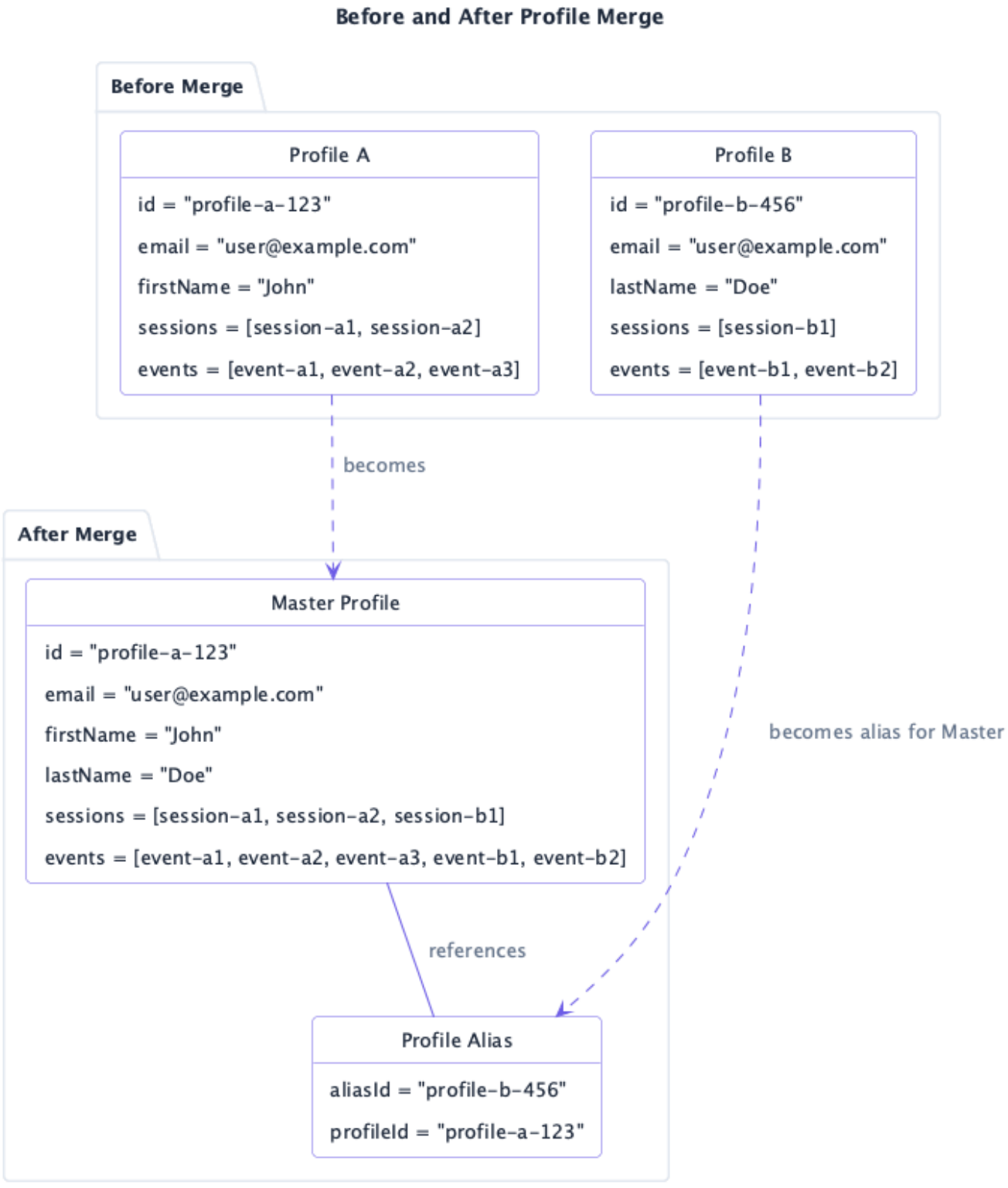
5.11.1. How profile merging works

When a profile merge is triggered, the following process takes place:



5.11.2. Before and after a profile merge

This diagram illustrates the state of profiles before and after a merge:



Starting from Unomi 2:

- The oldest profile (by firstVisit timestamp) becomes the master profile by default
- All properties from merged profiles are combined into the master profile
- An alias is created for each merged profile, pointing to the master profile
- The original merged profiles are deleted
- All sessions and events from merged profiles are reassigned to the master profile asynchronously

5.11.3. Configuration parameters

The MergeProfilesOnPropertyAction supports the following parameters:

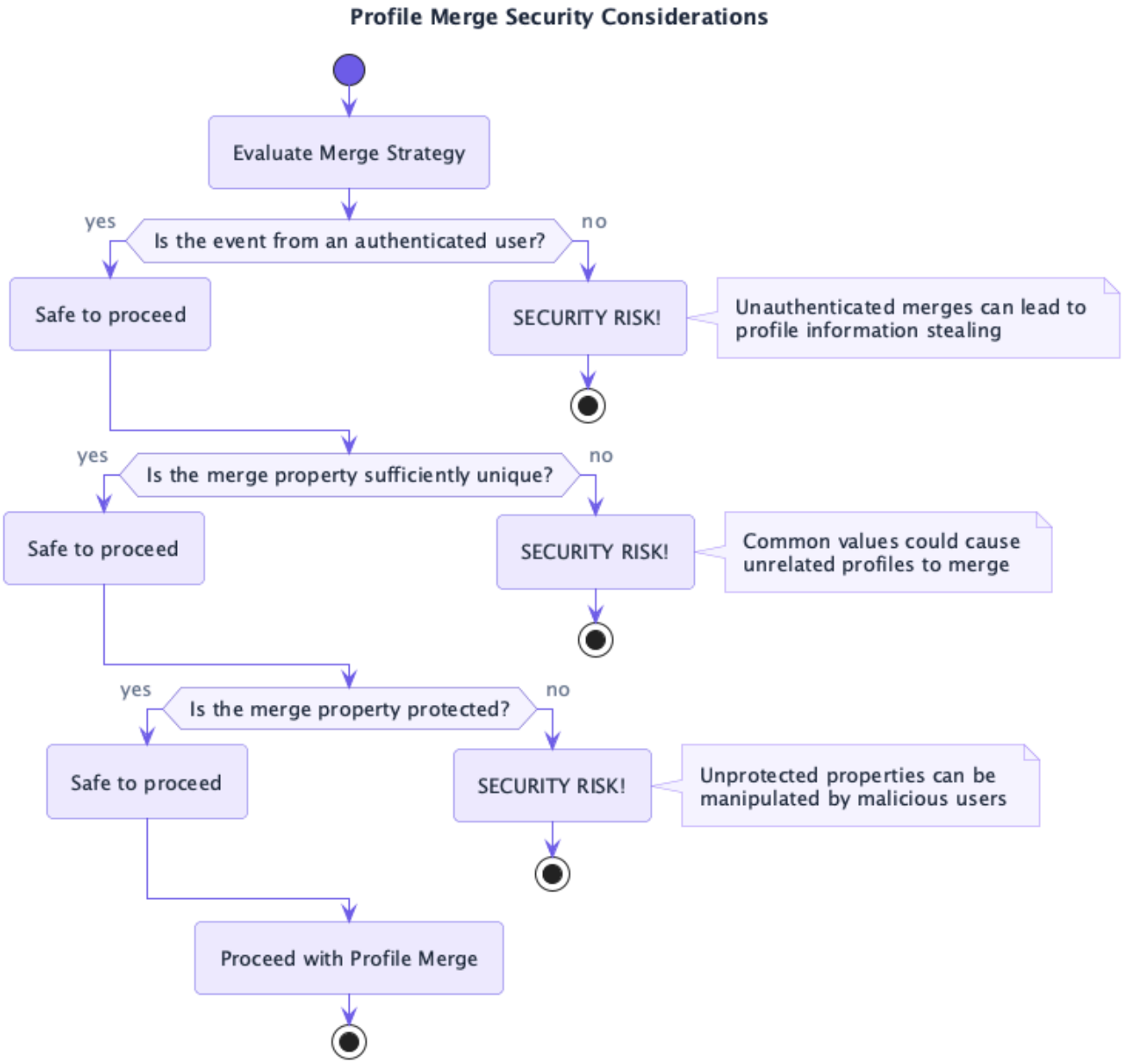
Parameter	Type	Description	Required
mergeProfilePropertyName	String	The system property name used to identify profiles for merging	Yes
mergeProfilePropertyValue	String	The value to match against the property. Often set dynamically from event properties using eventProperty::target.properties(propertyName)	Yes
forceEventProfileAsMaster	Boolean	If true, forces the current event's profile to be the master profile after merging. If false, the oldest profile (by firstVisit) becomes the master.	No (defaults to false)

5.11.4. Security considerations



Never trigger profile merges from unauthenticated or public-key-only operations such as form submissions or public-facing context events. Merging into another profile (or switching identity after a merge) requires a **trusted** caller — system administrator or tenant administrator (private key). Always verify user identity on your application server before emitting a merge-triggering event. See client-facing hardening.

The following diagram highlights key security considerations:



Key security recommendations:

- **Always authenticate users** before performing profile merges
- **Use protected properties** that require authentication to modify
- **Choose unique identifiers** like verified email or account ID
- **Implement rate limiting** for merge operations to prevent abuse
- **Consider additional verification** before merging profiles with sensitive data

Requiring HTTPS for merges

A tenant private key or an administrator password sent over plain HTTP can be read by anyone on the network path, and whoever holds it can merge profiles. Set org.apache.unomi.plugins.base.requireSecureTransport=true (environment variable UNOMI_REQUIRE_SECURE_TRANSPORT) in production. It is false by default so that a local development install keeps working over HTTP.

When it is on, a caller trusted for its private key or administrator login may only merge profiles, record a merge identifier, update another profile or write systemProperties if the request arrived over HTTPS. The same request over HTTP is still processed, but those steps are refused and a warning is logged.

If a reverse proxy terminates TLS in front of Unomi, the request Unomi sees is plain HTTP. Unomi then accepts the proxy's word only when both of these hold:

- the proxy's address is listed in org.apache.unomi.ip.trustedProxies (UNOMI_IP_TRUSTED_PROXIES). The default list covers every private address range; narrow it to your proxies, or any host on those networks can claim HTTPS.
- the proxy sends X-Forwarded-Proto: https. Configure it to overwrite the header: if any listed protocol is not https, the request is treated as plain HTTP.

Changing org.apache.unomi.ip.trustedProxies in custom.system.properties needs a restart. The setting does not apply to X-Unomi-Peer keys in single-tenant compatibility mode, which keep their Unomi 3.0 behaviour, nor to merges Unomi runs itself outside an HTTP request.

5.11.5. Troubleshooting profile merges

If profiles aren't merging as expected, check:

1. The merge property exists on both profiles with exactly matching values
2. The merge property is stored as a system property (systemProperties.mergeIdentifier)
3. The rule containing the merge action is correctly triggered
4. The profiles aren't personas or anonymous profiles (which are skipped)

5.11.6. Performance considerations

- The maxProfilesInOneMerge parameter (default: 50) limits how many profiles are merged in a single operation
- Large numbers of merges can impact system performance
- Session and event reassignment happens asynchronously to prevent blocking the event processing pipeline
- Consider the impact of merges on your Elasticsearch cluster, especially for high-traffic sites

To test, simply configure the action in the "login" or "facebookLogin" rules and set it up on the "email" property. Upon sending one of the events, all matching profiles will be merged.

5.12. Securing a production environment

Before going live with a project, you should *absolutely* read the following section that will help you setup a proper secure environment for running your context server.

Step 1: Install and configure a firewall

You should setup a firewall around your cluster of context servers and/or Elasticsearch nodes. If you have an application-level firewall you should only allow the following connections open to the whole world :

- http://localhost:8181/cxs/context.js
- http://localhost:8181/cxs/eventcollector

All other ports should not be accessible to the world.

For your Apache Unomi client applications (such as the Jahia CMS), you will need to make the following ports accessible :

```
8181 (Context Server HTTP port)
9443 (Context Server HTTPS port)
```

The Apache Unomi actually requires HTTP Basic Auth for access to the Context Server administration REST API, so it is highly recommended that you design your client applications to use the HTTPS port for accessing the REST API.

The user accounts to access the REST API are actually routed through Karaf's JAAS support, which you may find the documentation for here :

- https://karaf.apache.org/manual/latest/#_security_2

You must set an admin password and a health-check password before using the REST API / health endpoints. There are **no** known default passwords shipped with Unomi.

Set the environment variables UNOMI_ROOT_PASSWORD and UNOMI_HEALTHCHECK_PASSWORD, or set org.apache.unomi.security.root.password and org.apache.unomi.healthcheck.password in $MY_KARAF_HOME/etc/unomi.custom.system.properties (or etc/custom.system.properties).

Example:

```
export UNOMI_ROOT_PASSWORD='choose-a-strong-password'
export UNOMI_HEALTHCHECK_PASSWORD='choose-a-strong-health-password'
```

The JAAS admin user name defaults to karaf. Authenticate with karaf:$UNOMI_ROOT_PASSWORD. The health-check user name defaults to health. Authenticate with health:$UNOMI_HEALTHCHECK_PASSWORD.

To change the admin user name, edit $MY_KARAF_HOME/etc/users.properties and also set:

```
karaf.local.user = <your-user>
```

in unomi.custom.system.properties.

For your context servers, and for any standalone Elasticsearch nodes you will need to open the following ports for proper node-to-node communication : 9200 (Elasticsearch REST API), 9300 (Elasticsearch TCP transport)

Of course any ports listed here are the default ports configured in each server, you may adjust them if needed.

Step 2 : Follow industry recommended best practices for securing Elasticsearch

You may find more valuable recommendations here :

- https://www.elastic.co/blog/found-elasticsearch-security
- https://www.elastic.co/blog/scripting-security

Step 4 : Setup a proxy in front of the context server

As an alternative to an application-level firewall, you could also route all traffic to the context server through a proxy, and use it to filter any communication.

5.13. Integrating with an Apache HTTP web server

If you want to setup an Apache HTTP web server in from of Apache Unomi, here is an example configuration using mod_proxy.

In your Unomi package directory, in $MY_KARAF_HOME/etc/unomi.custom.system.properties setup the public address for the hostname unomi.apache.org:

org.apache.unomi.cluster.public.address=https://unomi.apache.org/
org.apache.unomi.cluster.internal.address=http://192.168.1.1:8181

and you will also need to change the cookie domain in the same file:

org.apache.unomi.profile.cookie.domain=apache.org

Main virtual host config:

```
<VirtualHost *:80>
      Include /var/www/vhosts/unomi.apache.org/conf/common.conf
</VirtualHost>

<IfModule mod_ssl.c>
   <VirtualHost *:443>
      Include /var/www/vhosts/unomi.apache.org/conf/common.conf

      SSLEngine on

      SSLCertificateFile    /var/www/vhosts/unomi.apache.org/conf/ssl/24d5b9691e96eafa.crt
      SSLCertificateKeyFile /var/www/vhosts/unomi.apache.org/conf/ssl/apache.org.key
      SSLCertificateChainFile /var/www/vhosts/unomi.apache.org/conf/ssl/gd_bundle-g2-g1.crt

      <FilesMatch "\.(cgi|shtml|phtml|php)$">
            SSLOptions +StdEnvVars
      </FilesMatch>
      <Directory /usr/lib/cgi-bin>
            SSLOptions +StdEnvVars
      </Directory>
      BrowserMatch "MSIE [2-6]" \
            nokeepalive ssl-unclean-shutdown \
            downgrade-1.0 force-response-1.0
      BrowserMatch "MSIE [17-9]" ssl-unclean-shutdown

   </VirtualHost>
</IfModule>
```

common.conf:

```
ServerName unomi.apache.org
ServerAdmin webmaster@apache.org

DocumentRoot /var/www/vhosts/unomi.apache.org/html
CustomLog /var/log/apache2/access-unomi.apache.org.log combined
<Directory />
        Options FollowSymLinks
        AllowOverride None
</Directory>
<Directory /var/www/vhosts/unomi.apache.org/html>
        Options FollowSymLinks MultiViews
        AllowOverride None
        Order allow,deny
        allow from all
</Directory>
<Location /cxs>
    Order deny,allow
    deny from all
    allow from 88.198.26.2
    allow from www.apache.org
</Location>

RewriteEngine On
RewriteCond %{REQUEST_METHOD} ^(TRACE|TRACK)
RewriteRule .* - [F]
ProxyPreserveHost On
ProxyPass /server-status !
ProxyPass /robots.txt !

RewriteCond %{HTTP_USER_AGENT} Googlebot [OR]
RewriteCond %{HTTP_USER_AGENT} msnbot [OR]
RewriteCond %{HTTP_USER_AGENT} Slurp
RewriteRule ^.* - [F,L]

ProxyPass / http://localhost:8181/ connectiontimeout=20 timeout=300 ttl=120
ProxyPassReverse / http://localhost:8181/
```

5.14. Changing the default tracking location

When performing localhost requests to Apache Unomi, a default location will be used to insert values into the session to make the location-based personalization still work. You can modify the default location settings using the centralized configuration file (`$MY_KARAF_HOME/etc/unomi.custom.system.properties`).

Here are the default values for the location settings :

```
# The following settings represent the default position that is used for localhost requests
org.apache.unomi.ip.database.location=${env:UNOMI_IP_DB:-${karaf.etc}/GeoLite2-City.mmdb}
org.apache.unomi.ip.trustedProxies=${env:UNOMI_IP_TRUSTED_PROXIES:-
127.0.0.0/8,::1,10.0.0.0/8,172.16.0.0/12,192.168.0.0/16,169.254.0.0/16,fc00::/7,fe80::/10}
org.apache.unomi.ip.default.countryCode=${env:UNOMI_IP_DEFAULT_COUNTRYCODE:-CH}
org.apache.unomi.ip.default.countryName=${env:UNOMI_IP_DEFAULT_COUNTRYNAME:-
Switzerland}
org.apache.unomi.ip.default.city=${env:UNOMI_IP_DEFAULT_CITY:-Geneva}
org.apache.unomi.ip.default.subdiv1=${env:UNOMI_IP_DEFAULT_SUBDIV1:-2660645}
org.apache.unomi.ip.default.subdiv2=${env:UNOMI_IP_DEFAULT_SUBDIV2:-6458783}
org.apache.unomi.ip.default.isp=${env:UNOMI_IP_DEFAULT_ISP:-Cablecom}
org.apache.unomi.ip.default.latitude=${env:UNOMI_IP_DEFAULT_LATITUDE:-46.1884341}
org.apache.unomi.ip.default.longitude=${env:UNOMI_IP_DEFAULT_LONGITUDE:-6.1282508}
```

You might want to change these for testing or for demonstration purposes.

5.15. Apache Karaf SSH Console

The Apache Karaf SSH console is available inside Apache Unomi, but the port has been changed from the default value of 8101 to 8102 to avoid conflicts with other Karaf-based products. So to connect to the SSH console you should use:

```
ssh -p 8102 karaf@localhost
```

or the user/password you have setup to protect the system if you have changed it. You can find the list of Apache Unomi shell commands in the "Shell commands" section of the documentation.

5.16. Search engine authentication and security

Apache Unomi 4.x talks to **Elasticsearch 9** or **OpenSearch 3** using the settings below. Prefer etc/custom.system.properties (or unomi.custom.system.properties / environment variables). The persistence OSGi configs (etc/org.apache.unomi.persistence.elasticsearch.cfg / …opensearch.cfg) expose the same values as short keys (username, password, sslEnable, sslTrustAllCertificates) that resolve from those system properties.

Elasticsearch security reference: Elasticsearch security. OpenSearch security reference: OpenSearch Security.

5.16.1. User authentication

If the search cluster requires authenticated users, set credentials in custom.system.properties:

```
# Elasticsearch
org.apache.unomi.elasticsearch.username=elastic
org.apache.unomi.elasticsearch.password=${env:ELASTIC_PASSWORD}

# OpenSearch (defaults often use admin + OPENSEARCH_INITIAL_ADMIN_PASSWORD)
org.apache.unomi.opensearch.username=admin
org.apache.unomi.opensearch.password=${env:OPENSEARCH_INITIAL_ADMIN_PASSWORD:-
admin}
```

Equivalent keys in the persistence .cfg files: username= / password=.

5.16.2. SSL communication

By default Unomi may use plain HTTP depending on the backend defaults (org.apache.unomi.elasticsearch.sslEnable defaults to false; org.apache.unomi.opensearch.sslEnable defaults to true in the package samples). To use HTTPS:

```
# Elasticsearch
org.apache.unomi.elasticsearch.sslEnable=true
org.apache.unomi.elasticsearch.sslTrustAllCertificates=true   # development only

# OpenSearch
org.apache.unomi.opensearch.sslEnable=true
org.apache.unomi.opensearch.sslTrustAllCertificates=true      # development only; package default
is true
```

Environment variable equivalents include UNOMI_ELASTICSEARCH_SSL_ENABLE, UNOMI_ELASTICSEARCH_SSL_TRUST_ALL_CERTIFICATES, UNOMI_OPENSEARCH_SSL_ENABLE, and UNOMI_OPENSEARCH_SSL_TRUST_ALL_CERTIFICATES.

Equivalent keys in the persistence .cfg files: sslEnable= / sslTrustAllCertificates=.



sslTrustAllCertificates=true disables certificate validation. Use proper trust stores in production.

5.16.3. Permissions

Apache Unomi requires a particular set of search-engine permissions for its operation.

If you are using Elasticsearch or OpenSearch in a production environment, you will most likely need to fine tune permissions given to the user used by Unomi.

The following privileges are required by Unomi:

- required cluster privileges: manage OR all
- required index privileges on Unomi indices: write, manage, read OR all

5.17. Unomi Distribution (features configuration)

You can define a specific distribution for Apache Unomi to configure desired features when the server boots up. Is it a classic Karaf feature that defines desired features for Unomi. Be aware that, even if a distribution is a classic Karaf feature XML file, it must only be composed feature's dependencies as this file is interpreted by the Unomi ManagementService. No bundle nor config file will be processed from this file.

To set the desired distribution, set the unomi.distribution system property or UNOMI_DISTRIBUTION environment variable to the name of your desired distribution. You can also use the dedicated unomi:setup command to set the distribution interactively.

Apache Unomi comes with some predefined distributions that you can use directly.

5.17.1. Default Distributions

By default, Apache Unomi comes with four predefined distributions:

unomi-distribution-elasticsearch, unomi-distribution-elasticsearch-graphql, unomi-distribution-opensearch, unomi-distribution-opensearch-graphql

Key Differences Between Distributions:

The only difference between the Elasticsearch and OpenSearch configurations is the persistence layer:

- **Elasticsearch**: Uses unomi-elasticsearch-core and unomi-elasticsearch-conditions
- **OpenSearch**: Uses unomi-opensearch-core and unomi-opensearch-conditions

All other features remain identical between both configurations. Each one is derived with or without GraphQL support.

5.17.2. Environment-Specific Distributions

You can create different distributions for different deployment environments by creating a dedicated distribution feature.

Be aware that in distribution's feature, you should only reference other features, not bundles or configuration directly.

Development Environment (includes development tools and debugging features):

```
<features name="unomi-distributions" xmlns="http://karaf.apache.org/xmlns/features/v1.6.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://karaf.apache.org/xmlns/features/v1.6.0
https://karaf.apache.org/xmlns/features/v1.6.0">

  <repository>mvn:org.apache.karaf.features/specs/${karaf.version}/xml/features</repository>
  <repository>mvn:org.apache.cxf.karaf/apache-cxf/${cxf.version}/xml/features</repository>
  <repository>mvn:org.apache.unomi/unomi-kar/${project.version}/xml/features</repository>

  <feature name="unomi-distribution-elasticsearch-dev" description="Apache Unomi ::
Elasticsearch Development Distribution" version="${project.version}">
    <feature dependency="true" version="${project.version}">unomi-base</feature>
    <feature dependency="true" version="${project.version}">unomi-startup</feature>
    <feature dependency="true" version="${project.version}">unomi-elasticsearch-core</feature>
    <feature dependency="true" version="${project.version}">unomi-persistence-core</feature>
    <feature dependency="true" version="${project.version}">unomi-services</feature>
    <feature dependency="true" version="${project.version}">unomi-rest-api</feature>
    <feature dependency="true" version="${project.version}">unomi-cxs-lists-extension</feature>
    <feature dependency="true" version="${project.version}">unomi-cxs-geonames-
extension</feature>
    <feature dependency="true" version="${project.version}">unomi-cxs-privacy-
extension</feature>
    <feature dependency="true" version="${project.version}">unomi-elasticsearch-
conditions</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-base</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-request</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-mail</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-optimization-
test</feature>
    <feature dependency="true" version="${project.version}">unomi-shell-dev-
commands</feature>
    <feature dependency="true" version="${project.version}">unomi-wab</feature>
    <feature dependency="true" version="${project.version}">unomi-web-tracker</feature>
    <feature dependency="true" version="${project.version}">unomi-healthcheck-
elasticsearch</feature>
NOTE: For OpenSearch distributions, use `unomi-healthcheck-opensearch` instead of `unomi-
healthcheck-elasticsearch` in custom feature XML.

    <feature dependency="true" version="${project.version}">unomi-router-karaf-feature</feature>
    <feature dependency="true" version="${project.version}">unomi-groovy-actions</feature>
    <feature dependency="true" version="${project.version}">unomi-rest-ui</feature>
    <feature dependency="true" version="${project.version}">unomi-startup-complete</feature>
  </feature>
</features>
```

Staging Environment (production-like but with some development features):

```
<features name="unomi-distributions" xmlns="http://karaf.apache.org/xmlns/features/v1.6.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://karaf.apache.org/xmlns/features/v1.6.0
https://karaf.apache.org/xmlns/features/v1.6.0">

  <repository>mvn:org.apache.karaf.features/specs/${karaf.version}/xml/features</repository>
  <repository>mvn:org.apache.cxf.karaf/apache-cxf/${cxf.version}/xml/features</repository>
  <repository>mvn:org.apache.unomi/unomi-kar/${project.version}/xml/features</repository>

  <feature name="unomi-distribution-elasticsearch-staging" description="Apache Unomi ::
Elasticsearch Staging Distribution" version="${project.version}">
    <feature dependency="true" version="${project.version}">unomi-base</feature>
    <feature dependency="true" version="${project.version}">unomi-startup</feature>
    <feature dependency="true" version="${project.version}">unomi-elasticsearch-core</feature>
    <feature dependency="true" version="${project.version}">unomi-persistence-core</feature>
    <feature dependency="true" version="${project.version}">unomi-services</feature>
    <feature dependency="true" version="${project.version}">unomi-rest-api</feature>
    <feature dependency="true" version="${project.version}">unomi-cxs-lists-extension</feature>
    <feature dependency="true" version="${project.version}">unomi-cxs-geonames-
extension</feature>
    <feature dependency="true" version="${project.version}">unomi-cxs-privacy-
extension</feature>
    <feature dependency="true" version="${project.version}">unomi-elasticsearch-
conditions</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-base</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-request</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-mail</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-optimization-
test</feature>
    <feature dependency="true" version="${project.version}">unomi-shell-dev-
commands</feature>
    <feature dependency="true" version="${project.version}">unomi-wab</feature>
    <feature dependency="true" version="${project.version}">unomi-web-tracker</feature>
    <feature dependency="true" version="${project.version}">unomi-healthcheck-
elasticsearch</feature>
    <feature dependency="true" version="${project.version}">unomi-router-karaf-feature</feature>
    <feature dependency="true" version="${project.version}">unomi-groovy-actions</feature>
    <feature dependency="true" version="${project.version}">unomi-rest-ui</feature>
    <feature dependency="true" version="${project.version}">unomi-startup-complete</feature>
  </feature>
</features>
```

Production Environment (minimal, secure configuration):

```
<features name="unomi-distributions" xmlns="http://karaf.apache.org/xmlns/features/v1.6.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://karaf.apache.org/xmlns/features/v1.6.0
https://karaf.apache.org/xmlns/features/v1.6.0">

  <repository>mvn:org.apache.karaf.features/specs/${karaf.version}/xml/features</repository>
  <repository>mvn:org.apache.cxf.karaf/apache-cxf/${cxf.version}/xml/features</repository>
  <repository>mvn:org.apache.unomi/unomi-kar/${project.version}/xml/features</repository>

  <feature name="unomi-distribution-elasticsearch-prod" description="Apache Unomi ::
Elasticsearch Production Distribution" version="${project.version}">
    <feature dependency="true" version="${project.version}">unomi-base</feature>
    <feature dependency="true" version="${project.version}">unomi-startup</feature>
    <feature dependency="true" version="${project.version}">unomi-elasticsearch-core</feature>
    <feature dependency="true" version="${project.version}">unomi-persistence-core</feature>
    <feature dependency="true" version="${project.version}">unomi-services</feature>
    <feature dependency="true" version="${project.version}">unomi-rest-api</feature>
    <feature dependency="true" version="${project.version}">unomi-cxs-lists-extension</feature>
    <feature dependency="true" version="${project.version}">unomi-cxs-geonames-
extension</feature>
    <feature dependency="true" version="${project.version}">unomi-cxs-privacy-
extension</feature>
    <feature dependency="true" version="${project.version}">unomi-elasticsearch-
conditions</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-base</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-request</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-mail</feature>
    <feature dependency="true" version="${project.version}">unomi-plugins-optimization-
test</feature>
    <feature dependency="true" version="${project.version}">unomi-wab</feature>
    <feature dependency="true" version="${project.version}">unomi-web-tracker</feature>
    <feature dependency="true" version="${project.version}">unomi-healthcheck-
elasticsearch</feature>
    <feature dependency="true" version="${project.version}">unomi-router-karaf-feature</feature>
    <feature dependency="true" version="${project.version}">unomi-groovy-actions</feature>
    <feature dependency="true" version="${project.version}">unomi-rest-ui</feature>
    <feature dependency="true" version="${project.version}">unomi-startup-complete</feature>
  </feature>
</features>
```

Environment Differences Summary:

Feature	Development	Staging	Production
unomi-plugins-optimization-test	□ (included)	□ (excluded)	□ (excluded)
unomi-shell-dev-commands	□ (included)	□ (included)	□ (excluded)

5.17.3. Distribution Feature Format

Each start features configuration follows the classic Karaf features XML format, but you must only reference other features in your distribution feature. If you reference bundles or configuration files

directly, they will be ignored by Unomi.

To use a custom distribution, the corresponding feature must be accessible by Karaf. It can be done by publishing the feature maven artefact and adding it as a feature repository in Karaf os for any other feature. Of course you can also add your distribution feature in the existing source code and repackage Unomi making it available directly.

5.17.4. Using Custom Distributions

You can use your custom distribution with:

```
# Using system property
-Dunomi.distribution=unomi-distribution-elasticsearch

# Using environment variable
UNOMI_DISTRIBUTION=unomi-distribution-opensearch

# Using the setup command
unomi:setup unomi-distribution-elasticsearch-staging
```

Once you have called the unomi:setup command with your desired distribution, you can start Apache Unomi using the unomi:start command. The setup command only needs to be called once unless you want to change the distribution ; it creates a marker file to remember the chosen distribution. In case you want to change it, you can use the unomi:setup --force command again with the desired distribution name (using the --force option will override the existing distribution).

5.17.5. Auto-Start

You can configure Apache Unomi to automatically start when the server boots up. See the next section for details. This is useful for production deployments where you want the server to start automatically without manual intervention.

To enable auto-start, set the unomi.autoStart system property or UNOMI_AUTO_START environment variable to 'true'

Note: Auto-start only works when Apache Unomi is not already running. If the server is already started, the auto-start setting will be ignored.

5.18. Customizing Apache Unomi Distribution

Apache Unomi allows you to create custom distributions by repackaging the standard distribution with your own configuration files and customizations. This is useful for creating deployment-specific packages that include your custom configurations, plugins, and settings.

5.18.1. Note on Custom Distributions (Advanced)

You can build custom distributions of Unomi using the Karaf Maven Plugin to assemble features and overlay configuration. This is an advanced path and not required for most users. We recommend

Docker-based packaging and configuration overrides as documented above.

For details about custom distributions, see the Karaf documentation (Karaf Maven Plugin):

https://karaf.apache.org/manual/latest/#_karaf_maven_plugin

5.18.2. Important Notes

- **Distribution changes require restart**: After forcing a modification of the distribution, you must restart Apache Unomi for the changes to take effect (be aware that in production it can have unpredictable side effects)
- **Feature dependencies**: When adding custom features, ensure they have all required dependencies and are compatible with your chosen persistence implementation
- **Backup your distribution**: As a distribution is a maven artifact, use the versionning to ensure you can rollback to a previous version.
- **Feature availability**: According to Karaf maven repository access, you can add any feature repository that holds a Unomi distribution's feature and use it into an existing Unomi instance.

5.18.3. Available Features

To see all available features in your Apache Unomi installation, you can use the Karaf shell command:

```
feature:list
```

This will show you all available features that can be included in your distribution.

5.18.4. Docker Deployment with Custom Distribution

For Docker deployments, you can declare a custom distribution feature repository (online) or a local one and mount the corresonding XML file accordingly: -e KARAF_FEATURES_REPOSITORIES=mvn:com.example/myfeatures/1.0.0/xml/features

```
version: '3.8'
services:
  unomi:
    image: apache/unomi:4.0.0
    volumes:
      - ./unomi-custom-distribution-features.xml:/opt/apache-unomi/features/unomi-custom-distribution-features.xml
    environment:
      - KARAF_FEATURES_REPOSITORIES=file:/opt/apache-unomi/features/unomi-custom-distribution-features.xml
      - UNOMI_DISTRIBUTION=unomi-distribution-custom
    depends_on:
      - elasticsearch
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:9.4.3
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
```

Key Points: - Mount your custom distribution feature repository file to /opt/apache-unomi/etc/features/your-features-files.xml (not necessary if the feature file is available in a online maven repository) - Declare the feature repository in Karaf using KARAF_FEATURES_REPOSITORIES environment variable pointing to your mounted file or a online maven artifact (mvn:com.example/custom-unomi-distribs/1.0.0/xml/features) - Use the UNOMI_DISTRIBUTION environment variable to specify which distribution (using the feature name) to use from that feature's repository

5.18.5. Configuration Override Priority

Apache Unomi follows this priority order for configuration files (highest to lowest priority):

1. **Environment Variables**: Override any configuration property
2. **Custom Configuration Files**: Files in etc/ directory override defaults
3. **Default Configuration Files**: Built-in configuration files

Example of Configuration Override:

Default configuration in etc/custom.system.properties:

```
org.apache.unomi.cluster.public.address=${env:UNOMI_PUBLIC_ADDRESS:-http://localhost:8181}
```

Custom override in etc/unomi.custom.system.properties:

```
org.apache.unomi.cluster.public.address=https://unomi.example.com
```

Environment variable override:

```
export UNOMI_PUBLIC_ADDRESS=https://production.unomi.example.com
```

Final Result: The system will use https://production.unomi.example.com (environment variable takes precedence).

5.18.6. Best Practices for Custom Distributions

1. **Version Control**: Keep your custom configuration files in version control
2. **Documentation**: Document all customizations and their purposes
3. **Testing**: Test your custom distribution thoroughly before deployment
4. **Backup**: Always backup the original distribution before customization
5. **Upgrade Path**: Plan how to apply your customizations to future Apache Unomi versions

5.18.7. Deployment with Custom Distribution

Once you have created and published your custom distribution, you can deploy it using the same methods as the standard distribution:

Docker Deployment:

```
version: '3.8'
services:
  unomi:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - KARAF_FEATURES_REPOSITORIES=mvn:com.example/your-custom-unomi-
distributions/1.0.0/xml/features
      - UNOMI_DISTRIBUTION=elasticsearch-prod
    depends_on:
      - elasticsearch
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:9.4.3
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
```

Dockerfile:

```
FROM apache/unomi:4.0.0
COPY apache-unomi-custom-*.tar.gz /tmp/
RUN cd /opt/apache-unomi && tar -xzf /tmp/apache-unomi-custom-*.tar.gz
```

Traditional Deployment:

```
# Extract your custom distribution
tar -xzf apache-unomi-custom-*.tar.gz
cd apache-unomi-custom-*

# Start with your custom configuration
./bin/karaf start
./bin/karaf shell
unomi:start elasticsearch-prod
```

5.19. Health Check Extension

The Health Check extension provides a way to check if required Unomi components are 'live'.

It consists in a simple http endpoint that provide a JSON view of integrated health checks. It can then be used to determine if the server is up and running and can serve requests.

The health check endpoint is available at the following URL: /health/check and returns a simple JSON response that includes all health check provider responses.

Basic Http Authentication enforce security for the health check endpoint using the existing karaf realm. The user needs to have the specific role **health** to access the endpoint. Users and roles can be configured in the etc/users.properties file. The shipped health user has no default password: its password comes from UNOMI_HEALTHCHECK_PASSWORD, which must be set before starting (see REST API security). An empty password is never accepted, on this endpoint or any other.

Specific configuration is located in : org.apache.unomi.healthcheck.cfg Existing health checks are using configuration from that file, including authentication realm.

Existing health checks give information about:

- Karaf (as soon as the karaf container is started, that check is LIVE)
- Elasticsearch **or** OpenSearch (connection to the search cluster and its health) — provider name elasticsearch or opensearch depending on the distribution
- Unomi (unomi bundles status)
- Persistence (Unomi binding to the search/persistence backend)
- Cluster health (unomi cluster status and nodes information)

Set UNOMI_HEALTHCHECK_PROVIDERS (or healthcheck.providers in the cfg file) to the comma-separated list that matches your deployment, for example cluster,elasticsearch,unomi,persistence or cluster,opensearch,unomi,persistence. The karaf probe is always reported and does not need to be listed.

All healthcheck can have a status : - DOWN (service is not available) - UP (service is running or starting) - LIVE (service is ready to serve requests) - ERROR (an error occurred during service health check)

Any subsystem health check uses the configured timeout (default 400 ms); a check that does not answer

in time is cancelled and returned as an error.

Typical response to /health/check when Unomi is not started yet (Elasticsearch distribution). Items are sorted by name (same as Unomi 3.0). Empty data is omitted; search backends may add cluster details when the probe answers:


```
[
 {
  "name":"cluster",
  "status":"DOWN",
  "collectingTime":0
 },
 {
  "name":"elasticsearch",
  "status":"LIVE",
  "collectingTime":6,
  "data":{
   "cluster_name":"elasticsearch",
   "status":"green",
   "number_of_nodes":1
  }
 },
 {
  "name":"karaf",
  "status":"LIVE",
  "collectingTime":0
 },
 {
  "name":"persistence",
  "status":"DOWN",
  "collectingTime":0
 },
 {
  "name":"unomi",
  "status":"DOWN",
  "collectingTime":0
 }
]
```


Each object has name, status (DOWN | UP | LIVE | ERROR), collectingTime (milliseconds), and optional data. HTTP 200 only when every status is LIVE; otherwise 206.

Existing health check can be extended by adding specific provider in the extension. A provider is a class that implements the HealthCheckProvider interface.

```
package org.apache.unomi.healthcheck;

public interface HealthCheckProvider {
   String name();
   HealthCheckResponse execute();
}
```

Calls to provider are supposed to be done at a regular rate (every 15 seconds for example) and should be fast to execute. Feel free to include any caching strategy if needed.

5.19.1. Health check for another persistence backend

A persistence backend that is not packaged with Unomi (PostgreSQL, for example) integrates with the health check the same way as Elasticsearch and OpenSearch:

1. Register a HealthCheckProvider OSGi service whose name() is the backend id (for example postgresql). Import org.apache.unomi.healthcheck from the healthcheck bundle. The generic persistence provider already works with any PersistenceService.
2. Ship a Karaf feature (for example unomi-healthcheck-postgresql) with the healthcheck bundle and a configfile for /etc/org.apache.unomi.healthcheck.cfg. Only the provider list is required; healthcheck.enabled, healthcheck.timeout and authentication.realm default to true, 400 and karaf. Keep the placeholders so the UNOMI_HEALTHCHECK_* variables still apply:

```
healthcheck.enabled = ${org.apache.unomi.healthcheck.enabled:-true}
healthcheck.providers = ${org.apache.unomi.healthcheck.providers:-
cluster,postgresql,unomi,persistence}
healthcheck.timeout = ${org.apache.unomi.healthcheck.timeout:-400}
authentication.realm = ${org.apache.unomi.security.realm:-karaf}
```

3. Use that feature instead of unomi-healthcheck-elasticsearch or unomi-healthcheck-opensearch in the backend's distribution feature.

The Elasticsearch and OpenSearch providers stay inactive when they are not listed, so they do not report errors on other backends.

5.19.2. Configuration

Healthcheck extension configuration is located in the file etc/org.apache.unomi.healthcheck.cfg

Extension can be enabled by setting the property healthcheck.enabled to true (default). Environment variable: UNOMI_HEALTHCHECK_ENABLED (maps to org.apache.unomi.healthcheck.enabled). Any other value than true (for example false) unregisters /health/check. It never makes the endpoint answer without authentication.

By default the cfg file lists the providers for the active distribution (cluster,elasticsearch,unomi,persistence or cluster,opensearch,unomi,persistence). Override with healthcheck.providers or UNOMI_HEALTHCHECK_PROVIDERS (maps to org.apache.unomi.healthcheck.providers). Names must match the name() of a registered provider, including custom ones. A name that no installed bundle provides is skipped with a warning. The karaf probe is always reported as LIVE and does not need to be listed.

The per-provider timeout is healthcheck.timeout in milliseconds (default 400, allowed range 1–60000). Environment variable: UNOMI_HEALTHCHECK_TIMEOUT (maps to org.apache.unomi.healthcheck.timeout). Out-of-range or invalid values fall back to 400 with a warning.

None of these settings can turn authentication off; a blank health-check password is still refused.

Chapter 6. Multi-tenancy

6.1. Multi-tenancy guide

Apache Unomi 4.0+ isolates profiles, events, segments, rules, and schemas per tenant. Each tenant has public and private API keys. See also Migrate from 3.0 to 4.0 for authentication changes.

6.1.1. Overview

Apache Unomi provides robust multi-tenancy support, allowing multiple organizations to use the same Unomi instance while maintaining complete data isolation. Each tenant gets their own dedicated space with separate data storage, configuration, and API keys.

6.1.2. Key Features

- Complete data isolation between tenants
- Dual API key system (public and private keys)
- Tenant-specific configuration
- Per-tenant usage metrics (quotas enforced upstream)
- Migration tools for existing data
- Support for REST APIs (and GraphQL when enabled)

6.1.3. Authentication Methods

Public API Key

Used for public endpoints (e.g., context requests) that are typically accessed from client-side applications.

```
X-Unomi-Api-Key: <PUBLIC_KEY>
```

Private API Key

Used for administrative operations and sensitive endpoints. Requires Basic Authentication using tenant ID and private key.

```
--user "TENANT_ID:PRIVATE_KEY"
```



curl automatically handles Base64 encoding when using the --user option, so you don't need to manually encode the credentials.

6.1.4. Getting Started

Creating Your First Tenant

To create a new tenant, use the Tenant API endpoint:

```
curl -X POST http://localhost:8181/cxs/tenants \
  --user "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Content-Type: application/json" \
  -d '{
    "requestedId": "my-tenant",
    "properties": {
      "name": "My Organization",
      "description": "My organization description",
      "publicCorsAllowedOrigins": ["https://www.my-organization.com"]
    }
  }'
```

publicCorsAllowedOrigins is optional: it lists the websites that may use the visitor cookie with this tenant's key (see Websites listed by each tenant). An invalid entry is refused with 400.

Protected event types (sessionCreated, sessionReassigned, profileUpdated, ruleFired, updateProperties, plus any extra types listed in the tenant's restrictedEventTypes) are accepted from a public API key caller only when the caller IP is in the tenant's authorizedIPs (addresses or CIDR ranges). An empty authorizedIPs list, the default, refuses them from every public caller; update the tenant to set it. The GraphQL API passes no caller IP, so public GraphQL callers can never send protected events. Callers using the tenant private key or administrator credentials are not IP-checked.

The create response includes the tenant with **masked** API key metadata only (plaintext secrets are not returned and cannot be recovered from storage):

```
{
 "itemId": "my-tenant",
 "itemType": "tenant",
 "status": "ACTIVE",
 "apiKeys": [
  {
   "maskedKey": "unomi_v1_****ab12",
   "keyType": "PUBLIC",
   "revoked": false
  },
  {
   "maskedKey": "unomi_v1_****cd34",
   "keyType": "PRIVATE",
   "revoked": false
  }
 ],
 "properties": {
  "name": "My Organization",
  "description": "My organization description"
 }
}
```



Store plaintext keys immediately after calling the key creation endpoint. Creating a tenant generates keys server-side, but only masked values appear on the tenant resource. Regenerate (replace) keys to obtain plainTextKey:

```
curl -X POST "http://localhost:8181/cxs/tenants/my-tenant/apikeys?type=PUBLIC" \
 --user "karaf:$UNOMI_ROOT_PASSWORD"

curl -X POST "http://localhost:8181/cxs/tenants/my-tenant/apikeys?type=PRIVATE" \
 --user "karaf:$UNOMI_ROOT_PASSWORD"
```

Example response:

```
{
 "apiKey": {
  "maskedKey": "unomi_v1_****ab12",
  "keyType": "PUBLIC",
  "revoked": false
 },
 "plainTextKey":
"unomi_v1_0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF"
}
```

Making Your First API Call

Public Endpoint Example (Context Request)

```
curl http://localhost:8181/cxs/context.json \
 -H "X-Unomi-Api-Key: <PUBLIC_KEY>" \
 -H "Content-Type: application/json" \
 -d '{
  "sessionId": "session-123",
  "source": {
   "itemId": "homepage",
   "itemType": "page",
   "scope": "example"
  }
 }'
```

Private Endpoint Example (Profile Management)

```
curl -X GET http://localhost:8181/cxs/profiles \
 --user "TENANT_ID:PRIVATE_KEY" \
 -H "Content-Type: application/json"
```

6.1.5. Configuration

Basic Setup

Configure the default tenant in etc/org.apache.unomi.rest.authentication.cfg (or org.apache.unomi.rest.authentication.defaultTenantId / UNOMI_REST_AUTHENTICATION_DEFAULTTENANTID):

```
# Tenant for JAAS admin requests that omit X-Unomi-Tenant-Id (UNOMI-1008).
# Defaults to "default"; set it to the tenant id chosen during the 3.0→4.0 migration.
default.tenant.id=default
```

In 4.0 this setting only applies to JAAS admin requests. Single-tenant compatibility mode does not use it yet: its REST requests always run on tenant default, so keep the setting at default while that mode is on, or GraphQL admin queries will work on a different tenant than REST.

Other tenant-related settings live in etc/org.apache.unomi.tenant.cfg:

```
# API key validity period
tenant.apikey.validity.period=30
tenant.apikey.validity.unit=DAYS

# Warn this many days before an API key expires
tenant.apikey.rotation.warning.days=7

# Prefix for tenant-specific security roles
tenant.security.roles.prefix=unomi_tenant_
```

Security Provider Configuration

For Elasticsearch:

```
tenant.security.provider=elasticsearch
```

For OpenSearch:

```
tenant.security.provider=opensearch
```

6.1.6. Security Model

Roles and Permissions

Apache Unomi implements a hierarchical role-based access control (RBAC) system. The main roles are:

- ROLE_UNOMI_SYSTEM: Highest privilege level, used for system operations
- ROLE_UNOMI_ADMIN: Administrative access across the platform
- ROLE_UNOMI_TENANT_USER: Basic tenant access for public operations
- ROLE_UNOMI_TENANT_ADMIN: Extended tenant access for private operations
- ROLE_UNOMI_TENANT_PUBLIC_PREFIX_*: Tenant-specific public roles
- ROLE_UNOMI_TENANT_PRIVATE_PREFIX_*: Tenant-specific private roles

Configure system roles in etc/org.apache.unomi.security.cfg:

```
# Define system roles
systemRoles=ROLE_UNOMI_SYSTEM,ROLE_UNOMI_ADMIN

# Enable encryption for sensitive data
enableEncryption=false

# Operation role mappings
operation.roles.QUERY=ROLE_UNOMI_TENANT_USER,ROLE_UNOMI_TENANT_ADMIN
operation.roles.PROFILE_UPDATE=ROLE_UNOMI_TENANT_ADMIN
operation.roles.SYSTEM_MAINTENANCE=ROLE_UNOMI_SYSTEM
operation.roles.TENANT_MANAGEMENT=ROLE_UNOMI_ADMIN
operation.roles.DECRYPT_PROFILE_DATA=ROLE_UNOMI_TENANT_ADMIN
operation.roles.SEGMENT_UPDATE=ROLE_UNOMI_TENANT_ADMIN
operation.roles.RULE_UPDATE=ROLE_UNOMI_TENANT_ADMIN
```

The configuration uses the format operation.roles.OPERATION_NAME=ROLE1,ROLE2,... where: - OPERATION_NAME is the uppercase operation identifier - Multiple roles are comma-separated - Changes take effect immediately without restart

Operation Configuration

Operations in Unomi can be customized to require specific roles. This is configured through OSGi configuration files.

Configuration File

Create or modify the file etc/org.apache.unomi.security.cfg:

```
# Define system roles
systemRoles=ROLE_UNOMI_SYSTEM,ROLE_UNOMI_ADMIN

# Enable encryption for sensitive data
enableEncryption=false

# Operation role mappings
operation.roles.QUERY=ROLE_UNOMI_TENANT_USER,ROLE_UNOMI_TENANT_ADMIN
operation.roles.PROFILE_UPDATE=ROLE_UNOMI_TENANT_ADMIN
operation.roles.SYSTEM_MAINTENANCE=ROLE_UNOMI_SYSTEM
operation.roles.TENANT_MANAGEMENT=ROLE_UNOMI_ADMIN
operation.roles.DECRYPT_PROFILE_DATA=ROLE_UNOMI_TENANT_ADMIN
operation.roles.SEGMENT_UPDATE=ROLE_UNOMI_TENANT_ADMIN
operation.roles.RULE_UPDATE=ROLE_UNOMI_TENANT_ADMIN
```

The configuration uses the format operation.roles.OPERATION_NAME=ROLE1,ROLE2,... where: - OPERATION_NAME is the uppercase operation identifier - Multiple roles are comma-separated - Changes take effect immediately without restart

Common Operations

Here are some common operations and their typical role requirements:

Operation	Description	Default Required Roles
QUERY	Basic data querying	ROLE_UNOMI_TENANT_USER, ROLE_UNOMI_TENANT_ADMIN
PROFILE_UPDATE	Update profile data	ROLE_UNOMI_TENANT_ADMIN
SYSTEM_MAINTENANCE	System-level operations	ROLE_UNOMI_SYSTEM
TENANT_MANAGEMENT	Tenant administration	ROLE_UNOMI_ADMIN
DECRYPT_PROFILE_DATA	Access to encrypted profile data	ROLE_UNOMI_TENANT_ADMIN
SEGMENT_UPDATE	Update user segments	ROLE_UNOMI_TENANT_ADMIN
RULE_UPDATE	Update business rules	ROLE_UNOMI_TENANT_ADMIN

Custom Operations

To define custom operations:

1. Define the operation name (use uppercase by convention)
2. Add the operation-role mapping to the configuration file
3. Use securityService.validateTenantOperation() to enforce the permission

Example:

1. Add to etc/org.apache.unomi.security.cfg:

```
operation.roles.CUSTOM_OPERATION=ROLE_UNOMI_TENANT_ADMIN
```

2. Use in your code:

```
public void performCustomOperation() {
    securityService.validateTenantOperation("CUSTOM_OPERATION");
    // Operation implementation
}
```

Subjects and Authentication

A Subject represents an authenticated entity in the system. There are three types of subjects:

1. System Subject:
    - Used for system-level operations
    - Has full access across all tenants
    - Created with ROLE_UNOMI_SYSTEM
2. Admin Subject:
    - Used for administrative operations
    - Has tenant management capabilities
    - Created with ROLE_UNOMI_ADMIN
3. Tenant Subject:
    - Represents a tenant-specific user
    - Has access only to their tenant's resources
    - Created with tenant-specific roles

Example of subject creation:

```
Subject tenantSubject = new Subject();
tenantSubject.getPrincipals().add(new UserPrincipal("tenant-id"));
tenantSubject.getPrincipals().add(new RolePrincipal("ROLE_UNOMI_TENANT_ADMIN"));
```

Tenant-Role Relationship

Each tenant has associated public and private roles:

1. User Role (ROLE_UNOMI_TENANT_USER):
    - Basic read-only access to tenant data
    - Can perform queries and view profiles
2. Admin Role (ROLE_UNOMI_TENANT_ADMIN):
    - Full access to tenant data
    - Can perform all tenant operations

Operation Validation

The security service validates operations based on:

1. Subject's roles
2. Operation type
3. Tenant context

Example of operation validation:

```
// Validate a tenant operation
securityService.validateTenantOperation("SYSTEM_MAINTENANCE");

// Execute with elevated privileges
securityService.executeAsSystemSubject(() -> {
    // Perform system operation
});
```

Best Practices

1. Role Assignment:
    - Assign minimum required roles
    - Use tenant-specific roles when possible
    - Avoid using system roles for regular operations
2. Subject Management:
    - Clear subjects after operations
    - Use temporary privileged subjects sparingly
    - Always validate tenant context
3. Security Configuration:

- Regularly rotate API keys
- Enable encryption for sensitive data
- Monitor failed authentication attempts

4. Operation Execution:
   - Use executeAsSystemSubject for system operations
   - Validate operations before execution
   - Maintain proper audit trails

6.1.7. Tenant Management

Listing Tenants

```
curl -X GET http://localhost:8181/cxs/tenants \
  --user "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Content-Type: application/json"
```

Updating a Tenant

PUT replaces the tenant's settings with the body, which must carry the tenant's itemId. Read the tenant first, change what you need, and send it back. API keys and the creation date are never changed by PUT: keys are managed with the API key endpoints below, and those fields in the body are ignored. If the tenant changes while the update is being applied, for example because one of its keys is regenerated, the update answers 409 rather than undoing that change; send it again.

```
curl -s --user "karaf:$UNOMI_ROOT_PASSWORD" http://localhost:8181/cxs/tenants/my-tenant \
  | jq '.properties.name = "Updated Organization Name"' \
  | curl -X PUT http://localhost:8181/cxs/tenants/my-tenant \
    --user "karaf:$UNOMI_ROOT_PASSWORD" \
    -H "Content-Type: application/json" --data-binary @-
```

The websites that may use the visitor cookie with the tenant's key are listed in its publicCorsAllowedOrigins property; see Websites listed by each tenant.

Regenerating API Keys

Regenerating replaces any existing key of the same type. Store plainTextKey from the response immediately.

```
curl -X POST "http://localhost:8181/cxs/tenants/my-tenant/apikeys?type=PUBLIC&validityDays=30" \
  --user "karaf:$UNOMI_ROOT_PASSWORD"

curl -X POST "http://localhost:8181/cxs/tenants/my-tenant/apikeys?type=PRIVATE" \
  --user "karaf:$UNOMI_ROOT_PASSWORD"
```



type must be PUBLIC or PRIVATE. Optional validityDays sets expiration; omit it (or use 0) for no expiration.

When a key change takes effect:

- A new key works at once on the node that generated it. Other nodes accept it on its first request, or at the latest after the miss refresh interval below (5 seconds by default).
- The key it replaces is refused at once on every node, as are revoked and expired keys and the keys of a deleted tenant.
- While a node that has just started reads its tenants, it refuses API keys with 401 instead of making requests wait. Clients should retry.

Two settings control how other nodes learn about new keys:

Property	Environment variable	Default	Meaning
org.apache.unomi.tenant.apikey.refresh.interval	UNOMI_TENANT_APIKEY_REFRESH_INTERVAL	30000	How often, in milliseconds, each node reloads its list of API keys.
org.apache.unomi.tenant.apikey.miss.refresh.interval	UNOMI_TENANT_APIKEY_MISS_REFRESH_INTERVAL	5000	Minimum time, in milliseconds, between reloads triggered by unknown keys. Lower values let a key created on another node work sooner, but each reload reads every tenant, so this also limits the load that requests with wrong keys can cause.

Set them in etc/custom.system.properties or through the environment variables; they must be positive and are read at startup. They never delay refusing a revoked, replaced or expired key.

Deleting a Tenant

```
curl -X DELETE http://localhost:8181/cxs/tenants/my-tenant \
  --user "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Content-Type: application/json"
```

6.1.8. GraphQL Support

GraphQL endpoints support both public and private authentication methods:

```
curl -X POST http://localhost:8181/graphql \
  --user "TENANT_ID:PRIVATE_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "query": "{ profiles { edges { node { id } } } }"
  }'
```

Tenant-Specific GraphQL Schemas

The GraphQL API provides tenant-specific schemas, meaning each tenant can have a unique GraphQL schema based on their property types and configurations. This ensures that tenants only see the data and fields relevant to their specific implementation.

When a tenant accesses the GraphQL API:

1. The system automatically detects the tenant from the authentication context
2. It retrieves (or creates) a GraphQL schema specific to that tenant
3. The schema only includes property types defined for that tenant
4. Changes to a tenant's property types are automatically reflected in their schema

For complete details on the GraphQL multi-tenancy implementation, refer to the GraphQL API section of the documentation.

6.1.9. Monitoring and Management

Monitoring API Usage

Unomi exposes read-only usage metrics per tenant. Quota enforcement belongs in your upstream gateway or control plane.

```
curl -X GET "http://localhost:8181/cxs/tenants/my-tenant/usage?period=current-month" \
  --user "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Content-Type: application/json"
```

Supported period values are current-month (default), YYYY-MM (for example 2026-07), and legacy 24h (same as the current calendar month).

The response includes profile, scope, segment, and rule totals, per-scope segment/rule breakdown (scopeUsages), monthly eventCount for the requested period (periodStart / periodEnd in epoch millis), active API key count, storage document count, in-memory REST request count since process start, and collectedAt (epoch millis). Values refresh on a background schedule and may be stale until the next collection cycle.

Event retention purge

Upstream control planes can delete old tenant events through Unomi instead of talking to the search cluster directly:

```
curl -X POST "http://localhost:8181/cxs/tenants/my-tenant/purge/events?retentionDays=90" \
  --user "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Content-Type: application/json"
```

The response reports how many events matched the retention cutoff before deletion ran (eventsMatched) and whether the delete-by-query completed successfully (purgeRequested); a purgeRequested of false means the deletion failed and the request returns HTTP 500 (see server logs for the cause). The minimum accepted retention is seven days, to guard against accidentally purging recent or active event data.

Data Migration

There is no REST endpoint to copy data between tenants. Version upgrades that introduce or adjust multi-tenancy run through the Karaf shell with Unomi stopped:

```
unomi:migrate 3.0.0
```

See Migrate from 3.0 to 4.0 and Shell commands.

6.1.10. Best Practices

API Key Management

- Rotate keys regularly using the key regeneration endpoint
- Use public keys only for public endpoints
- Never expose private keys in client-side code
- Monitor API key usage and implement rate limiting

Resource Management

- Set appropriate quotas for each tenant
- Monitor resource usage through the tenant usage API (GET /cxs/tenants/{tenantId}/usage)
- Configure alerts for quota limits
- Regularly review and adjust limits based on usage patterns

Security

- Always use HTTPS in production

- Implement proper key rotation policies
- Conduct regular security audits
- Monitor for suspicious activity patterns
- Keep tenant configurations up to date

6.1.11. Troubleshooting

Common Issues

401 Unauthorized

- Verify API key is correct
- Check if using public key for private endpoint
- Ensure tenant ID matches the API key
- If the key was just generated on another node, or this node has just started, retry after a few seconds

400 Bad Request

- Check if API key header is present
- Verify request format is correct

404 Not Found

- Verify tenant ID exists
- Check if endpoint path is correct

Logging

Enable debug logging for tenant-related operations:

```
log4j.logger.org.apache.unomi.tenant=DEBUG
```

6.1.12. Migration Guide

Migrating Existing Data

To attach existing 3.0 data to the multi-tenant model when upgrading to 4.0:

1. Stop Apache Unomi.
2. Run the bundled migration from the Karaf shell:

```
unomi:migrate 3.0.0
```

This runs the migrate-4.0.0-* Groovy scripts (tenant document IDs, system item IDs, default tenant initialization, legacy queryBuilder IDs, and related fixes). See Migrate from 3.0 to 4.0.

3. Start Unomi 4.0 and regenerate tenant API keys if you need the plaintext secrets (create/migrate responses only expose masked keys). See Creating Your First Tenant / API key regeneration below.

Verification

After migration, verify tenant context and data access:

```
# List tenants (JAAS admin)
curl -X GET http://localhost:8181/cxs/tenants \
  --user "karaf:$UNOMI_ROOT_PASSWORD" \
  -H "Accept: application/json"

# List profiles for a tenant (tenant private key auth)
curl -X GET http://localhost:8181/cxs/profiles \
  --user "TENANT_ID:PRIVATE_KEY" \
  -H "Accept: application/json"
```

6.1.13. Working with Events and Rules

Creating Custom Event Types

First, create a JSON schema for your custom event type and deploy it using the JSON schema endpoint:

```
curl -X POST http://localhost:8181/cxs/jsonSchema \
 --user "TENANT_ID:PRIVATE_KEY" \
 -H "Content-Type: application/json" \
--data-raw '{
   "$id": "https://unomi.apache.org/schemas/json/events/purchaseCompleted/1-0-0",
   "$schema": "https://json-schema.org/draft/2019-09/schema",
   "self": {
      "vendor": "org.apache.unomi",
      "name": "purchaseCompleted",
      "format": "jsonschema",
      "target": "events",
      "version": "1-0-0"
   },
   "title": "Purchase Completed Event",
   "type": "object",
   "allOf": [{ "$ref": "https://unomi.apache.org/schemas/json/event/1-0-0" }],
   "properties": {
      "properties": {
         "type": "object",
         "properties": {
            "orderId": {
               "type": "string",
               "description": "The unique order identifier"
            },
            "amount": {
               "type": "number",
               "description": "The total purchase amount"
            },
            "currency": {
               "type": "string",
               "description": "The currency code (e.g., USD)"
            },
            "items": {
               "type": "array",
               "items": {
                  "type": "object",
                  "properties": {
                     "productId": {
                        "type": "string"
                     },
                     "quantity": {
                        "type": "integer"
                     },
                     "price": {
                        "type": "number"
                     }
                  },
                  "required": ["productId", "quantity", "price"]
               }
            }
         },
         "required": ["orderId", "amount", "currency"]
      }
   },
   "unevaluatedProperties": false
}'
```

You can verify your schema has been deployed by listing all available schemas:

```
curl -X GET http://localhost:8181/cxs/jsonSchema \
  --user "TENANT_ID:PRIVATE_KEY" \
  -H "Content-Type: application/json"
```



Replace TENANT_ID and PRIVATE_KEY with your actual tenant ID and private API key. Only the Tenant API (/cxs/tenants) uses system administrator authentication (the karaf user and your configured UNOMI_ROOT_PASSWORD).

You can also validate events against your schema using the validation endpoint:

```
curl -X POST http://localhost:8181/cxs/jsonSchema/validateEvent \
  --user "TENANT_ID:PRIVATE_KEY" \
  -H "Content-Type: application/json" \
  --data '{
    "eventType": "purchaseCompleted",
    "scope": "myapp",
    "properties": {
      "orderId": "order-123",
      "amount": 99.99,
      "currency": "USD",
      "items": [
        {
          "productId": "product-001",
          "quantity": 2,
          "price": 49.99
        }
      ]
    }
  }'
```

Sending Custom Events

Once the event type is defined, you can send events:



/cxs/context.json is a public endpoint, so the profile is carried by the context-profile-id cookie: since Unomi 4.0 a public caller's body profileId is ignored. Only a trusted caller (tenant private key or system administrator) may bind a profile explicitly through the body. See client-facing hardening.

```
curl -X POST http://localhost:8181/cxs/context.json \
  -H "X-Unomi-Api-Key: <PUBLIC_KEY>" \
  -H "Content-Type: application/json" \
  -b "context-profile-id=profile-456" \
  -d '{
    "sessionId": "session-123",
    "source": {
      "itemId": "checkout-page",
      "itemType": "page",
      "scope": "myapp"
    },
    "events": [{
      "eventType": "purchaseCompleted",
      "scope": "myapp",
      "properties": {
        "orderId": "order-789",
        "amount": 99.99,
        "currency": "USD",
        "items": [
          {
            "productId": "product-001",
            "quantity": 2,
            "price": 49.99
          }
        ]
      }
    }]
  }'
```

Creating Rules for Event Processing

Create a rule to update profile properties based on purchase events:

```
curl -X POST http://localhost:8181/cxs/rules \
 --user "TENANT_ID:PRIVATE_KEY" \
 -H "Content-Type: application/json" \
 -d '{
  "metadata": {
   "id": "updateTotalPurchases",
   "name": "Update total purchases",
   "description": "Updates profile properties when a purchase is completed",
   "scope": "myapp"
  },
  "condition": {
   "type": "eventTypeCondition",
   "parameterValues": {
    "eventTypeId": "purchaseCompleted"
   }
  },
  "actions": [
   {
    "type": "setPropertyAction",
    "parameterValues": {
     "setPropertyName": "properties.totalPurchases",
     "setPropertyValue": "script::profile.properties.totalPurchases != null ?
profile.properties.totalPurchases + 1 : 1",
     "setPropertyStrategy": "alwaysSet"
    }
   },
   {
    "type": "setPropertyAction",
    "parameterValues": {
     "setPropertyName": "properties.totalRevenue",
     "setPropertyValue": "script::profile.properties.totalRevenue != null ?
profile.properties.totalRevenue + event.properties.amount : event.properties.amount",
     "setPropertyStrategy": "alwaysSet"
    }
   }
  ]
 }'
```

Testing the Event Processing

To test that everything works:

1. Send a purchase event:

```
curl -X POST http://localhost:8181/cxs/context.json \
  -H "X-Unomi-Api-Key: <PUBLIC_KEY>" \
  -H "Content-Type: application/json" \
  -b "context-profile-id=profile-456" \
  -d '{
    "sessionId": "session-123",
    "source": {
      "itemId": "checkout-page",
      "itemType": "page",
      "scope": "myapp"
    },
    "events": [{
      "eventType": "purchaseCompleted",
      "scope": "myapp",
      "properties": {
        "orderId": "order-790",
        "amount": 149.99,
        "currency": "USD",
        "items": [
          {
            "productId": "product-002",
            "quantity": 1,
            "price": 149.99
          }
        ]
      }
    }]
  }'
```

+ 2. Verify profile properties were updated:

```
curl -X GET http://localhost:8181/cxs/profiles/profile-456 \
  --user "TENANT_ID:PRIVATE_KEY" \
  -H "Content-Type: application/json"
```

Expected response will show updated properties:

```
{
  "itemId": "profile-456",
  "properties": {
    "totalPurchases": 1,
    "totalRevenue": 149.99
  }
  // ... other profile properties ...
}
```

Advanced Rule Examples

Segmenting High-Value Customers

Create a segment for customers with high total revenue:

```
curl -X POST http://localhost:8181/cxs/segments \
  --user "TENANT_ID:PRIVATE_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "metadata": {
      "id": "highValueCustomers",
      "name": "High Value Customers",
      "scope": "myapp"
    },
    "condition": {
      "type": "profilePropertyCondition",
      "parameterValues": {
        "propertyName": "properties.totalRevenue",
        "comparisonOperator": "greaterThan",
        "propertyValueInteger": 1000
      }
    }
  }'
```

Tracking Purchase Frequency

Create a rule to track days between purchases:

```
curl -X POST http://localhost:8181/cxs/rules \
 --user "TENANT_ID:PRIVATE_KEY" \
 -H "Content-Type: application/json" \
 -d '{
   "metadata": {
    "id": "trackPurchaseFrequency",
    "name": "Track Purchase Frequency",
    "scope": "myapp"
   },
   "condition": {
    "type": "eventTypeCondition",
    "parameterValues": {
     "eventTypeId": "purchaseCompleted"
    }
   },
   "actions": [
    {
     "type": "setPropertyAction",
     "parameterValues": {
      "setPropertyName": "properties.lastPurchaseDate",
      "setPropertyValue": "script::currentDate",
      "setPropertyStrategy": "alwaysSet"
     }
    },
    {
     "type": "setPropertyAction",
     "parameterValues": {
      "setPropertyName": "properties.daysBetweenPurchases",
      "setPropertyValue": "script::profile.properties.lastPurchaseDate != null ?
Duration.between(profile.properties.lastPurchaseDate.toInstant(), currentDate.toInstant()).toDays()
: null",
      "setPropertyStrategy": "alwaysSet"
     }
    }
   ]
 }'
```

Chapter 7. JSON schemas

7.1. Introduction

Introduced with Apache Unomi 2.0, JSON Schema is used to validate data submitted through public API endpoints (for example /cxs/context.json). In Unomi 4.0 those endpoints require a tenant public API key (X-Unomi-Api-Key); they are no longer anonymous.

7.1.1. What is a JSON Schema

JSON Schema is a powerful standard for validating the structure of JSON data. Described as a JSON object, a JSON schema file contains format, types, patterns, and more. Used against JSON data, a JSON schema validates that the data is compatible with the specified schema.

Example of a basic JSON schema that validates that the path property is a string property:

```
{
  "$id":"https://unomi.apache.org/schemas/json/example/1-0-0",
  "$schema":"https://json-schema.org/draft/2019-09/schema",
  "title":"Example of a basic schema",
  "type":"object",
  "properties":{
    "path":{
      "type":"string",
      "$comment":"Example of a property."
    }
  }
}
```

```
{
  "path": "example/of/path" //Is valid
}
```

```
{
  "path": 100  // Is not valid
}
```

Apache Unomi is using json-schema-validator to integrate JSON schema. The library and its source code is available at: https://github.com/networknt/json-schema-validator, you can refer to the feature's pom.xml available at json-schema/service/pom.xml to identify which version of the library is currently integrated.

You can discover and play with JSON schema using online tools such as JSON Schema Validator. Such tools allow you to validate a schema against JSON data (such as the example above), and can point to particular errors. More details about JSON schema are available on the official specification's website: https://json-schema.org/specification.html

7.1.2. Key concepts

This section details concepts that are important to understand in order to use JSON schema validation with Apache Unomi.

$id keyword

The **$id** keyword:

Each schema in Apache Unomi should have a **$id**, the **$id** value is an URI which will be used to retrieve the schema and must be unique.

Example:

```
{
  "$id":"https://unomi.apache.org/schemas/json/example/1-0-0"
}
```

$ref keyword

The **$ref** keyword allows you to reference another JSON schema by its **$id** keyword. It's possible to separate complex structures or repetitive parts of schema into other small files and use **$ref** to include them into several json schemas.

Example with a person and an address:

```
{
  "$id": "https://example.com/schemas/address",
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" }
  }
}
```

```
{
  "type": "object",
  "properties": {
    "first_name":{ "type": "string" },
    "last_name": { "type": "string" },
    "shipping_address": {
      "$ref": "https://example.com/schemas/address"
    },
    "billing_address": {
      "$ref": "https://example.com/schemas/address"
    }
  }
}
```

More details about **$ref** can be found in the specifications: https://json-schema.org/understanding-json-schema/structuring.html#ref

allOf keyword

The allOf keyword is an array of fields which allows schema composition. The data will be valid against a schema if the data are valid against all of the given subschemas in the allOf part and are valid against the properties defined in the schema.

```
{
  "$id": "https://unomi.apache.org/schemas/json/example/1-0-0",
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "type": "object",
  "allOf": [
    {
      "type": "object",
      "properties": {
        "fromAllOf": {
          "type": "integer",
          "$comment": "Example of allOf."
        }
      }
    }
  ],
  "properties": {
    "myProperty": {
      "type": "string",
      "$comment": "Example of a property."
    }
  }
}
```

Valid JSON:

```
{
  "myProperty": "My property",
  "fromAllOf" : 10
}
```

Invalid JSON:

```
{
  "myProperty": "My property",
  "fromAllOf" : "My value"
}
```

It's also possible to use a reference **$ref** in the **allOf** keyword to reference another schema.

In Unomi, there is an example of using **$ref** in the **allOf** keyword to validate the properties which are

defined in the event schema. This schema contains properties common to all events. It's done in the the view event schema. The file can be found on github: view.json More details about allOf can be found in the specifications: https://json-schema.org/understanding-json-schema/reference/combining.html#allof

unevaluatedProperties keyword

The **unevaluatedProperties** keyword is useful for schema composition as well as enforcing stricter schemas. This keyword is similar to **additionalProperties** except that it can recognize properties declared in sub schemas. When setting the **unevaluatedProperties** value to **false**, the properties which are not present in the properties part and are not present in the sub schemas will be considered as invalid.

Example with the following schema:

```
{
  "$id": "https://unomi.apache.org/schemas/json/example/1-0-0",
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "type": "object",
  "allOf": [
    {
      "$ref": "https://unomi.apache.org/schemas/json/subschema/1-0-0"
    }
  ],
  "properties": {
    "myProperty": {
      "type": "string",
      "$comment": "Example of a property."
    }
  },
  "unevaluatedProperties": false
}
```

Sub schema:

```
{
  "$id": "https://unomi.apache.org/schemas/json/subschema/1-0-0",
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "type": "object",
  "properties": {
    "fromAllOf": {
      "type": "string",
      "$comment": "Example of allOf."
    }
  }
}
```

With the following data, the validation will fail because the property **myNewProperty** is not defined neither the **properties** part nor the **allOf** part.

```
{
   "myProperty": "My property",
   "fromAllOf" : 10,
   "myNewProperty": "another one" //Not valid
}
```

7.1.3. How are JSON Schema used in Unomi

JSON Schema is used in Unomi to validate the data coming from the two public endpoints **/contextRequest** and **/eventCollector**. Both endpoints have a custom deserializer which will begin by validating the payload of the request, then will filter invalid events present in this payload. If an event is not valid it will not be processed by the system. The internal events are not validated by JSON schema as they are not sent through the public endpoints.

In Unomi, each event type must have an associated JSON schema. To validate an event, Unomi will search for a schema in which the target of the schema is **events**, and with the name of the schema matching the event type.

A custom keyword named **self** has to be present in the JSON schemas to store the information related to each schema. The following example is the **self** part of the view event JSON schema. Having the target set to **events** and the name set to **view**, this schema will be used to validate the events of type **view**.

```
...
"self":{
   "vendor":"org.apache.unomi",
   "target" : "events",
   "name": "view",
   "format":"jsonschema",
   "version":"1-0-0"
},
...
```

Link to the schema on github: view.json.

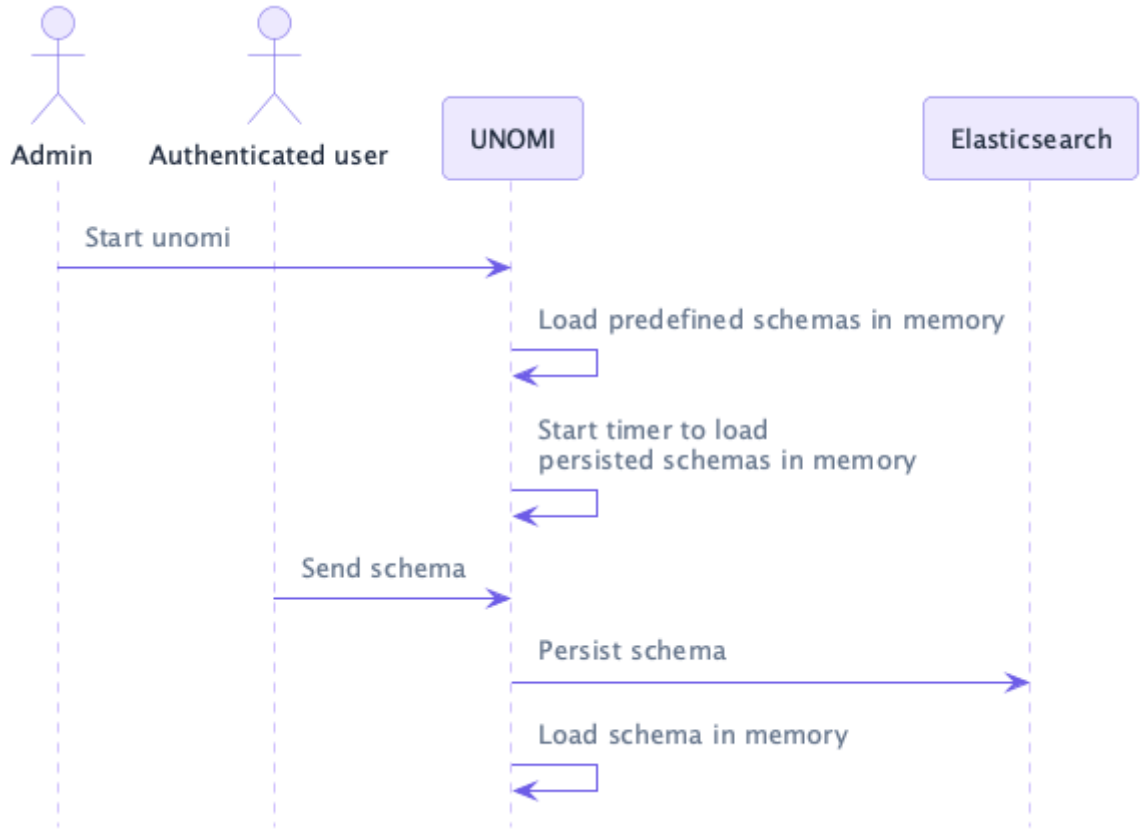
A set of predefined schema are present in Unomi, these schemas can be found under the folder : extensions/json-schema/services/src/main/resources/META-INF/cxs/schemas.

These schemas will be loaded in memory at startup. Each schema where the **target** value is set to **events**, will be used to validate events. The others are simply used as part of JSON schema or can be used in additional JSON schemas.

It's possible to add JSON schemas to validate your own event by using the API, the explanations to manage JSON schema through the API are in the Create / update a JSON schema to validate an event section.

Contrary to the predefined schemas, the schemas added through the API will be persisted in Elasticsearch in the jsonSchema index. Schemas persisted in Elasticsearch do not require a restart of the platform to reflect changes.

Process of creation of schemas:



7.2. JSON schema API

The JSON schema endpoints are private, so the user has to be authenticated to manage the JSON schema in Unomi.



JSON schema endpoints require tenant authentication using Basic Auth with tenantId:privateKey. Only the Tenant API (/cxs/tenants) uses system administrator authentication (the karaf user and your configured UNOMI_ROOT_PASSWORD).

7.2.1. List existing schemas

The REST endpoint GET {{url}}/cxs/jsonSchema allows to get all ids of available schemas and subschemas.

List of predefined schemas:

```
[
  "https://unomi.apache.org/schemas/json/events/modifyConsent/properties/1-0-0",
  "https://unomi.apache.org/schemas/json/item/1-0-0",
  "https://unomi.apache.org/schemas/json/events/login/1-0-0",
  "https://unomi.apache.org/schemas/json/events/modifyConsent/1-0-0",
  "https://unomi.apache.org/schemas/json/consentType/1-0-0",
  "https://unomi.apache.org/schemas/json/items/page/properties/1-0-0",
  "https://unomi.apache.org/schemas/json/items/page/properties/attributes/1-0-0",
  "https://unomi.apache.org/schemas/json/events/incrementInterest/1-0-0",
  "https://unomi.apache.org/schemas/json/events/view/flattenProperties/1-0-0",
  "https://unomi.apache.org/schemas/json/interests/1-0-0",
  "https://unomi.apache.org/schemas/json/items/site/1-0-0",
  "https://unomi.apache.org/schemas/json/items/page/properties/pageInfo/1-0-0",
  "https://unomi.apache.org/schemas/json/rest/requestIds/1-0-0",
  "https://unomi.apache.org/schemas/json/rest/eventscollectorrequest/1-0-0",
  "https://unomi.apache.org/schemas/json/events/view/properties/1-0-0",
  "https://unomi.apache.org/schemas/json/items/page/1-0-0",
  "https://unomi.apache.org/schemas/json/URLParameters/1-0-0",
  "https://unomi.apache.org/schemas/json/event/1-0-0",
  "https://unomi.apache.org/schemas/json/timestampeditem/1-0-0",
  "https://unomi.apache.org/schemas/json/events/updateProperties/1-0-0",
  "https://unomi.apache.org/schemas/json/consent/1-0-0",
  "https://unomi.apache.org/schemas/json/events/incrementInterest/flattenProperties/1-0-0",
  "https://unomi.apache.org/schemas/json/events/view/1-0-0"
]
```


Custom schemas will also be present in this list once added.

7.2.2. Read a schema

It's possible to get a schema by its id by calling the endpoint POST {{url}}/cxs/jsonSchema/query with the id of the schema in the payload of the query.

Example:

```
curl --location --request POST 'http://localhost:8181/cxs/jsonSchema/query' \
--user 'TENANT_ID:PRIVATE_KEY' \
--header 'Content-Type: text/plain' \
--data-raw 'https://unomi.apache.org/schemas/json/event/1-0-0'
```



Replace TENANT_ID and PRIVATE_KEY with your actual tenant ID and private API key. You can obtain these when creating a tenant via the Tenant API (which requires system administrator authentication).

7.2.3. Create / update a JSON schema to validate an event

It's possible to add or update JSON schema by calling the endpoint POST {{url}}/cxs/jsonSchema with the JSON schema in the payload of the request. If the JSON schema exists it will be updated with the new one.

Example of creation:

```
curl --location --request POST 'http://localhost:8181/cxs/jsonSchema' \
--user 'TENANT_ID:PRIVATE_KEY' \
--header 'Content-Type: application/json' \
--data-raw '{
   "$id": "https://vendor.test.com/schemas/json/events/dummy/1-0-0",
   "$schema": "https://json-schema.org/draft/2019-09/schema",
   "self": {
      "vendor": "com.vendor.test",
      "name": "dummy",
      "format": "jsonschema",
      "target": "events",
      "version": "1-0-0"
   },
   "title": "DummyEvent",
   "type": "object",
   "allOf": [
      {
         "$ref": "https://unomi.apache.org/schemas/json/event/1-0-0"
      }
   ],
   "properties": {
      "properties": {
         "$ref": "https://vendor.test.com/schemas/json/events/dummy/properties/1-0-0"
      }
   },
   "unevaluatedProperties": false
}'
```

7.2.4. Deleting a schema

To delete a schema, call the endpoint POST {{url}}/cxs/jsonSchema/delete with the id of the schema into the payload of the request

Example:

```
curl --location --request POST 'http://localhost:8181/cxs/jsonSchema/delete' \
--user 'TENANT_ID:PRIVATE_KEY' \
--header 'Content-Type: text/plain' \
--data-raw 'https://vendor.test.com/schemas/json/events/dummy/1-0-0'
```



Replace TENANT_ID and PRIVATE_KEY with your actual tenant ID and private API key.

7.2.5. Error Management

When calling an endpoint with invalid data, such as an invalid value for the **sessionId** property in the contextRequest object or eventCollectorRequest object, the server would respond with a 400 error code and the message **Request rejected by the server because: Invalid received data**.

7.2.6. Details on invalid events

If it's an event which is incorrect the server will continue to process the request but will exclude the invalid events.

7.3. Develop with Unomi and JSON Schemas

Schemas can be complex to develop, and sometimes, understanding why an event is rejected can be challenging.

This section of the documentation details mechanisms put in place to facilitate the development when working around JSON Schemas (when creating a new schema, when modifying an existing event, ...etc).

7.3.1. Logs in debug mode

Running Apache Unomi with the logs in debug level will add to the logs the reason why events are rejected. You can set the log level of the class validating the events to debug by using the following karaf command:

```
log:set DEBUG org.apache.unomi.schema.impl.SchemaServiceImpl
```

Doing so will output logs similar to this:

```
08:55:28.128 DEBUG [qtp1422628821-128] Schema validation found 2 errors while validating against schema: https://unomi.apache.org/schemas/json/events/view/1-0-0
08:55:28.138 DEBUG [qtp1422628821-128] Validation error: There are unevaluated properties at following paths $.source.properties
08:55:28.140 DEBUG [qtp1422628821-128] Validation error: There are unevaluated properties at following paths $.source.itemId, $.source.itemType, $.source.scope, $.source.properties
08:55:28.142 ERROR [qtp1422628821-128] An event was rejected - switch to DEBUG log level for more information
```

7.3.2. validateEvent endpoint

A dedicated endpoint (requires tenant authentication), accessible at: cxs/jsonSchema/validateEvent, was created to validate events against JSON Schemas loaded in Apache Unomi.

For example, sending an event not matching a schema:

```
curl --request POST \
  --url http://localhost:8181/cxs/jsonSchema/validateEvent \
  --user 'TENANT_ID:PRIVATE_KEY' \
  --header 'Content-Type: application/json' \
  --data '{
    "eventType": "no-event",
    "scope": "unknown_scope",
    "properties": {
        "workspace": "no_workspace",
        "path": "some/path"
    }
}'
```

Would return the following:

```
Request rejected by the server because: Unable to validate event: Schema not found for event type: no-event
```

And if we were to submit a valid event type but make a typo in one of the properties name, the endpoint will point us towards the incorrect property:

```
[
    {
        "error": "There are unevaluated properties at following paths $.source.scopee"
    }
]
```

7.3.3. validateEvents endpoint

A dedicated endpoint (requires tenant authentication), accessible at: cxs/jsonSchema/validateEvents, was created to validate a list of event at once against JSON Schemas loaded in Apache Unomi.

For example, sending a list of event not matching a schema:

```
curl --request POST \
  --url http://localhost:8181/cxs/jsonSchema/validateEvents \
  --user 'TENANT_ID:PRIVATE_KEY' \
  --header 'Content-Type: application/json' \
  --data '[{
    "eventType": "view",
    "scope": "scope",
    "properties": {
       "workspace": "no_workspace",
       "path": "some/path",
       "unknowProperty": "not valid"
    }, {
    "eventType": "view",
    "scope": "scope",
    "properties": {
       "workspace": "no_workspace",
       "path": "some/path",
       "unknowProperty": "not valid",
       "secondUnknowProperty": "also not valid"
    }, {
    "eventType": "notKnownEvent",
    "scope": "scope",
    "properties": {
       "workspace": "no_workspace",
       "path": "some/path"
    }
}]'
```

Would return the errors grouped by event type as the following:

```
{
   "view": [
      {
         "error": "There are unevaluated properties at following paths $.properties.unknowProperty"
      },
      {
         "error": "There are unevaluated properties at following paths
$.properties.secondUnknowProperty"
      }
   ],
   "notKnownEvent": [
      {
         "error": "No Schema found for this event type"
      }
   ]
}
```

If several events have the same issue, only one message is returned for this issue.

7.4. Extend an existing schema

7.4.1. When a extension is needed?

Apache Unomi provides predefined schemas to validate some known events such as a view event.

The Apache Unomi JSON schemas are designed to consider invalid any properties which are not defined in the JSON schema. So if an unknown property is part of the event, the event will be considered as invalid.

This means that if your events include additional properties, you will need extensions to describe these.

7.4.2. Understanding how extensions are merged in unomi

An extension schema is a JSON schema whose id will be overridden and be defined by a keyword named **extends** in the **self** part of the extension.

When sending an extension through the API, it will be persisted in the search engine then will be merged to the targeted schema.

What does “merge a schema” mean? The merge will simply add in the **allOf** keyword of the targeted schema a reference to the extensions. It means that to be valid, an event should be valid against the base schema and against the ones added in the **allOf**.

Example of an extension to allow to add a new property in the view event properties:

```
{
  "$id": "https://vendor.test.com/schemas/json/events/dummy/extension/1-0-0",
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "self":{
    "vendor":"com.vendor.test",
    "name":"dummyExtension",
    "format":"jsonschema",
    "extends": "https://unomi.apache.org/schemas/json/events/view/properties/1-0-0",
    "version":"1-0-0"
  },
  "title": "DummyEventExtension",
  "type": "object",
  "properties": {
    "myNewProp": {
      "type": "string"
    }
  }
}
```

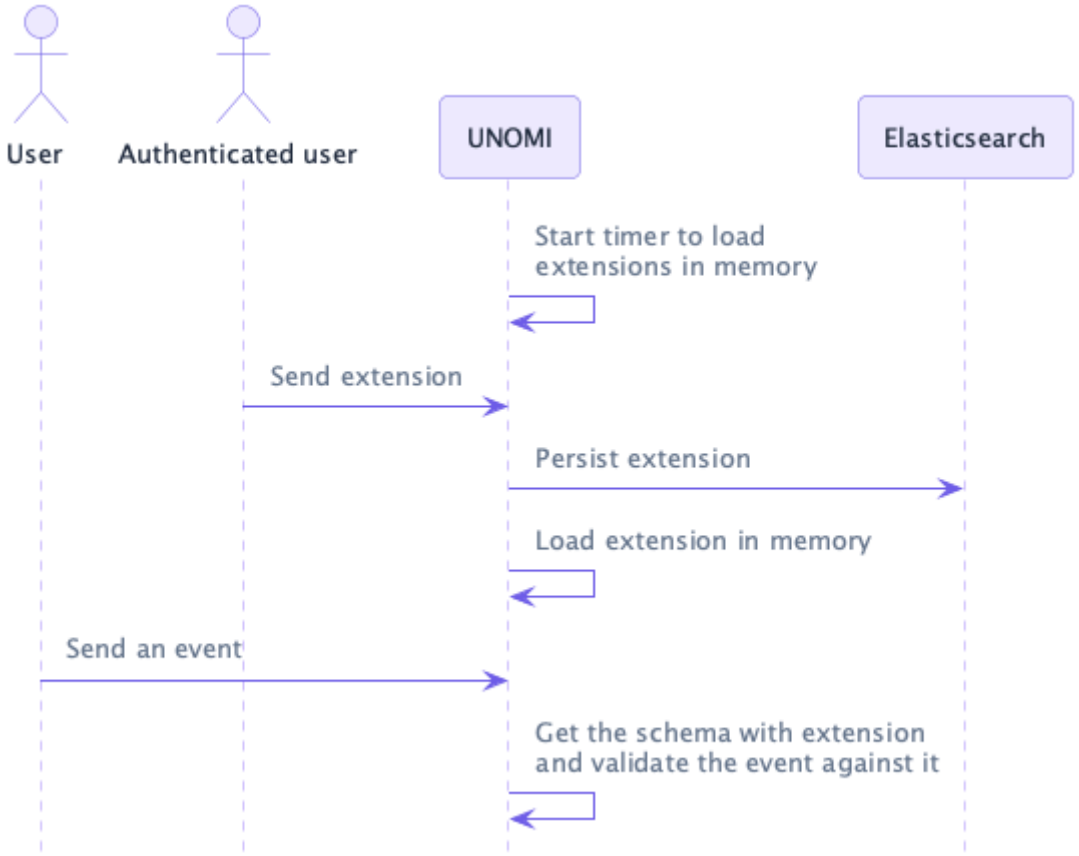
When validating the events of type view, the extension will be added to the schema with the id **https://unomi.apache.org/schemas/json/events/view/properties/1-0-0** like the following:

```
"allOf": [{
  "$ref": "https://vendor.test.com/schemas/json/events/dummy/extension/1-0-0"
}]
```

With this extension the property **myNewProp** can now be added to the event.

```
...
"properties": {
   "myNewProp" : "newValue"
},
...
```

Process when adding extension:



7.4.3. How to add an extension through the API

Since an extension is also a JSON schema, it is possible to add extensions by calling the endpoint to add a JSON schema. By calling POST {{url}}/cxs/jsonSchema with the JSON schema in the payload of the request, the extension will be persisted and will be merged to the targeted schema.

Chapter 8. GraphQL API

8.1. Introduction

First introduced in Apache Unomi 2.0, a GraphQL API is available as an alternative to REST for interacting with the platform. Disabled by default, the GraphQL API is currently considered a beta feature.

We look forward for this new GraphQL API to be used, feel free to open discussion on Unomi Slack channel or create tickets on Jira

8.2. Enabling the API

The GraphQL API must be enabled using the related distribution (unomi-distribution-elasticsearch-graphql or unomi-distribution-opensearch-graphql) or use the dedicated feature (cdp-graphql-feature) in your own.

8.3. Endpoints

Two endpoints were introduced for Apache Unomi 2 GraphQL API: * /graphql is the primary endpoint for interacting programatically with the API and aims at receiving POST requests. * /graphql-ui provides access to the GraphQL UI and aims at being accessed by a Web Browser.

8.4. GraphQL Schema

Thanks to GraphQL introspection, there is no dedicated documentation per-se as the Schema itself serves as documentation.

You can easily view the schema by navigrating to /graphql-ui, depending on your setup (localhost, public host, ...), you might need to adjust the URL to point GraphQL UI to the /graphql endpoint.

8.4.1. Authentication

GraphQL requests follow the same tenant security model as REST:

- **Public read operations** on the cdp root field: send X-Unomi-Api-Key with a tenant **public** API key.
- **Mutations and administrative queries**: HTTP Basic authentication with tenantId:privateApiKey, or JAAS credentials. With JAAS, X-Unomi-Tenant-Id selects the tenant; when it is omitted, Unomi uses the same admin default tenant as REST (default default). The system context needs an explicit X-Unomi-Tenant-Id: system and a full administrator.
- **Introspection** (IntrospectionQuery): allowed without credentials for schema discovery.

See Multi-tenancy for key management.

8.5. Multi-Tenancy Support

The GraphQL API includes comprehensive multi-tenancy support, allowing different tenants to have customized GraphQL schemas based on their specific data models and property types.

8.5.1. Overview

The GraphQL schema provider automatically creates and manages tenant-specific schemas, ensuring:

- Each tenant has its own GraphQL schema that reflects only their specific property types
- Changes to property types trigger automatic schema updates
- Proper tenant isolation is maintained throughout the GraphQL API

8.5.2. Architecture

The multi-tenancy support is built on these key components:

Schema Cache

Each tenant's GraphQL schema is created on demand and cached for performance:

```
private final ConcurrentMap<String, GraphQL> tenantSchemas = new ConcurrentHashMap<>();
```

Tenant Context Detection

When processing a GraphQL request, the system automatically detects the current tenant from the execution context:

```
String tenantId = executionContextManager.getCurrentContext() != null ?
   executionContextManager.getCurrentContext().getTenantId() : null;
```

Dynamic Schema Creation

Each tenant's schema is built dynamically based on its specific property types and configurations:

```
GraphQL graphQL = graphQLSchemaUpdater.getGraphQLForTenant(tenantId);
```

Schema Invalidation

When property types change, the affected tenant's schema is automatically invalidated:

```
public void invalidateTenantSchema(String tenantId) {
    tenantSchemas.remove(tenantId);
}
```

8.5.3. Schema Lifecycle

Tenant schemas follow this lifecycle:

1. **Creation**: Schemas are created on-demand when first requested
2. **Caching**: Created schemas are cached for performance
3. **Invalidation**: When property types change, the affected schemas are invalidated
4. **Regeneration**: Invalid schemas are regenerated on the next request

8.5.4. Performance Considerations

- Schemas are created lazily on first request to avoid unnecessary overhead
- Schema creation can be resource-intensive, so caching is essential
- Property type changes trigger selective schema invalidation to minimize rebuilding
- Only the affected tenant's schema is invalidated when property types change

8.5.5. Benefits

This tenant-aware design provides several advantages:

- **Isolation**: Each tenant has access only to their own data model
- **Customization**: Tenants can define custom property types that appear in their schema
- **Performance**: Schema caching improves response times
- **Consistency**: Changes to property types are immediately reflected in the API

8.5.6. Troubleshooting

If tenant-specific schemas aren't working as expected:

- Check that the tenant context is properly set
- Verify property types have correct tenant IDs
- Review logs for schema creation and invalidation events
- Try explicitly invalidating the tenant schema to force regeneration

8.6. Graphql request examples

You can use embedded GraphiQL interface available at http://localhost:8181/graphql-ui or use any other GraphQL client using that url for requests.

8.6.1. Retrieving your first profile

Profile can be retrieved using getProfile query

```
query($profileID: CDP_ProfileIDInput!, $createIfMissing: Boolean) {
 cdp {
  getProfile(profileID: $profileID, createIfMissing: $createIfMissing) {
   firstName
   lastName
   gender
   cdp_profileIDs {
    client {
     ID
     title
    }
    id
   }
  }
 }
}
```

This query accepts two variables that need to be provided in the Query variables section:

```
{
 "profileID": {
  "client":{
   "id": "defaultClientId"
  },
  "id": 1001
 },
 "createIfMissing": true
}
```



If you don't want profile to be created if missing, set createIfMissing to false.

The response will look like this:

```
{
 "data": {
  "cdp": {
   "getProfile": {
    "firstName": null,
    "lastName": null,
    "gender": null,
    "cdp_profileIDs": [
     {
      "client": {
       "ID": "defaultClientId",
       "title": "Default Client"
      },
      "id": "1001"
     }
    ]
   }
  }
 }
}
```

8.6.2. Updating profile

Now let's update our profile with some data. It can be done using processEvents mutation:

```
mutation($events: [CDP_EventInput]!) {
 cdp {
  processEvents(events: $events)
 }
}
```

This mutation accepts one variable that needs to be provided in the Query variables section:

```
{
 "events": [
  {
   "cdp_objectID": 1001,
   "cdp_profileID": {
    "client": {
     "id": "defaultClientId"
    },
    "id": 1001
   },
   "cdp_profileUpdateEvent": {
    "firstName": "John",
    "lastName": "Doe",
    "gender": "Male"
   }
  }
 ]
}
```

The response will have the number of processed events:

```
{
 "data": {
  "cdp": {
   "processEvents": 1
  }
 }
}
```



processEvents accepts a number of other event types that are listed on CDP_EventInput type.

If you run the getProfile query again, you will see that the profile has been updated.

8.6.3. Restricted methods

Some methods are restricted to authenticated users only. One example is findProfiles query:

```
query {
 cdp {
  findProfiles {
   totalCount
    edges {
     node {
      cdp_profileIDs {
       client{
        title
        ID
       }
       id
      }
     }
    }
  }
 }
}
```

And if you run it now, you will get an error.

To make this query work you need to supply authorization token in the HTTP headers section:

```
{
 "authorization": "Basic BASE64_OF_KARAF_AND_ROOT_PASSWORD"
}
```

Generate the Base64 value from your own credentials, for example with printf 'karaf:%s' "$UNOMI_ROOT_PASSWORD" | base64.



GraphQL requests need authentication. For **mutations and administrative queries**, use HTTP Basic with JAAS (the karaf user and your configured UNOMI_ROOT_PASSWORD) or tenantId:privateApiKey. For **public read** operations on the cdp root field, send X-Unomi-Api-Key with a tenant public API key. See GraphQL API authentication.

When using curl, you can use the --user option instead of manually encoding credentials. For example, --user "karaf:$UNOMI_ROOT_PASSWORD" or --user TENANT_ID:PRIVATE_KEY automatically handles Base64 encoding for Basic authentication. For public reads:

```
curl -X POST http://localhost:8181/graphql \
  -H "Content-Type: application/json" \
  -H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
  -d '{"query":"{ cdp { ... } }"}'
```

The result will now show the list of profiles:

```
{
  "data": {
    "cdp": {
      "findProfiles": {
        "totalCount": 1,
        "edges": [
          {
            "node": {
              "cdp_profileIDs": [
                {
                  "client": {
                    "title": "Default Client",
                    "ID": "defaultClientId"
                  },
                  "id": "1001"
                }
              ]
            }
          }
        ]
      }
    }
  }
}
```

8.6.4. Deleting profile

Profile can be deleted using deleteProfile mutation:

```
mutation($profileID: CDP_ProfileIDInput!) {
  cdp {
    deleteProfile(profileID: $profileID)
  }
}
```

This mutation accepts one variable that needs to be provided in the Query variables section:

```
{
  "profileID": {
    "client":{
      "id": "defaultClientId"
    },
    "id": 1001
  }
}
```

The response will show the result of the operation:

```
{
  "data": {
    "cdp": {
      "deleteProfile": true
    }
  }
}
```

8.6.5. Where to go from here

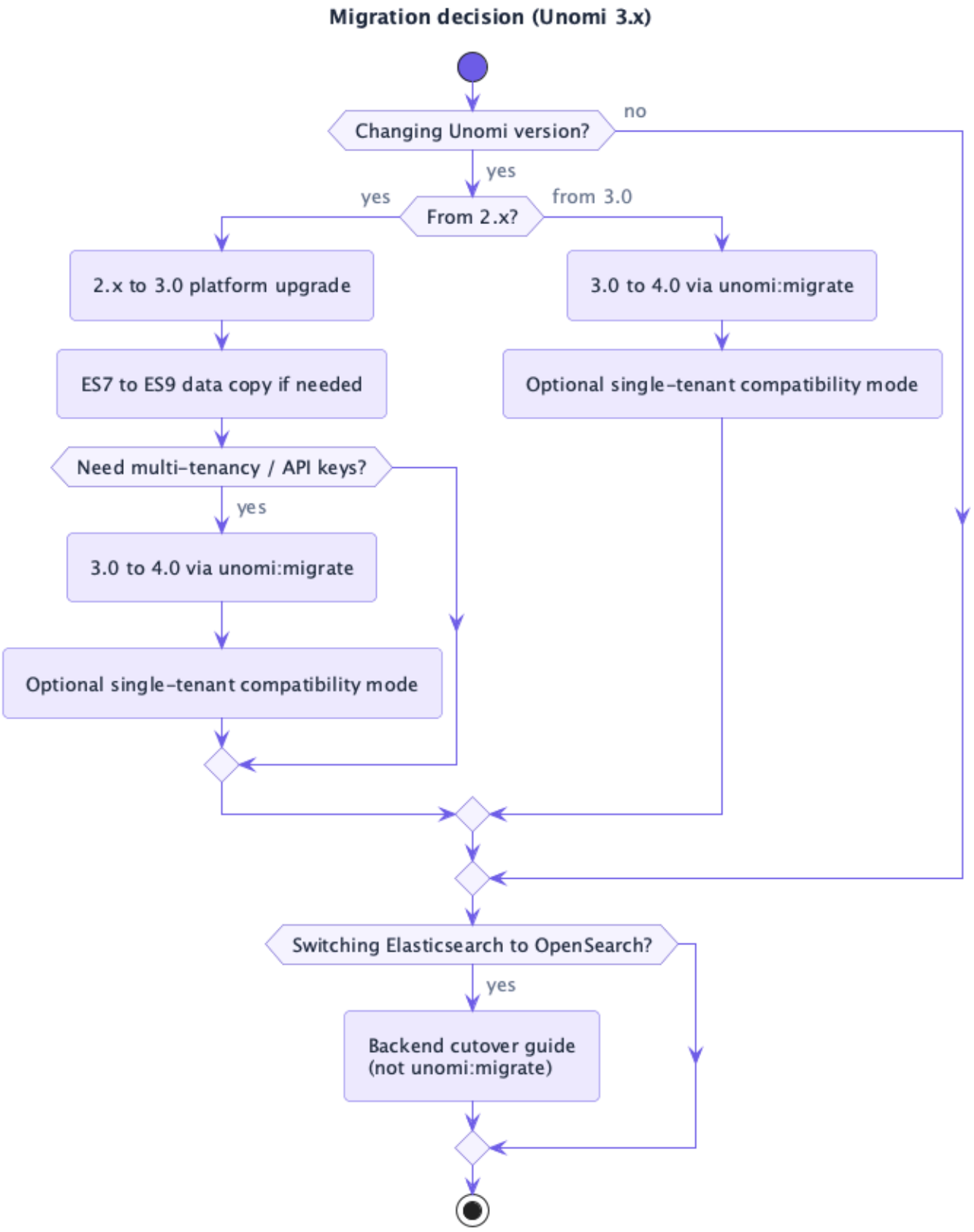
- You can find more useful Apache Unomi URLs that can be used in the same way as the above examples.
- Read GraphQL documentation to learn more about GraphQL syntax.

Chapter 9. Migrations

This section contains information and steps to migrate between major Unomi versions.

Use this decision guide to pick the right runbook:

- **Unomi 2.x → 3.0** (platform): Migrate from 2.x to 3.0 (+ ES7→ES9 if needed)
- **Unomi 3.0 → 4.0** (tenants / API keys / client hardening): Migrate from 3.0 to 4.0 (unomi:migrate); optional single-tenant compatibility mode for Unomi 2 or Unomi 3 clients; see also client-facing hardening
- **Elasticsearch → OpenSearch** (same Unomi version, backend swap): Migrate from Elasticsearch to OpenSearch (not unomi:migrate)



9.1. Writing migration scripts

This section is for **contributors** who add or change Groovy migration scripts shipped in the shell-commands module. Operators upgrading a cluster should follow the version-specific guides in this chapter instead.

Migration scripts run with **Apache Unomi stopped**, via unomi:migrate (see Shell commands). They talk

directly to Elasticsearch or OpenSearch over HTTP and transform persisted data in place.

9.1.1. Script location and naming

Bundled scripts live under:

```
tools/shell-commands/src/main/resources/META-INF/cxs/migration/
```

Each file must match:

```
migrate-<version>-<priority>-<name>.groovy
```

- <version> — three-part version the script belongs to (for example 4.0.0)
- <priority> — two-digit order within that version (00, 01, 05, ...)
- <name> — short camelCase label (for example tenantDocumentIds)

Example: migrate-4.0.0-01-tenantDocumentIds.groovy.

Scripts are sorted by version, then priority, then name. Use gaps in priority (00, 05, 10) when you may need to insert a step later.

Operators can also drop scripts under {karaf.data}/migration/scripts/ for one-off fixes; the same naming rules apply.

9.1.2. Use performMigrationStep for every resumable unit of work

MigrationContext.performMigrationStep(stepKey, closure) is the idempotency and **crash-recovery** mechanism:

- Each stepKey is persisted in {karaf.data}/migration/history.json.
- If a step is already COMPLETED, it is skipped on the next run.
- If a run fails mid-step, only steps not yet marked complete are executed again.

Rules:

1. **One logical outcome per step key** — for example “configure session rollover alias”, not “do everything in one giant step”.
2. **Step keys must be stable** — changing a key after release makes resume behaviour confusing for operators who already have history files.
3. **Prefer many small steps over one large step** — especially when a step can take a long time (reindex, scroll, bulk update).

MigrationUtils.reIndex(...) already registers its own sub-steps (clone, recreate index, delete clone,

refresh). You do not need to wrap those again unless you add work outside reIndex.

9.1.3. Pitfall: do not nest performMigrationStep inside another step

This was the root cause of UNOMI-943.

If step B is registered **inside** the closure of step A, then:

- B is only evaluated while A runs.
- When A is already marked COMPLETED in history, A's closure is never entered again.
- B is never registered and never runs — even if B never completed.

Real impact: rollover **write aliases** for event/session indices were skipped after a failed/resumed migration, so new writes failed silently.

Wrong — alias configuration hidden inside get-all-indices:

```
context.performMigrationStep("4.0.0-get-all-indices", () -> {
   // ... reindex each index ...
   context.performMigrationStep("4.0.0-configure-rollover-aliases", () -> {
      // configure aliases — NEVER REACHED on resume if outer step completed
   })
})
```

Right — each tracked step at script top level:

```
context.performMigrationStep("4.0.0-get-all-indices", () -> {
   // ... reindex each index ...
})

context.performMigrationStep("4.0.0-configure-rollover-aliases", () -> {
   // configure aliases — runs independently on resume
})
```

9.1.4. Pitfall: make data transforms safe to re-run

Step history prevents re-running a **whole** step, but partial failure inside a step can still leave mixed state. Design Painless scripts and update queries so a second application does not corrupt data.

Good patterns:

- **Check before mutate** — skip documents that already have the target shape (tenant prefix on _id, tenantId set, legacy field already renamed).
- **Copy-once fields** — only copy nbOfVisits totalNbOfVisits when totalNbOfVisits is null.
- **Existence guards** — create an index or tenant document only when MigrationUtils.indexExists(...) is false.

- **Scoped queries** — limit update_by_query to documents that still match the old state (wildcard on legacy queryBuilder IDs, etc.).

Example guard (document ID already tenant-prefixed):

```
if (!ctx._id.startsWith(params.tenantId + '_') && !ctx._id.startsWith(params.systemTenantId + '_')) {
    // transform ...
}
```

9.1.5. Pitfall: bundled resources must exist and fail clearly

Scripts load JSON and Painless from the bundle via:

- MigrationUtils.resourceAsString(bundleContext, "requestBody/...")
- MigrationUtils.getFileWithoutComments(bundleContext, "requestBody/.../script.painless")

Both throw RuntimeException("Resource not found: ...") when the path is wrong. A missing file used to surface as an opaque NullPointerException in getFileWithoutComments (fixed in UNOMI-943).

Keep request bodies under tools/shell-commands/src/main/resources/requestBody/<version>/. After adding a file, run unit tests or a dry migration in a staging cluster — do not rely on compile-only checks.

9.1.6. Configuration keys

Prefer constants from MigrationConfig instead of raw string literals:

```
import static org.apache.unomi.shell.migration.service.MigrationConfig.*

String esAddress = context.getConfigString(CONFIG_ES_ADDRESS)
String indexPrefix = context.getConfigString(INDEX_PREFIX)
String tenantId = context.getConfigString(TENANT_ID)
```

Legacy scripts may still use "esAddress" and "indexPrefix"; new scripts should use the constants.

Optional settings and prompts are documented in org.apache.unomi.migration.cfg (see shell command help for unomi:migrate).

9.1.7. Destructive operations

Reindex and index deletion are intentional but risky:

- MigrationUtils.reIndex clones the source index, deletes the original, recreates mappings, then copies data back.
- Always confirm the index name is not a -cloned suffix (the helper rejects reindexing clones).
- For rollover indices, configure **aliases in a separate top-level step** after all reindex work finishes.

Log progress with context.printMessage(...) so operators can correlate Karaf console output with history.json.

9.1.8. Testing

Test	Module	What it validates
MigrationUtilsTest	tools/shell-commands	Painless comment stripping, missing resource errors
MigrationIT	itests	Step history recovery (checkMigrationRecoverySystem); nested-step pitfall guard (checkNestedMigrationStepPitfall, UNOMI-943)
Migrate16xToCurrentVersionIT	itests	Full chain from 1.6.x snapshot through all bundled scripts (Elasticsearch only)

When you add or change a 3.x script, extend Migrate16xToCurrentVersionIT if the change affects observable data (tenant IDs, aliases, queryBuilder IDs, profile fields, etc.).

Run locally before opening a PR:

```
mvn -pl tools/shell-commands -am test -Dtest=MigrationUtilsTest
```

Full migration ITs run in CI on the integration matrix (~2 h); they are required for script changes that affect persisted data shape.

9.1.9. Pre-merge checklist

Use this before submitting a migration PR:

- ☐ File name matches migrate-<version>-<priority>-<name>.groovy
- ☐ Every resumable unit uses its own **top-level** performMigrationStep (no nesting)
- ☐ Step keys are stable and describe one outcome
- ☐ Painless / update logic is safe if applied twice inside a step
- ☐ New requestBody/ assets are committed and referenced with correct paths
- ☐ indexExists / query filters guard create-only work
- ☐ Unit or IT coverage updated where behaviour is testable
- ☐ Version-specific operator doc updated if the upgrade path changes (migrate-**to**-.adoc)

9.2. Unomi 2 / 3 to 4.0 API compatibility

9.3. Apache Unomi 2 / 3 to 4.0 API differences

This document explains the key API differences between Apache Unomi 2.x / 3.x and Unomi 4.0.

9.3.1. Overview

Apache Unomi **4.0** introduces comprehensive multi-tenancy support, enabling complete data isolation between different tenants. This fundamental architectural change requires a new tenant-based authentication model while keeping all API endpoints unchanged.

Multi-Tenancy in 4.0

The key innovation in Unomi 4.0 is the introduction of **tenant isolation** for all data:

- **Profiles, events, segments, rules, and schemas** are now tenant-specific
- **Complete data separation** between tenants - no cross-tenant data access
- **Tenant-specific API keys** for secure access control
- **Backward compatibility** with system administrator access for management operations

This multi-tenancy support necessitates the authentication changes described below, as the system must now identify which tenant context to operate in for every request.

9.3.2. Key differences between Unomi 2 / 3 and Unomi 4

Authentication Model

Aspect	Unomi 2 / 3	Unomi 4.0
Authentication Method	System Administrator Authentication (karaf user)	Tenant-based API Keys + System Administrator Authentication
Public API Endpoints	No authentication required	Public API Key required (via X-Unomi-Api-Key header)
Private API Endpoints	System Administrator Authentication	Tenant Authentication (tenantId/privateKey) OR System Administrator Authentication
Tenant Administration	System Administrator Authentication (karaf user)	System Administrator Authentication (karaf user)

API Key Types (4.0 only)

Unomi 4.0 introduces two types of API keys per tenant:

- **Public Key**: Used for public endpoints (event collection via /context.json)
- **Private Key**: Used with tenantId for tenant-specific administrative operations

Authentication Requirements by Endpoint Type

Endpoint Category	Unomi 2 / 3 Authentication	Unomi 4.0 Authentication
Event Collection (/context.json)	None	Public API Key only
Administrative Operations	System Admin (karaf user)	Tenant Auth (tenantId/privateKey) OR System Admin (karaf user)
Tenant Administration (/cxs/tenants)	System Admin (karaf user)	System Admin (karaf user)

9.3.3. Authentication Flow (4.0)

The AuthenticationFilter in 4.0 follows this resolution order:

1. **Tenant endpoints** (/cxs/tenants): Requires system administrator authentication only
2. **Public endpoints** (e.g., /context.json): Requires public API key via X-Unomi-Api-Key header
3. **Private endpoints**: Tries tenant authentication first, then falls back to system administrator authentication:
   - **Tenant Authentication**: Basic Auth with tenantId:privateKey
   - **System Administrator Authentication**: Basic Auth with the karaf user and your configured UNOMI_ROOT_PASSWORD

9.3.4. Code Examples

Unomi 2 / 3 Authentication

```
// Global system administrator authentication for all endpoints
RestAssured.authentication = RestAssured.preemptive()
    .basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"));

// Context requests require no authentication
RestAssured.given()
    .auth().none()
    .contentType(ContentType.JSON)
    .body(contextJson)
    .post("/context.json");
```

Unomi 4.0 Authentication

```
// For public endpoints (event collection)
given()
    .header("X-Unomi-Api-Key", publicKey)
    .contentType(ContentType.JSON)
    .body(contextJson)
    .post("/context.json");

// For private endpoints using tenant authentication
given()
    .auth().preemptive().basic(tenantId, privateKey)
    .contentType(ContentType.JSON)
    .body(payload)
    .post("/cxs/profiles");

// For private endpoints using system administrator authentication
given()
    .auth().preemptive().basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"))
    .contentType(ContentType.JSON)
    .body(payload)
    .post("/cxs/profiles");

// For tenant administration (system admin only)
given()
    .auth().preemptive().basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"))
    .contentType(ContentType.JSON)
    .body(tenantPayload)
    .post("/cxs/tenants");
```

9.3.5. Implementation Strategy

Client Factory Pattern

```
public class UnomiConfiguration {
    public UnomiClient createClient(String baseUrl) {
        String version = System.getProperty("unomi.version", "4.0");

        if ("4.0".equals(version)) {
            return new UnomiV40Client(baseUrl);
        } else {
            return new UnomiLegacyClient(baseUrl);
        }
    }
}
```

Version-Specific Authentication

```
// Unomi 2 / 3 Client
public void init() {
   RestAssured.baseURI = baseUrl;
   RestAssured.authentication = RestAssured.preemptive()
      .basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"));
}

// Unomi 4.0 Client
public void init() {
   RestAssured.baseURI = baseUrl;
}

public void updateKeys(String publicKey, String privateKey) {
   this.publicKey = publicKey;
   this.privateKey = privateKey;
}
```

9.3.6. Migration Guidelines

From Unomi 2 or 3 to Unomi 4

1. **Understand Multi-Tenancy Impact**
   - All data (profiles, events, segments, rules, schemas) becomes tenant-specific
   - Each tenant operates in complete isolation with their own data space
   - Tenant context must be established for every API operation
2. **Update Authentication Configuration**
   - Configure tenant-specific public and private API keys
   - Or, while clients are being updated, enable single-tenant compatibility mode
   - Implement endpoint-specific authentication logic
3. **Endpoint-Specific Changes**
   - Add X-Unomi-Api-Key header with public key for event collection
   - Use tenant authentication (tenantId/privateKey) for tenant-specific administrative operations
   - Keep system administrator authentication as fallback for administrative operations
   - Continue using system administrator authentication for tenant administration
4. **API paths stay the same**
   - Endpoint paths remain the same
   - Most request/response payloads keep the same shape
   - Authentication and tenant resolution differ; see the 3.0→4.0 migration guide for other breaking changes

Benefits of Multi-Tenancy in 4.0

- **Data Isolation**: Complete separation ensures tenant data never crosses boundaries
- **Scalability**: Support for multiple customers/organizations in a single Unomi instance
- **Security**: Tenant-specific API keys prevent unauthorized cross-tenant access
- **Compliance**: Easier to meet data privacy regulations with clear tenant boundaries
- **Cost Efficiency**: Shared infrastructure with isolated data reduces operational costs

9.3.7. Conclusion

The fundamental difference between Unomi 2 / 3 and Unomi 4 is the introduction of **comprehensive multi-tenancy support**:

- **Unomi 2 / 3**: Single-tenant architecture with system administrator authentication for all operations
- **Unomi 4**: Multi-tenant architecture with complete data isolation and tenant-specific authentication
- **API Endpoints**: Paths stay the same; authentication and some behaviours change (see the migration guide)
- **Data Model**: All entities (profiles, events, segments, rules, schemas) become tenant-specific in 4.0
- **Authentication**: New tenant-based authentication model with system administrator authentication as fallback

The authentication changes in 4.0 are driven by the need to establish tenant context for every operation, ensuring complete data isolation while maintaining backward compatibility for administrative operations.

9.4. single-tenant compatibility mode

This document explains how to use single-tenant compatibility mode in Apache Unomi 4.0. The mode lets client applications written for Unomi 2.x or Unomi 3.x keep working without tenant API keys while you move them to the 4.0 authentication model.

9.4.1. Overview

The mode is a temporary bridge for Unomi 2 and Unomi 3 clients. It keeps public endpoints reachable without API keys, while Unomi 4 still stores data under tenants. Every REST request in this mode uses tenant default.

How It Works

When single-tenant compatibility mode is enabled:

- **Public endpoints** (like /context.json) require no authentication (like Unomi 2 and Unomi 3)
- **Protected events** (like login, updateProperties) require IP + X-Unomi-Peer (like Unomi 2 and Unomi 3 third-party servers)

- **Private endpoints** require system administrator authentication (like Unomi 2 and Unomi 3)
- **Tenant default** is used for every operation. The mode cannot be pointed at another tenant: it ignores X-Unomi-Tenant-Id and the defaultTenantId setting on REST. GraphQL is not affected by the mode and still uses defaultTenantId for JAAS admin requests without a header, so keep that setting at default while the mode is on. The tenant is created on a public request if it is missing. Each node then keeps it in memory for about 30 seconds, so public requests do not wait on each other or re-read it from the database every time. A change made on another node is picked up after about that time.
- **No authentication** is required for non-protected events (like Unomi 2 and Unomi 3)
- **The visitor cookie is limited to listed websites** (unlike Unomi 2.x and 3.x, which sent credentialed CORS headers to every origin; their SameSite=Lax cookie still reached only pages on the same site). A website on another origin, another subdomain included, must be listed in UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS or in the default tenant's publicCorsAllowedOrigins property, for its requests to use and receive the cookie, whether it calls context.json or embeds context.js with a <script> tag. A plain <script> tag (without a crossorigin attribute) sends no Origin, so the listed website is then recognised by its Referer, which must be turned on with org.apache.unomi.public.cors.matchReferer=true (read the trade-offs first). A website served from Unomi's own origin needs no change. See Compatibility with Unomi 2.x and 3.x.

This allows Unomi 2 and Unomi 3 clients to work with Unomi 4.0 immediately after migration, giving you time to update applications to tenant API keys. Tracker 1.x is one of those clients: it keeps working while this mode is on, because it cannot send an API key. When you turn the mode off, those pages need tracker 2.0.0 or later and apiKey. See Upgrading the tracker from 1.x to 2.0.

Unomi 2 and Unomi 3 clients behave the same way in this mode. Third-party / peer servers keep using X-Unomi-Peer and etc/org.apache.unomi.thirdparty.cfg while the mode is on; that path is not used in normal multi-tenant mode, where the tenant's authorizedIPs apply instead. See Secured events configuration and UNOMI-1001.

9.4.2. Prerequisites

Before enabling single-tenant compatibility mode, ensure that:

1. **Data Migration Completed**: Your data has been migrated into tenant default using the migration scripts. The mode does not read data that lives in any other tenant.
2. **Unomi 4.0 Installation**: Unomi 4.0 is properly installed and configured. If tenant default is missing, the first public request creates it.

9.4.3. Configuration

Enable single-tenant compatibility mode

You can also set the environment variable UNOMI_REST_AUTHENTICATION_SINGLETENANTCOMPATIBILITYMODEENABLED=true (maps to org.apache.unomi.rest.authentication.singleTenantCompatibilityModeEnabled in custom.system.properties).

1. **Edit the configuration file**: bash # Edit the configuration file vi etc/org.apache.unomi.rest.authentication.cfg
2. **Enable single-tenant compatibility mode**: properties # Enable single-tenant compatibility mode. # The mode always uses tenant "default". There is no setting for another tenant id. singletenantcompatibility.enabled = true
3. **Restart the server** to apply the configuration changes: ```bash # Stop the server ./bin/stop

```
# Start the server
./bin/start
```

Configuration Management

single-tenant compatibility mode is managed through configuration files only. This approach is safer and prevents accidental changes to authentication settings.

9.4.4. Migration Workflow

Step 1: Migrate Data

First, migrate your data with Apache Unomi stopped using the single shell command (bundled Groovy scripts under META-INF/cxs/migration/ run automatically for versions after the origin you pass):

```
# From the Karaf shell (Unomi runtime stopped). Example when upgrading from 2.x through 4.0:
unomi:migrate 2.0.0
```

The migrate-4.0.0-10-tenantInitialization script (run as part of that chain when migrating through 4.0) creates tenant default, which is the only tenant this mode uses. If that tenant is missing when the mode is on, the first public request creates it. See Migrate from 3.0 to 4.0 and Shell commands.

Step 2: Enable single-tenant compatibility mode

Enable single-tenant compatibility mode by updating the configuration file:

```
# Edit the configuration file
vi etc/org.apache.unomi.rest.authentication.cfg

# Set singletenantcompatibility.enabled = true
# The mode always uses tenant "default".

# Restart the server to apply changes
./bin/stop
./bin/start
```

Step 3: Test Unomi 2 / Unomi 3 clients

Your existing clients should now work without API key changes:

```
// Unomi 2 / 3 style authentication still works
RestAssured.authentication = RestAssured.preemptive()
    .basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"));

// Context requests work without API keys
RestAssured.given()
    .auth().none()
    .contentType(ContentType.JSON)
    .body(contextJson)
    .post("/context.json");
```

Step 4: Gradual Migration

Over time, gradually update your clients to use 4.0 authentication:

1. **Update client applications** to use API keys
2. **Test with 4.0 authentication** while keeping single-tenant compatibility mode enabled
3. **Disable single-tenant compatibility mode** once all clients are updated

9.4.5. Client Migration Examples

From Unomi 2 or 3 to Unomi 4 (with compatibility mode)

Legacy client (continues to work):

```
// This continues to work in single-tenant compatibility mode
RestAssured.authentication = RestAssured.preemptive()
    .basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"));

RestAssured.given()
    .auth().none()
    .contentType(ContentType.JSON)
    .body(contextJson)
    .post("/context.json");
```

Unomi 4 client (new implementation):

```
// New 4.0 client using API keys
given()
    .header("X-Unomi-Api-Key", publicKey)
    .contentType(ContentType.JSON)
    .body(contextJson)
    .post("/context.json");
```

Gradual Migration Strategy

1. **Phase 1**: Enable single-tenant compatibility mode, Unomi 2 / 3 clients continue working
2. **Phase 2**: Develop and test 4.0 clients alongside legacy clients
3. **Phase 3**: Migrate clients one by one to 4.0 authentication
4. **Phase 4**: Disable single-tenant compatibility mode once all clients are migrated

9.4.6. Security Considerations

Compatibility mode security

When single-tenant compatibility mode is enabled:

- **Public endpoints** are accessible without authentication (same as Unomi 2 and Unomi 3)
- **Protected events** require IP + X-Unomi-Peer authentication (same as Unomi 2 / 3 peer servers)
- **Private endpoints** require system administrator authentication (same as Unomi 2 and Unomi 3)
- **All operations** use tenant default
- **Non-protected events** require no authentication (same as Unomi 2 and Unomi 3)

Protected Events and Peer Abilities

In single-tenant compatibility mode, a server that sends a valid X-Unomi-Peer key from an allowed IP address keeps the same abilities it had in Unomi 3.0 on **public** endpoints only:

- send protected events listed for that provider (for example login, updateProperties)
- set the event item id
- choose the profile id in the request body
- merge profiles on login / claim a merge identifier
- update another profile's properties, segments, scores, consents and systemProperties

The peer header does **not** open private / admin endpoints, and it does nothing when compatibility mode is off. Visitors without a valid peer key still get none of these abilities (UNOMI-975 / UNOMI-978 stay in force for them).

By default the following event types are protected for provider1:

- login - User authentication events
- updateProperties - Profile property updates

For protected events, clients must:

1. Send the request from an authorized IP address (or from behind a configured trusted reverse proxy)

2. Include the X-Unomi-Peer header with the provider **key** (the secret, not the provider name)

All other event types are non-protected and require no peer header.

Third-party configuration (compatibility mode only)

Providers are configured in etc/org.apache.unomi.thirdparty.cfg with the pattern thirdparty.{providerName}.{property}:

```
# Global (optional)
# allowExampleKey=true logs a warning at every start. Never use it in production: the example key is public
allowExampleKey=${org.apache.unomi.thirdparty.allowExampleKey:-false}
# When the direct connection is in this list, the last X-Forwarded-For entry that is not a trusted proxy is used for IP checks
trustedProxies=${org.apache.unomi.thirdparty.trustedProxies:-}

# Provider 1 — replace the key before production use
thirdparty.provider1.key=${org.apache.unomi.thirdparty.provider1.key:-670c26d1cc413346c3b2fd9ce65dab41}
thirdparty.provider1.ipAddresses=${org.apache.unomi.thirdparty.provider1.ipAddresses:-127.0.0.1,::1}
thirdparty.provider1.allowedEvents=${org.apache.unomi.thirdparty.provider1.allowedEvents:-login,updateProperties}
thirdparty.provider1.abilities=${org.apache.unomi.thirdparty.provider1.abilities:-setEventId,chooseProfileId,mergeOnLogin,updateOtherProfiles}

# Additional providers
thirdparty.myapp.key=replace-with-a-long-unique-secret
thirdparty.myapp.ipAddresses=192.168.1.0/24
thirdparty.myapp.allowedEvents=login,updateProperties,sessionCreated
# abilities omitted  all four Unomi 3.0 peer abilities
```

Ability names are case-sensitive. A provider with no abilities property gets all four. An empty value (thirdparty.myapp.abilities=) grants no peer ability, so the provider can only send its protected events. A value that names no known ability, for example a misspelt one, disables the provider and logs an error, so a typo never widens what a provider may do.

When an X-Unomi-Peer header is present but not accepted, Unomi logs a warning that says whether the key is unknown or the caller's IP address is not in the provider's allowlist. The request then continues as an ordinary visitor.

A provider whose key is empty, shorter than 16 characters, or equal to the well-known example value is **disabled** at startup with a clear error, unless allowExampleKey=true. Keys are compared in constant time and never written in full to the logs.

Configuration Management

The third-party configuration supports dynamic updates:

1. **Edit the configuration file**: bash # Edit the third-party configuration vi etc/org.apache.unomi.thirdparty.cfg
2. **Update protected events**: properties # Add more protected event types thirdparty.provider1.allowedEvents=${org.apache.unomi.thirdparty.provider1.allowedEvents:-login,updateProperties,sessionCreated,profileUpdated}
3. **Add additional providers**: ```properties # Configure additional providers (any name is supported) thirdparty.myapp.key=${org.apache.unomi.thirdparty.myapp.key:-your-secret-key-here} thirdparty.myapp.ipAddresses=${org.apache.unomi.thirdparty.myapp.ipAddresses:-192.168.1.0/24,10.0.0.1} thirdparty.myapp.allowedEvents=${org.apache.unomi.thirdparty.myapp.allowedEvents:-login,updateProperties}

   thirdparty.analytics.key=${org.apache.unomi.thirdparty.analytics.key:-analytics-secret} thirdparty.analytics.ipAddresses=${org.apache.unomi.thirdparty.analytics.ipAddresses:-10.0.0.0/8} thirdparty.analytics.allowedEvents=${org.apache.unomi.thirdparty.analytics.allowedEvents:-login,updateProperties,sessionCreated} ```
4. **Restart the server** to apply changes: bash ./bin/stop ./bin/start

IP allowlists and reverse proxies

Peer checks use the TCP peer Unomi sees (remoteAddr). They only read X-Forwarded-For when that peer is listed in trustedProxies. That stops a client on the open network from inventing a client IP in a header.

Operators should still treat IP allowlists as a second factor next to the peer key, not as the only control:

- Always set an explicit thirdparty.{name}.ipAddresses list (single addresses or CIDR ranges). The list is required: a provider with an empty list is refused from every address, as in Unomi 3.0, and Unomi logs an error for it at startup. The key alone is never enough. An entry that is not an address or CIDR range is reported as an error at startup and never matches.
- Keep trustedProxies limited to reverse proxies that sit in front of Unomi, and list every proxy in the chain. Unomi reads X-Forwarded-For from the last entry backwards and uses the first entry that is not a trusted proxy, so an entry a client puts at the front of the header is ignored whether the proxy appends to the header or overwrites it.
- Prefer placing Unomi (or the peer callers) on a private network or behind a firewall so only known hosts can reach the public endpoints.
- Rotate peer keys and keep them long and unique. For stronger assurance than key + IP, put mutual TLS or equivalent at the edge in front of Unomi.

Recommendations

1. **Use single-tenant compatibility mode temporarily** during migration
2. **Plan for gradual migration** to 4.0 authentication

3. **Monitor access patterns** during the transition
4. **Disable single-tenant compatibility mode** once migration is complete
5. **Configure non-empty IP allowlists** and minimal trustedProxies before production use

9.4.7. Troubleshooting

Common Issues

Legacy clients still not working: - Check configuration file: etc/org.apache.unomi.rest.authentication.cfg - Verify singletenantcompatibility.enabled = true - Confirm migrated data is in tenant default. The mode cannot be pointed at another tenant.

Authentication errors: - Verify system administrator credentials (the karaf user and your configured UNOMI_ROOT_PASSWORD) - Check that the server is running properly - Review logs for authentication errors

Tenant context issues: - Migrated data must be in tenant default. Other tenants are not used while the mode is on. - If tenant default is missing, the first public request creates it. A private request does not. - Check tenant configuration in the migration scripts

Debugging

Enable debug logging for authentication:

```
# Enable debug logging
log:set DEBUG org.apache.unomi.rest.authentication
```

Check authentication filter logs:

```
# View recent logs
log:display | grep AuthenticationFilter
```

9.4.8. Disabling single-tenant compatibility mode

Once all clients are migrated to 4.0 authentication:

1. **Update configuration**: properties singletenantcompatibility.enabled = false
2. **Restart the server**: bash ./bin/stop ./bin/start
3. **Verify all clients work** with 4.0 authentication
4. **Monitor for any issues** and address them before final deployment

9.4.9. Testing single-tenant compatibility mode

The integration test SingleTenantCompatibilityModeIT (in itests) covers this mode. It turns the mode on and off at runtime through the singletenantcompatibility.enabled setting of org.apache.unomi.rest.authentication, so it runs as part of the normal integration test suite; no extra system property is needed.

9.4.10. Conclusion

The single-tenant compatibility mode provides a smooth migration path from Unomi 2 or Unomi 3 to Unomi 4, allowing you to:

- **Maintain existing clients** during migration
- **Gradually migrate** to 4.0 authentication
- **Leverage Unomi 4 features** while maintaining backward compatibility
- **Minimize downtime** during the migration process
- **Test both modes** with the integration test suite

Use this mode as a temporary solution during your migration journey, and plan to disable it once all clients are updated to use 4.0 authentication.

9.5. From version 3.0 to 4.0

9.6. Migration Overview

Apache Unomi 4.0 introduces comprehensive multi-tenancy support, enabling complete data isolation between different tenants. This fundamental architectural change requires a new tenant-based authentication model while keeping all API endpoints unchanged.

The key innovation in 4.0 is the introduction of **tenant isolation** for all data: - **Profiles, events, segments, rules, and schemas** are now tenant-specific - **Complete data separation** between tenants - no cross-tenant data access - **Tenant-specific API keys** for secure access control - **Backward compatibility** with system administrator access for management operations

9.7. Updating applications consuming Unomi

9.7.1. Authentication Model Changes

The main change in 4.0 is the introduction of tenant-based authentication. The system must now identify which tenant context to operate in for every request.

Aspect	Unomi 3.0	Unomi 4.0
Authentication Method	System Administrator Authentication (configured admin password)	Tenant-based API Keys + System Administrator Authentication
Public API Endpoints	No authentication required	Public API Key required (via X-Unomi-Api-Key header)
Private API Endpoints	System Administrator Authentication	Tenant Authentication (tenantId/privateKey) OR System Administrator Authentication
Tenant Administration	System Administrator Authentication	System Administrator Authentication (configured UNOMI_ROOT_PASSWORD)

9.7.2. API Key Types (4.0 Only)

4.0 introduces two types of API keys per tenant:

- **Public Key**: Used for public endpoints (event collection via /context.json)
- **Private Key**: Used with tenantId for tenant-specific administrative operations

9.7.3. Authentication Requirements by Endpoint Type

Endpoint Category	3.0 Authentication	4.0 Authentication
Event Collection (/context.json)	None	Tenant public API key (or tenant private key / System Admin)
Administrative Operations	System Admin (karaf:$UNOMI_ROOT_PASSWORD)	Tenant Auth (tenantId/privateKey) OR System Admin (karaf:$UNOMI_ROOT_PASSWORD)
Tenant Administration (/cxs/tenants)	System Admin (karaf:$UNOMI_ROOT_PASSWORD)	System Admin (karaf:$UNOMI_ROOT_PASSWORD)

9.7.4. Authentication Flow (4.0)

The AuthenticationFilter in 4.0 follows this resolution order:

1. **Tenant endpoints** (/cxs/tenants): Requires system administrator authentication only
2. **Public endpoints** (e.g., /context.json): Requires public API key via X-Unomi-Api-Key header
3. **Private endpoints**: Tries tenant authentication first, then falls back to system administrator authentication:

- **Tenant Authentication**: Basic Auth with tenantId:privateKey
- **System Administrator Authentication**: Basic Auth with karaf:$UNOMI_ROOT_PASSWORD (or configured admin credentials)

9.7.5. Code Examples

3.0 Authentication

```
// Global system administrator authentication for all endpoints
RestAssured.authentication = RestAssured.preemptive()
    .basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"));

// Context requests require no authentication
RestAssured.given()
    .auth().none()
    .contentType(ContentType.JSON)
    .body(contextJson)
    .post("/context.json");
```

4.0 Authentication

```
// For public endpoints (event collection)
given()
    .header("X-Unomi-Api-Key", publicKey)
    .contentType(ContentType.JSON)
    .body(contextJson)
    .post("/context.json");

// For private endpoints using tenant authentication
given()
    .auth().preemptive().basic(tenantId, privateKey)
    .contentType(ContentType.JSON)
    .body(payload)
    .post("/cxs/profiles");

// For private endpoints using system administrator authentication.
// Without X-Unomi-Tenant-Id, Unomi uses the admin default tenant (default "default", which is
// also the migration's default tenant id; see the note below if you chose another one).
// To work in the system context, send .header("X-Unomi-Tenant-Id", "system") explicitly.
given()
    .auth().preemptive().basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"))
    .contentType(ContentType.JSON)
    .body(payload)
    .post("/cxs/profiles");

// Same call scoped to a named tenant
given()
    .auth().preemptive().basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"))
    .header("X-Unomi-Tenant-Id", "default")
    .contentType(ContentType.JSON)
    .body(payload)
    .post("/cxs/profiles");

// For tenant administration (system admin only). No tenant header is needed:
// these calls run in the system context, even before any tenant exists.
given()
    .auth().preemptive().basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"))
    .contentType(ContentType.JSON)
    .body(tenantPayload)
    .post("/cxs/tenants");
```



JAAS admin calls without X-Unomi-Tenant-Id now act on the admin default tenant. If your migration used another tenant id than default, set org.apache.unomi.rest.authentication.defaultTenantId to the same value. Definitions that every tenant must inherit (for example shared rules or segments) are written with X-Unomi-Tenant-Id: system.

9.7.6. Implementation Strategy

Client Factory Pattern

```
public class UnomiConfiguration {
    public UnomiClient createClient(String baseUrl) {
        String version = System.getProperty("unomi.version", "4.0");

        if ("4.0".equals(version)) {
            return new UnomiV40Client(baseUrl);
        } else {
            return new UnomiV30Client(baseUrl);
        }
    }
}
```

Version-Specific Authentication

```
// 3.0 Client
public void init() {
    RestAssured.baseURI = baseUrl;
    RestAssured.authentication = RestAssured.preemptive()
        .basic("karaf", System.getenv("UNOMI_ROOT_PASSWORD"));
}

// 4.0 Client
public void init() {
    RestAssured.baseURI = baseUrl;
}

public void updateKeys(String publicKey, String privateKey) {
    this.publicKey = publicKey;
    this.privateKey = privateKey;
}
```

9.7.7. API paths stay the same; behaviour does not

API paths stay the same between 3.0 and 4.0, and most request/response payloads keep the same shape. Authentication, tenant resolution, and which identity fields a public caller may set on /context.json and /eventcollector do change — see Client-facing hardening (4.0) below. Other breaking changes are listed in Breaking changes.

9.7.8. Client-facing hardening (4.0)

/cxs/context.json and /cxs/eventcollector need a tenant **public** API key (or a tenant private key or administrator credentials); without one Unomi answers 401, unless single-tenant compatibility mode is enabled. In 4.0 the identity fields those endpoints accept are treated as claims to be checked rather than as instructions.

A caller is **trusted** when it holds a tenant private key or authenticates as a system administrator (ROLE_UNOMI_ADMIN or ROLE_UNOMI_TENANT_ADMIN). A tenant **public** API key is **not** trusted: it identifies the tenant, not the visitor.

Table 2. What changed for public callers

Field	3.0	4.0
Body/query profileId	Selected the profile.	Ignored. The profile cookie (context-profile-id by default) is the only profile bearer, including when no cookie is present. Trusted callers may still pass it, and it may differ from the cookie.
sessionId	Any existing session was continued, and was rebound to the caller's profile when the two differed.	Continued only when the profile cookie already owns that session. Otherwise the session is **detached** for the request rather than rebound, and its id is not echoed back in the response.
sessionId naming an **anonymous** session	Rebound to the caller's profile when the caller was not itself browsing anonymously.	Left anonymous. An anonymous session records no owner, so there is nothing to check the cookie against; only a trusted caller may bind it to a named profile. A visitor who turns anonymous browsing off keeps an anonymous session until their client uses a new session id.
invalidateSession=true	Re-created the session under the supplied id.	Allowed only when the profile cookie owns that session. A session with no recorded owner — an anonymous one — cannot be re-created this way by a public caller.

Anonymous browsing, personas and profile overrides are otherwise unchanged, and a caller with no cookie is still issued a profile: that is how tracking has always started.

Two behaviour changes to check before upgrading

- **The profile cookie now defaults to HttpOnly.** The model above holds only while page script cannot read the cookie, so the default is part of the design rather than a preference. If your page code reads document.cookie for the profile id, read profileId from the JSON response instead. Setting org.apache.unomi.profile.cookie.httpOnly=false (env UNOMI_PROFILE_COOKIE_HTTPONLY) restores the old default, at the cost of that guarantee.
- **A malformed profile cookie is rejected on more requests than before.** The cookie is validated against a JSON schema and an invalid value answers 400. In 3.0 the cookie was only read when no profileId was supplied, so a request carrying both an explicit profileId and a malformed cookie was still served; in 4.0 it fails. This mainly affects server-side callers that forward a browser's raw Cookie header.

Migrating a client

- Browser trackers need no change for the profile cookie: the browser sends it, and the response still carries profileId and sessionId. They do need the tenant's public API key once you leave compatibility mode. The Unomi web tracker sends that key from version 2.0.0. Tracker 1.x cannot. See Upgrading the tracker from 1.x to 2.0 and Configuring the tracker for your Unomi server.

- A backend that named a profile through body profileId while authenticating with a **public** key must switch to a tenant private key, or to system administrator credentials. Until it does, the server logs a WARN naming the ignored profileId on every such request.
- A client that supplied a session id it did not own now finds sessionId absent from the response. Treat that as "start a new session" and generate a fresh id rather than replaying the old one.

9.7.9. A search condition states its type (4.0)

A search body carries a condition, and 4.0 resolves that condition by its type before it runs the query. A condition with no resolvable type answers an empty list, and the server logs the reason:

```
WARN ParserHelper      | Condition has no type ID for profile search
WARN ProfileServiceImpl | Cannot execute query: condition type 'null' could not be resolved
```

3.0 answered the same body by running the query with no condition at all, which returned every item. The two answers are opposite, and neither the request nor the response says which one you got.

Table 3. What changed

Body	3.0	4.0
"condition": {}	The condition was dropped, and the search returned every item.	The search returns an empty list.
"condition" naming a type the server does not know	The condition was dropped, and the search returned every item.	The search returns an empty list.

This affects every endpoint that takes a Query, among them /cxs/profiles/search, /cxs/profiles/search/sessions and /cxs/profiles/personas/search.

Migrating a client

- A client that searched over everything by sending an empty condition states matchAllCondition instead:

```
{ "text": "", "offset": 0, "limit": 1000,
  "condition": { "type": "matchAllCondition", "parameterValues": {} } }
```

- A client that omits the condition field entirely still searches over everything, and it needs no change.
- A client whose condition names a type that the server does not know now reads an empty list where it used to read every item. Check the server log for the condition type it names, and deploy the plugin that defines it.

9.7.10. An unset optional parameter no longer makes an OR match (4.0)

A condition parameter that reads parameter::name is optional: when nothing supplies name, the branch that holds it states no constraint and is left out. 3.0 left it out of an AND, and 4.0 now also leaves it out of an OR instead of counting it as a match. A hoverEventCondition that sets only targetId, for example, used to match every hover event, and now matches only hovers on that target. That condition comes from the hover-event plugin, which is not deployed by default.

Table 4. What changed

Condition	3.0	4.0
An OR with a branch whose parameter is unset	Matched every item.	The branch is left out, and the OR matches on its other branches.
An OR whose branches are all unset	Matched every item.	Matches no item.

An OR whose subConditions list is empty is unchanged: it matches every item when run as a search query, and no item when evaluated in memory, as a rule is.

Migrating a client

- An OR of optional parameters needs at least one of them set to match anything.
- An AND whose branches are all unset still states no constraint and matches every item, as in 3.0, also when it sits inside an OR.

9.7.11. Segment and scoring lists merge system and tenant items (4.0)

A tenant sees its own segments and scoring plans together with those of the system tenant, and a tenant item replaces a system item with the same id. Because the two sets are merged after they are read, 4.0 sorts and pages the segment and scoring lists itself instead of leaving it to the search engine. This applies to GET /cxs/segments, POST /cxs/segments/query, GET /cxs/scoring, POST /cxs/scoring/query and the GraphQL findSegments query.

Paging and sorting follow the 3.0 rules:

- offset and limit page the merged list. A missing limit returns defaultQueryLimit items, 10 unless configured (see the search engine configuration), and -1 returns every item. The response echoes the requested offset and limit.
- sortby takes the same syntax: comma-separated field or field:desc criteria. Any criterion other than an exact :desc suffix sorts ascending.
- A field with no value sorts last in both directions. A field with several values, such as metadata.tags, sorts on its smallest value ascending and its largest value descending.
- A sort field the server does not know is ignored, as it was when the search engine ignored an unmapped field.

The sortable fields are metadata.id, metadata.name, metadata.description, metadata.scope, metadata.tags, metadata.systemTags, metadata.enabled, metadata.missingPlugins, metadata.hidden, metadata.readOnly, itemId, tenantId, createdBy, lastModifiedBy, creationDate, lastModificationDate and lastSyncDate. Each metadata.* field can also be named without its prefix, for example name.

Table 5. What changed

Request	3.0	4.0
limit is -1	Every item, with totalSize reported as 0.	Every item, with totalSize reported as the number of items.
Several items have the same value for the sort field	Their order was left to the search engine.	They are ordered by itemId, so pages are stable.
sortby names a field without its metadata. prefix, or an audit field such as creationDate	Ignored.	Sorted on.
A text value longer than 256 characters, in a sort	Sorted as a missing value, because the search engine does not index it for sorting.	Sorted on its full value.
A negative offset, or offset plus limit above 10,000	Rejected by the search engine.	A negative offset counts as 0, and large pages are returned.

Migrating a client

- A client that pages with limit and offset, or reads every item with -1, needs no change.
- A client that read totalSize after asking for -1 now gets the real count instead of 0.

9.8. Migrating your existing data

9.8.1. Multi-Tenancy Impact

When migrating to 4.0, you need to understand that:

- All data (profiles, events, segments, rules, schemas) becomes tenant-specific
- Each tenant operates in complete isolation with their own data space
- Tenant context must be established for every API operation

9.8.2. Recurrent import and export files move under a tenant directory

A recurrent import source or export destination that uses the file scheme must now resolve inside the deployment's base directory for that direction (org.apache.unomi.router.config.import.baseDir and

org.apache.unomi.router.config.export.baseDir, defaulting to ${karaf.data}/router/import/ and ${karaf.data}/router/export/), under the subdirectory named after the configuration's tenant ({baseDir}/{tenantId}). This applies to the endpoint directory, to its path-bearing options (fileName, move, moveFailed, …) and to localWorkDirectory on ftp, sftp and ftps endpoints.

Existing configurations are not migrated. After the upgrade, a configuration whose endpoint resolves anywhere else builds no route and is marked INVALID_ENDPOINT; one error line is logged per configuration, followed by a summary count at the end of each route build. To keep such a configuration running, either move its files under {baseDir}/{tenantId} and update its source or destination accordingly, or point the two base-directory properties at the directories already in use. The mark clears on its own once the endpoint is accepted again. The "Profile import and export" chapter describes the rule in full.

9.8.3. Background tasks and recurrent routes run for every tenant

The scheduled purge, the daily recalculation of date-relative segments and scoring plans, the rule statistics refresh and the start of recurrent import and export routes apply to every tenant, the tenant holding the migrated data included.

- The first purge after the upgrade applies the configured retention (org.apache.unomi.profile.purge.*, org.apache.unomi.session.purge.existTime, org.apache.unomi.event.purge.existTime, org.apache.unomi.monthly.index.purge.existTime) to the migrated tenant in one pass, shortly after startup. Review these values before starting 4.0 if the 3.0 deployment ran with the purge disabled or with a longer retention.
- A purge or recalculation that fails for one tenant is still run for the others; the task is then reported as failed, naming the tenants concerned, and the scheduler retries it.
- The Camel route of a recurrent import or export configuration is identified by {tenantId}:{configId} instead of the configuration identifier alone, since configuration identifiers are only unique within a tenant. Code that looks routes up in the router's Camel context must build the identifier with org.apache.unomi.router.api.RouteIds. The killExistingRoute, updateProfileImportReaderRoute and updateProfileExportReaderRoute methods of IRouterCamelContext take the tenant identifier as their first parameter. A tenant identifier containing a colon gets no recurrent route: the identifier given to org.apache.unomi.migration.tenant.id must follow the tenant identifier rules (letters, digits, hyphens and underscores).

9.8.4. Migration Steps

1. **Understand Multi-Tenancy Impact**
    - All data (profiles, events, segments, rules, schemas) becomes tenant-specific
    - Each tenant operates in complete isolation with their own data space
    - Tenant context must be established for every API operation
2. **Update Authentication Configuration**
    - Remove global system administrator authentication
    - Configure tenant-specific public and private API keys

- Implement endpoint-specific authentication logic

3. **Endpoint-Specific Changes**
   - Add X-Unomi-Api-Key header with public key for event collection
   - Use tenant authentication (tenantId/privateKey) for tenant-specific administrative operations
   - Keep system administrator authentication as fallback for administrative operations
   - Continue using system administrator authentication for tenant administration
4. **Related client contract changes (4.0 hardening)**
   - Public /context.json and /eventcollector no longer treat body/query profileId as identity — see below
   - Endpoint paths and most payload shapes remain the same; authentication and public profile-binding rules change

9.8.5. Client-facing hardening (profile cookie, passwords, privileged APIs)

In addition to tenant API keys, Unomi 4.0 hardens several contracts that can break older clients and scripts. See also How profile tracking works and UNOMI-972.

Why cookie-only for public profile identity? On public context endpoints the profile id is a bearer token. Accepting a client-chosen body/query profileId let any public caller try to continue another visitor's context if they obtained that UUID. The cookie is issued by Unomi, sent automatically by the browser, and (with default HttpOnly) is not readable from page script — so public clients prove continuity with a server-issued bearer instead of an application-supplied id. Trusted callers may still override with body profileId when a backend intentionally selects a profile. Full rationale: How profile tracking works (section *Why the cookie is safer…*).

Area	Old common pattern	Required in 4.0
Public profile identity	Body/query profileId often selected the profile (sometimes ahead of the cookie)	Public callers: **cookie only**. Body/query profileId is ignored. Trusted callers (tenant private key / system admin) may still override.
Session resume without cookie	Sending only sessionId could switch onto that session's profile	Public callers switch only if the cookie already owns the session; otherwise the session is detached for the request.

Area	Old common pattern	Required in 4.0
Tracker on another origin	Public endpoints answered any origin with credentialed CORS and never checked where a request came from. A page on another subdomain could use the visitor cookie. A website on another site never received it (the cookie was always SameSite=Lax), so it kept the profile id itself and sent it as body profileId.	Only origins listed in org.apache.unomi.public.cors.allowedOrigins (UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS), or in the requesting tenant's publicCorsAllowedOrigins property, get credentialed CORS and have the cookie used and set; other pages get a new anonymous profile and their cookie is left untouched. The default is empty. A website served from Unomi's own origin through a reverse proxy needs no change. One on another subdomain must be listed. One on another site must be listed and, since body profileId is now ignored for public callers, also needs HTTPS with SameSite=None (UNOMI_PROFILE_COOKIE_SAMESITE), or a move to the same origin. A 2.x client that embeds context.js with a <script> tag also needs org.apache.unomi.public.cors.matchReferer=true (and, from another site, HTTPS with SameSite=None). See Compatibility with Unomi 2.x and 3.x.
Admin UI on another origin	Admin REST and GraphQL answered any origin with credentialed CORS, so a page on another site could reuse the operator's browser credentials to read admin responses.	Only origins listed in org.apache.unomi.admin.cors.allowedOrigins (UNOMI_ADMIN_CORS_ALLOWED_ORIGINS) get credentialed CORS on admin REST and GraphQL. The default is empty. A same-origin admin UI needs no change. See Admin UI origins.
GraphQL query limits	GraphQL documents had no server-side size, depth, or complexity bound beyond container limits.	GraphQL under /graphql applies configurable length, body size, depth, and complexity limits (org.apache.unomi.graphql.*). Defaults leave GraphiQL introspection and the public getProfile / processEvents fields usable. See GraphQL query limits.

Area	Old common pattern	Required in 4.0
GraphQL over HTTP	GET /graphql ran any operation, mutations included, and POST /graphql parsed a JSON body whatever its Content-Type.	GET runs queries only (405 for mutations, subscriptions, and getProfile with createIfMissing: true). POST requires Content-Type: application/json (415 otherwise); a charset parameter is fine. Clients that send JSON without that header must add it: for example curl -d alone sends application/x-www-form-urlencoded. See GraphQL query limits.
Profile cookie HttpOnly	Often defaulted to false (JS could read document.cookie)	Default **true**. Prefer profileId from the JSON response. Opt out with org.apache.unomi.profile.cookie.httpOnly=false only if you accept the tradeoff.
Admin / health passwords	Known defaults such as karaf / karaf were common in samples	Set UNOMI_ROOT_PASSWORD and UNOMI_HEALTHCHECK_PASSWORD before start (no shipped known defaults).
Protected events from public callers	Accepted only from a configured third-party server (X-Unomi-Peer key and IP)	Accepted only from an IP in the tenant's authorizedIPs; an empty list refuses them. Tenant private key and administrator callers are not IP-checked. See Multi-tenancy.
Groovy Actions / Router import-export REST	Often callable with tenant private key	Requires **system administrator** (ADMINISTRATOR), not tenant administrator.
Startup without a password	Started with the shipped default	bin/karaf and the Docker entrypoint refuse to start. On Windows startup continues — see the warning below.

Area	Old common pattern	Required in 4.0
mergeProfilesOnProperty / cross-profile updateProperties / systemProperties.*	Sometimes driven from public context events	Cross-profile merge/update and systemProperties writes require a trusted caller (system or tenant admin); this includes recording the merge identifier on the current profile, so a public login event no longer claims an identity at all. updateProperties now applies an allowlist to property names: public callers may only write plain dotted properties.* paths on the **current** cookie-bound profile; systemProperties, segments, scores and consents need a trusted caller; identity fields (itemId, mergedWith, …) and beanutils mapped/indexed syntax are rejected for everyone.



Windows: the startup check cannot stop the launcher.

On Linux and macOS, bin/karaf sources bin/setenv directly, so a missing password aborts startup. The Docker entrypoint behaves the same way and the container exits.

On Windows, karaf.bat runs setenv.bat with call and does not test the exit code, so the error message is printed and **startup continues anyway**. This is a limitation of the Karaf launcher that Unomi cannot work around without replacing karaf.bat.

This matters because an unset password is not the same as a disabled account. ${env:UNOMI_ROOT_PASSWORD} resolves to the **empty string**, so etc/users.properties creates the karaf administrator with an empty password that authenticates successfully.

Windows operators must therefore treat the message as fatal and verify the password took effect before exposing the instance — for example by confirming that an empty password is rejected:

```
# Must return 401. A 200 means the account has a blank password.
curl -i -u "karaf:" http://localhost:8181/cxs/tenants
```

Unomi additionally refuses any REST call presenting Basic authentication with an empty password, so the admin API is not reachable that way even if the server did start unconfigured. The Karaf SSH console (port 8102) is not covered by that check.

Migration checklist for client applications

- Unomi web tracker: use version 2.0.0 or later. On a multi-tenant server, set apiKey to the tenant's public API key. Tracker 1.x cannot send that key, so leave single-tenant compatibility mode on until the page is updated. The same configuration still works on Unomi 2.x, 3.x, and on Unomi 4 in that mode. After the upgrade, clicks that 1.x missed are recorded, and the page's own click handlers run once per click. See Upgrading the tracker from 1.x to 2.0.
- Browser / tracker clients: ensure cookies are sent (withCredentials / same-site setup); list every website origin that calls Unomi from another origin in UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS or in its tenant's publicCorsAllowedOrigins property (and, for another site, use HTTPS with SameSite=None, or serve Unomi from the website's origin); stop treating body profileId as authoritative for public calls.
- Headless or mobile public clients that stored a UUID and posted it only in the body: switch to sending the profile cookie, or call with a **private key** when you intentionally bind a profile.
- Login / merge flows: emit merge-triggering events from a trusted server-side caller after real authentication — not from the public key alone.
- Ops scripts and Docker: export both password env vars; replace karaf:karaf with karaf:$UNOMI_ROOT_PASSWORD.
- Windows deployments: confirm the passwords actually took effect after upgrading — the startup check warns but cannot halt karaf.bat (see the warning above).
- Automation that uploaded Groovy actions or managed Router import/export with a tenant private key: switch to system administrator credentials.

9.8.6. Benefits of Multi-Tenancy in 4.0

- **Data Isolation**: Complete separation ensures tenant data never crosses boundaries
- **Scalability**: Support for multiple customers/organizations in a single Unomi instance
- **Security**: Tenant-specific API keys prevent unauthorized cross-tenant access
- **Compliance**: Easier to meet data privacy regulations with clear tenant boundaries
- **Cost Efficiency**: Shared infrastructure with isolated data reduces operational costs

9.9. Breaking changes

The sections above cover authentication and client-facing hardening. The lists below gather other changes that can break an existing 3.0 installation, integration, or plugin. Related tickets: UNOMI-1008 (admin default tenant), UNOMI-1005 (public cookie origins), UNOMI-1007 (health check), UNOMI-1001 (third-party peers in compatibility mode), UNOMI-1012 (compatibility mode for Unomi 2 and 3 clients).

9.9.1. Operators

- **Shell commands** such as profile-list, rule-list, and segment-view were removed. Use unomi:crud instead (for example crud list profile, crud read rule RULE_ID). See Shell commands.

- **Document ids** in the search index now start with the tenant id. Scripts or dashboards that read ids directly must be updated.
- **GraphQL** is enabled through the GraphQL distributions (unomi-distribution-elasticsearch-graphql or unomi-distribution-opensearch-graphql) or the cdp-graphql-feature feature, not the older 3.0 wiring alone. See GraphQL API.
- **Health check password**: UNOMI_HEALTHCHECK_PASSWORD no longer defaults to health and must be set before start (like UNOMI_ROOT_PASSWORD). The other UNOMI_HEALTHCHECK_* variables keep their 3.0 names and defaults. See Configuration.
- **Third-party / peer servers**: org.apache.unomi.thirdparty.* and the X-Unomi-Peer header apply only in single-tenant compatibility mode. In normal multi-tenant mode, protected events are checked against each tenant's authorizedIPs (plus any extra types in restrictedEventTypes); see Secured events configuration.
- **Migration key file**: migrate-4.0.0-10-tenantInitialization writes API keys under {karaf.data}/migration/secrets/. The public key goes in the X-Unomi-Api-Key header; the private key is used with HTTP Basic authentication as tenantId:privateKey. Store the file securely and delete it after the keys are saved elsewhere.

9.9.2. Integrators / REST clients

- Creating a profile without an itemId is refused.
- Merging profiles adds the points of properties.interests by key. Previously the last profile's list replaced the others. Values that are not numbers are kept unchanged. See Property Types.
- interests is now a built-in profile property type of type json, so GraphQL lists it as an interests field on CDP_Profile, CDP_Persona, CDP_PersonaInput and CDP_ProfileUpdateEventInput. GraphQL no longer generates string filters (_equals, _startsWith, ...) for json properties, since they could not match a JSON value.
- Condition validation failures on REST return a generic badRequest response. Detailed messages stay in the server log. See Condition validation.
- Public endpoints need a tenant public API key unless single-tenant compatibility mode is on (for clients written for Unomi 2 or Unomi 3). See Updating applications consuming Unomi and the client-facing hardening notes above.
- The Unomi web tracker 1.x cannot call a multi-tenant server. Use tracker 2.0.0 or later and set apiKey, or keep compatibility mode on. The script served at /tracker/ is updated with the server. An npm install or a copied file is not. See Upgrading the tracker from 1.x to 2.0.
- Websites on another origin must be listed for the visitor cookie; see Websites on other origins.

9.9.3. Plugin developers

- The **task scheduler** is a new service surface (SchedulerService and related APIs). Plugins that scheduled work their own way should move to it when they need cluster-aware tasks. See Task scheduler.
- The pastEventCondition and sourceEventPropertyCondition evaluators moved from the base plugin to the advanced-conditions plugin (package

org.apache.unomi.plugins.advancedconditions.conditions). Condition type ids are unchanged, so JSON definitions keep working; only Java code that references the evaluator classes must be updated.

- **Query builder ids** were renamed. Legacy *ESQueryBuilder ids in condition definitions are still mapped to the new ids at run time, with a warning in the log, and migrate-4.0.0-15-updateLegacyQueryBuilder rewrites the stored ones. Update plugin definitions to the new ids to remove the warning.
- Event, definitions, and profile **Java service** method signatures and condition parameter default-value behaviour may differ from 3.0. Rebuild plugins against the 4.0 API and run your tests.

9.10. Migration Checklist

Before starting the migration, please ensure that:

- You do have a backup of your data
- You did practice the migration in a staging environment, NEVER migrate a production environment without prior validation
- You are currently running Apache Unomi 3.0 (or a later 3.0.x version)
- You understand the multi-tenancy impact on your data model
- You have a plan to update client applications to tenant API keys (or temporary single-tenant compatibility mode for Unomi 2 or Unomi 3 clients)
- You have a plan for the web tracker: version 2.0.0 or later with apiKey on a multi-tenant server, or compatibility mode while tracker 1.x is still on the page (see Upgrading the tracker from 1.x to 2.0)
- You have reviewed the breaking changes and client-facing hardening notes
- You know how to obtain plaintext API keys after upgrade (regenerate via /cxs/tenants/{id}/apikeys; create responses expose masked keys only; the migration also writes a key file under data/migration/secrets/)
- Your recurrent import and export file endpoints resolve under {baseDir}/{tenantId}, or the router's base-directory properties point at the directories already in use (see above)

9.11. Migration Process

Upgrading from 3.0 to 4.0 requires a **data migration** (bundled Groovy scripts) plus client authentication updates. Contributors maintaining those scripts should follow Writing migration scripts (step tracking, idempotency, and common pitfalls).

1. **Shut down your Apache Unomi 3.0 cluster** (migration runs with Unomi stopped; the search engine must remain reachable).
2. **Install Apache Unomi 4.0** on the node(s) that will run the migration (Karaf shell available).
3. **Run the migration** from the Karaf shell:

```
unomi:migrate 3.0.0
```

Pass your actual origin version without qualifier (for example 3.0.0). Optional second argument skips the confirmation prompt. You can preconfigure etc/org.apache.unomi.migration.cfg instead of answering interactive questions; see Shell commands.

Scripts applied for the 4.0 line include (under tools/shell-commands/.../META-INF/cxs/migration/):

- migrate-4.0.0-00-fixProfileNbOfVisits
- migrate-4.0.0-01-tenantDocumentIds
- migrate-4.0.0-05-fixSystemItemIds
- migrate-4.0.0-10-tenantInitialization (creates the default tenant used for isolation)
- migrate-4.0.0-15-updateLegacyQueryBuilder

4. **Update client applications** to the 4.0 authentication model (public key header / tenant private key), or enable single-tenant compatibility mode while Unomi 2 or Unomi 3 clients are still in use.
5. **Start your Apache Unomi 4.0 cluster**.
6. **Store tenant API keys** from the migration key file under data/migration/secrets/ (or regenerate via /cxs/tenants/{id}/apikeys), then **test** applications end to end.

9.12. Transitional authentication options

Use one of the following approaches while migrating clients:

9.12.1. Option 1: Update clients to 4.0 authentication (recommended)

Adopt tenant public API keys for public endpoints and tenant private keys or JAAS for administrative work. See Multi-tenancy and the authentication examples in this guide. For the Unomi web tracker, upgrade to 2.0.0 or later and set apiKey. See Upgrading the tracker from 1.x to 2.0.

9.12.2. Option 2: single-tenant compatibility mode (Unomi 2 and Unomi 3 clients)

If clients were written for **Unomi 2.x or Unomi 3.x**, you can enable single-tenant compatibility mode so they work without tenant API keys during a phased rollout:

```
# etc/org.apache.unomi.rest.authentication.cfg
singletenantcompatibility.enabled = true
```

This mode always uses tenant default. There is no property to point it at another tenant: defaultTenantId does not apply to it yet, so leave that setting at default while the mode is on (GraphQL still follows it). If tenant default does not exist yet, the first public request creates it. Migrated data has to live in that tenant to be visible while the mode is on.

Environment variable equivalent: UNOMI_REST_AUTHENTICATION_SINGLETENANTCOMPATIBILITYMODEENABLED=true



single-tenant compatibility mode is a temporary bridge for Unomi 2 and Unomi 3 clients. It bypasses tenant API key requirements on public endpoints. Disable it once all clients use 4.0 authentication.

9.13. Post Migration

Once the migration has been completed, you will be able to start Apache Unomi 4.0 with full multi-tenancy support.

Remember that all data operations now require proper tenant context, either through tenant authentication or system administrator authentication.

The fundamental difference between Unomi 3.0 and 4.0 is the introduction of **comprehensive multi-tenancy support**:

- **3.0**: Single-tenant architecture with system administrator authentication for all operations
- **4.0**: Multi-tenant architecture with complete data isolation and tenant-specific authentication
- **API Endpoints**: Paths remain the same; authentication and public profile-binding rules change (see client-facing hardening above)
- **Data Model**: All entities (profiles, events, segments, rules, schemas) become tenant-specific in 4.0
- **Authentication**: New tenant-based authentication model with system administrator authentication as fallback

The authentication changes in 4.0 are driven by the need to establish tenant context for every operation, ensuring complete data isolation while maintaining backward compatibility for administrative operations.

9.13.1. Migrating from Elasticsearch to OpenSearch

Apache Unomi 4.0+ can use OpenSearch instead of Elasticsearch as its search/persistence backend. Moving existing data is a **search-engine cutover**: copy Unomi indices from Elasticsearch to OpenSearch, then point Unomi at the OpenSearch distribution. It is **not** performed by unomi:migrate (that command upgrades Unomi data shape between Unomi versions).

This page only covers Unomi-specific index naming and cutover steps. Use the official OpenSearch documentation for how to copy cluster data; tooling and supported version matrices change over time.

Prerequisites

- Apache Unomi already running against Elasticsearch (typically Elasticsearch 9.x for Unomi 3.x)
- A target OpenSearch cluster at the version Unomi expects (see opensearch.version in the Unomi root POM; currently **3.7.0**)

- Network path between the clusters (or shared snapshot storage, depending on the method you choose)
- Backup of the Elasticsearch indices, and a staging rehearsal before production
- Plan a write freeze (stop or quiesce Unomi) for the final cutover so source and target do not diverge

Unomi index naming

By default Unomi stores data under the index prefix context (org.apache.unomi.elasticsearch.index.prefix / org.apache.unomi.opensearch.index.prefix).

Examples:

- context-profile, context-segment, context-rule, ...
- Rollover types: context-event-, context-session-, ...

List what you must copy:

```
curl -s "http://SOURCE:9200/_cat/indices/context-*?v"
```

If you customized the prefix, replace context with your value everywhere below, and keep the **same** prefix on the OpenSearch side after cutover.

Also migrate **aliases** and any ILM/ISM (or equivalent) policies that Unomi relies on for rollover indices. Official migration tooling does not always move policies automatically; check the tool's component support list.

Choose a data-copy method (external docs)

Pick a method that matches your Elasticsearch/OpenSearch versions, downtime budget, and ops stack. Always confirm the current **supported source target matrix** before you start.

Approach	Where to start
Migration Assistant (metadata, snapshot backfill / Reindex-from-Snapshot, optional live Capture and Replay)	Migration Assistant for OpenSearch · Is it right for you? (version matrix) · opensearch-migrations on GitHub
Remote reindex into OpenSearch (copy documents from a remote Elasticsearch cluster)	Reindex data · Reindex API (create destination indices with mappings/settings first; remote host must be allowlisted)
Logstash (scroll from Elasticsearch, ship to OpenSearch)	Logstash and OpenSearch · Ship events to OpenSearch (logstash-output-opensearch plugin)



For a self-managed Unomi deployment with planned downtime, **remote reindex** is often the simplest path once destination mappings exist. For large or near-zero-downtime migrations, evaluate **Migration Assistant** against the published matrix for your Elasticsearch major version (Unomi 3.x commonly uses Elasticsearch **9.x**; confirm support in the upstream docs).

Verify the copy

Before switching Unomi:

- Compare _cat/indices document counts for every Unomi index (and rollover generations)
- Confirm aliases used by rollover types still resolve
- Spot-check a few documents by id

```
curl -s "http://SOURCE:9200/_cat/indices/context-*?v&h=index,docs.count"
curl -s "https://TARGET:9200/_cat/indices/context-*?v&h=index,docs.count"
```

(Adjust scheme, auth, and TLS as required by each cluster.)

Point Unomi at OpenSearch

1. Stop Apache Unomi (or otherwise freeze writes).
2. Switch the runtime distribution to OpenSearch:

   ```
   # Docker / environment
   UNOMI_DISTRIBUTION=unomi-distribution-opensearch

   # Or from the Karaf shell before starting Unomi features
   unomi:setup -d=unomi-distribution-opensearch -f=true
   ```

3. Point Unomi at the target cluster in etc/custom.system.properties (or unomi.custom.system.properties / env vars). Example:

   ```
   # org.apache.unomi.elasticsearch.*  (disable / stop using)

   org.apache.unomi.opensearch.addresses=localhost:9200
   org.apache.unomi.opensearch.cluster.name=opensearch-cluster
   org.apache.unomi.opensearch.index.prefix=context
   org.apache.unomi.opensearch.username=admin
   org.apache.unomi.opensearch.password=${env:OPENSEARCH_INITIAL_ADMIN_PASSWORD:-
   admin}
   org.apache.unomi.opensearch.sslEnable=true
   org.apache.unomi.opensearch.sslTrustAllCertificates=true
   ```

   Align sslEnable / trust settings with your OpenSearch security setup. See Configuration.

4. Start Unomi and verify health, tenant/API access, and a representative profile/event flow.

Keep the Elasticsearch cluster until you are satisfied the OpenSearch deployment is correct, then decommission it.

Further help

- Apache Unomi community: https://unomi.apache.org/community.html
- OpenSearch forum: https://forum.opensearch.org/

9.14. From version 2.x to 3.0

9.14.1. Migration Overview

Apache Unomi 3.0 is a major release that introduces several breaking changes. This document details the steps required to successfully migrate your environment from Apache Unomi 2.x to Apache Unomi 3.0.

9.14.2. Prerequisites

Before migrating to Apache Unomi 3.0, ensure you meet the following requirements:

Java Version Requirements

Apache Unomi 3.0 requires **Java 17 or later**. This is a breaking change from previous versions that supported Java 11.

- **Minimum Java Version**: Java 17
- **Recommended**: Java 17 LTS
- **Tested Versions**: Java 17 (most tested), Java 21

Search Engine Compatibility

Apache Unomi 3.0 supports the following search engine versions:

- **Elasticsearch**: Version 9.x (upgraded from 7.17.5)
- **OpenSearch**: Version 3.x (new support)



OpenSearch support is introduced in Apache Unomi 4.0. If you plan to use OpenSearch, first migrate to 3.0 and then follow the 3.0 4.0 migration guide.

Karaf Version Upgrade

Apache Unomi 3.0 includes an upgrade of the underlying Karaf framework:

- **Previous Version**: Karaf 4.2.15
- **New Version**: Karaf 4.4.11

This upgrade brings improved stability and support for the latest dependencies.

9.14.3. Elasticsearch Client Upgrade

Apache Unomi 3.0 includes a major upgrade to the Elasticsearch client:

- **Previous**: REST client (deprecated and no longer supported)
- **New**: Official Elasticsearch Java client

This change provides: - Better performance and reliability - Full compatibility with Elasticsearch 9.x - Improved error handling and connection management - Future-proof client that follows Elasticsearch's official client roadmap

The new client is automatically used when connecting to Elasticsearch - no configuration changes are required.

9.14.4. QueryBuilder ID Changes

In Apache Unomi 3.0, all queryBuilder IDs have been renamed to remove ElasticSearch-specific references and make them more generic. This change affects custom condition definitions that reference built-in query builders.

Impact on Custom Condition Definitions

If you have custom condition definitions that reference built-in query builders using the queryBuilderId parameter, you will need to update these references to use the new queryBuilder IDs.

QueryBuilder ID Mapping Table

The following table shows the mapping between old and new queryBuilder IDs:

Condition Type	Old QueryBuilder ID	New QueryBuilder ID
IDs Condition	idsConditionESQueryBuilder	idsConditionQueryBuilder
Geo Location by Point Session Condition	geoLocationByPointSessionConditionESQueryBuilder	geoLocationByPointSessionConditionQueryBuilder
Past Event Condition	pastEventConditionESQueryBuilder	pastEventConditionQueryBuilder
Boolean Condition	booleanConditionESQueryBuilder	booleanConditionQueryBuilder
Not Condition	notConditionESQueryBuilder	notConditionQueryBuilder
Match All Condition	matchAllConditionESQueryBuilder	matchAllConditionQueryBuilder

Condition Type	Old QueryBuilder ID	New QueryBuilder ID
Property Condition	propertyConditionESQueryBuilder	propertyConditionQueryBuilder
Source Event Property Condition	sourceEventPropertyConditionESQueryBuilder	sourceEventPropertyConditionQueryBuilder
Nested Condition	nestedConditionESQueryBuilder	nestedConditionQueryBuilder

Migration Steps

1. **Identify Custom Condition Definitions**: Review your custom condition definitions to find any that reference the old queryBuilder IDs listed in the table above.
2. **Update QueryBuilder References**: Replace the old queryBuilder IDs with the new ones according to the mapping table.
3. **Test Your Conditions**: After updating the references, test your custom conditions to ensure they continue to work as expected.

Example Migration

Before (Apache Unomi 2.x):

```
{
  "id": "customIdsCondition",
  "name": "Custom IDs Condition",
  "conditionType": "idsCondition",
  "queryBuilderId": "idsConditionESQueryBuilder",
  "parameters": [
    {
      "id": "ids",
      "type": "string",
      "multivalued": true
    }
  ]
}
```

After (Apache Unomi 3.0):

```
{
 "id": "customIdsCondition",
 "name": "Custom IDs Condition",
 "conditionType": "idsCondition",
 "queryBuilderId": "idsConditionQueryBuilder",
 "parameters": [
  {
   "id": "ids",
   "type": "string",
   "multivalued": true
  }
 ]
}
```

Important Notes

- This change affects both ElasticSearch and OpenSearch persistence implementations
- The functionality of the query builders remains unchanged; only the IDs have been renamed
- Built-in condition definitions provided by Apache Unomi have been automatically updated
- If you are using only built-in conditions without custom queryBuilder references, no migration is required

Backward Compatibility

Apache Unomi 3.0 includes a configurable mapping system that provides backward compatibility for legacy queryBuilder IDs. This means that existing custom condition definitions using the old queryBuilder IDs will continue to work without modification.

The mapping system automatically translates legacy queryBuilder IDs to their new equivalents:

- **Automatic Translation**: Legacy IDs are automatically mapped to new IDs at runtime
- **Optimized Performance**: New queryBuilder IDs are checked first, legacy mapping lookup only occurs when needed
- **Warning System**: Legacy ID usage triggers warning logs with specific condition type information
- **Deprecation Notices**: Clear warnings indicate that legacy mappings are deprecated

Built-in Mappings: The legacy mappings for built-in queryBuilder IDs are provided as hardcoded defaults in the dispatcher implementations (for both Elasticsearch and OpenSearch). These defaults cannot be modified at runtime:

- idsConditionESQueryBuilder idsConditionQueryBuilder
- geoLocationByPointSessionConditionESQueryBuilder geoLocationByPointSessionConditionQueryBuilder
- pastEventConditionESQueryBuilder pastEventConditionQueryBuilder

- booleanConditionESQueryBuilder booleanConditionQueryBuilder
- notConditionESQueryBuilder notConditionQueryBuilder
- matchAllConditionESQueryBuilder matchAllConditionQueryBuilder
- propertyConditionESQueryBuilder propertyConditionQueryBuilder
- sourceEventPropertyConditionESQueryBuilder sourceEventPropertyConditionQueryBuilder
- nestedConditionESQueryBuilder nestedConditionQueryBuilder

For custom query builder implementations, migrate to the new naming convention and provide both Elasticsearch and OpenSearch implementations as documented in the plugin guide.

IMPORTANT WARNING FOR PLUGIN DEVELOPERS:

If you have custom query builder implementations, you should **NOT** rely on legacy mappings for your custom query builders. Instead, you should:

1. **Rename your custom query builders** to follow the new naming convention (remove "ES" references, use generic "QueryBuilder" suffix)
2. **Provide separate implementations** for Elasticsearch and OpenSearch:
    - Create separate bundles for Elasticsearch (ConditionESQueryBuilder) and OpenSearch (ConditionOSQueryBuilder) implementations
    - Use separate Karaf features for each implementation
    - Configure the appropriate feature in your start configuration based on the selected search engine
3. **Update your condition type definitions** to use the new query builder IDs
4. **Use proper OSGi service registration** to ensure your query builders are only active when the corresponding search engine is in use

This approach ensures: - **Better maintainability**: No dependency on legacy mapping system - **Search engine compatibility**: Proper support for both Elasticsearch and OpenSearch - **Cleaner architecture**: Separate concerns for different search engines - **Future-proof**: No risk of legacy mapping removal affecting your plugins

Warning System: When legacy queryBuilder IDs are used, Apache Unomi will log warning messages that include: - The legacy queryBuilder ID being used - The specific condition type that needs to be updated - The new queryBuilder ID that should be used - A deprecation notice indicating that legacy mappings may be removed in future versions

Example Warning Log:

```
WARN - DEPRECATED: Using legacy queryBuilderId 'idsConditionESQueryBuilder' for condition type
'customIdsCondition'.
Please update your condition definition to use the new queryBuilderId 'idsConditionQueryBuilder'.
Legacy mappings are deprecated and may be removed in future versions.
```

Migration Strategy: While the mapping system provides backward compatibility, it is recommended to update your custom condition definitions to use the new queryBuilder IDs for better maintainability and to prepare for future versions where legacy support may be removed. The warning system will help you identify which condition definitions need to be updated.

9.14.5. Configuration Changes

OpenSearch Security Configuration (Unomi 4.0+)

OpenSearch as a Unomi backend is supported starting with **Apache Unomi 4.0** (not required for a pure 2.x 3.0 upgrade on Elasticsearch). When using OpenSearch 3.x with Unomi 4.0, security is commonly enabled and requires:

```
# OpenSearch security settings (sslEnable defaults to true in package samples)
org.apache.unomi.opensearch.sslEnable=true
org.apache.unomi.opensearch.username=admin
org.apache.unomi.opensearch.password=${env:OPENSEARCH_INITIAL_ADMIN_PASSWORD:-
admin}
org.apache.unomi.opensearch.sslTrustAllCertificates=true
```

Important Notes: - Security is enabled by default in OpenSearch 3.x - The default admin username is 'admin' - The initial admin password can be set via OPENSEARCH_INITIAL_ADMIN_PASSWORD environment variable - SSL/TLS is required for OpenSearch connections

Docker Configuration Changes

When using Docker, you can specify the search engine backend using the distribution environment variable UNOMI_DISTRIBUTION.:

- **For Elasticsearch**: UNOMI_DISTRIBUTION=unomi-distribution-elasticsearch
- **For OpenSearch**: UNOMI_DISTRIBUTION=unomi-distribution-opensearch
- **For custom configurations**: UNOMI_DISTRIBUTION=your-custom-distribution-name

9.14.6. Elasticsearch to OpenSearch Migration

Apache Unomi 4.0 introduces official support for OpenSearch as an alternative to Elasticsearch. If you want to migrate from Elasticsearch to OpenSearch, you can use the dedicated migration guide located in the 3.0 4.0 migration section.

For detailed instructions on migrating your data from Elasticsearch to OpenSearch, please refer to the Elasticsearch to OpenSearch migration guide.

9.14.7. Migration Checklist

Before upgrading to Apache Unomi 3.0, complete the following checklist:

Pre-Migration Requirements

- **Java 17+**: Ensure Java 17 or later is installed and configured
- **Search Engine**: Upgrade Elasticsearch to version 9.x
- **Backup**: Create a complete backup of your current Apache Unomi 2.x installation and data
- **Test Environment**: Test the migration in a non-production environment first

Custom Configuration Review

- **QueryBuilder IDs**: Review and update any custom condition definitions that reference old queryBuilder IDs
- **OpenSearch Security**: If migrating to OpenSearch in 4.0, configure security settings as required
- **Docker Configuration**: Update Docker environment variables if using containerized deployment

Post-Migration Verification

- **Functionality**: Test all custom conditions and plugins
- **Performance**: Monitor system performance and adjust resources if needed
- **Security**: Verify security configurations are working correctly
- **Data Integrity**: Confirm all data has been migrated successfully

9.14.8. Other Breaking Changes

For information about other breaking changes in Apache Unomi 3.0, please refer to the What's new section of the documentation.

9.14.9. Migrate from Elasticsearch 7 to Elasticsearch 9

You can use the **remote reindex** API to upgrade directly from Elasticsearch 7 to Elasticsearch 9. This approach runs both clusters in parallel and uses Elasticsearch's remote reindex feature.

To execute the migration, you should have one Elasticsearch 7 running (your source) and one Elasticsearch 9 running (your target).

This upgrade relies on a script. If you are sharing the Elasticsearch instance with other projects, it might need to be adjusted.

The script migration_es7-es9.sh at the root of the project handles: * Regular indices and rollover indices with their aliases * ILM policies migration * Data reindexing from ES7 to ES9 * Validation and comparison reporting

Prerequisites

- bash shell
- jq command-line JSON processor
- curl for HTTP requests
- Access to both ES7 (source) and ES9 (destination) clusters
- **ES9 must have reindex.remote.whitelist configured** (see configuration below)
- Ensure the machine where ES9 is running has access to the ES7 environment

Install jq if not already installed:

```
# macOS
brew install jq

# Linux
apt-get install jq
# or
yum install jq
```

Elasticsearch 9 Remote Reindex Configuration

Before running the script, you must configure the remote reindex whitelist on your ES9 cluster. Add this to your elasticsearch.yml configuration file:

```
reindex.remote.whitelist: "your-es7-host:9200"
```

Script Configuration

The script uses environment variables for configuration. Export variables before running the script:

```
export ES7_HOST="http://your-es7-host:9200"
export ES7_USER="elastic"
export ES7_HOST_FROM_ES9="http://your-es7-host-viewed-from-es9:9200"
export ES7_PASSWORD="your-es7-password"

export ES9_HOST="http://your-es9-host:9200"
export ES9_USER="elastic"
export ES9_PASSWORD="your-es9-password"

export INDEX_PREFIX="context-"
# Script default includes a trailing hyphen and concatenates patterns (context-profile, ...).
# Unomi system property org.apache.unomi.*.index.prefix is the prefix *without* a hyphen
(`context`);
# the persistence layer adds `-` when forming index names.
export BATCH_SIZE="1000"
```

Configuration Variables

Variable	Description	Default
ES7_HOST	Elasticsearch 7 URL	http://localhost:9200
ES7_HOST_FROM_ES9	Elasticsearch 7 URL visible from Elasticsearch 9	(value of ES7_HOST)
ES7_USER	ES7 username	elastic
ES7_PASSWORD	ES7 password	password
ES9_HOST	Elasticsearch 9 URL	http://localhost:9201
ES9_USER	ES9 username	elastic
ES9_PASSWORD	ES9 password	password
INDEX_PREFIX	Prefix for index names (script concatenates; include trailing -)	context-
BATCH_SIZE	Reindex batch size	1000

9.14.10. Execution

Make the script executable and run it:

```
chmod +x migration_es7-es9.sh
./migration_es7-es9.sh
```

What the Script Does

- Discovers indices matching the configured patterns on ES7
- Collects source statistics (document count, size) for each index
- Migrates ILM policies from ES7 to ES9 if they exist
- Creates indices on ES9 with the same settings and mappings
- Recreates aliases with proper write index flags for rollover indices
- Reindexes data from ES7 to ES9 using the remote reindex API
- Collects destination statistics after migration
- Displays a comparison report showing document counts and any mismatches

Output

The script provides detailed logging with timestamps and a final comparison report:

```
==========================================
MIGRATION COMPARISON REPORT
==========================================
Index                          |  Source Docs |     Dest Docs |    Difference |    Status
-----------------------------------+--------------+---------------+---------------+----------
context-profile                |        15420 |         15420 |           +0 |     OK
context-session-000001         |         3420 |          3420 |           +0 |     OK
==========================================
 All indices migrated successfully!
==========================================
```

9.15. From version 1.6 to 2.0

9.16. Migration Overview

Apache Unomi 2.0 is a major release, and as such it does introduce breaking changes. This portion of the document detail the various steps we recommend following to successfully migrate your environment from Apache Unomi 1.6 to Apache Unomi 2.0.

There are two main steps in preparing your migration to Apache Unomi 2.0: - Updating applications consuming Unomi - Migrating your existing data

9.17. Updating applications consuming Unomi

Since Apache Unomi is an engine, you've probably built multiple applications consuming its APIs, you might also have built extensions directly running in Unomi.

As you begin updating applications consuming Apache Unomi, it is generally a good practice to enable debug mode. Doing so will display any errors when processing events (such as JSON Schema validations), and will provide useful indications towards solving issues.

9.17.1. Data Model changes

There has been changes to Unomi Data model, please make sure to review those in the What's new section of the user manual.

9.17.2. Create JSON schemas

Once you updated your applications to align with Unomi 2 data model, the next step will be to create the necessary JSON Schemas.

Any event (and more generally, any object) received through Unomi public endpoints do require a valid JSON schema. Apache Unomi ships, out of the box, with all of the necessary JSON Schemas for its own operation as well as all event types generated from the Apache Unomi Web Tracker but you will need to create schemas for any custom event you may be using.

When creating your new schemas, there are multiple ways of testing them:

- Using a the event validation API endpoint available at the URL : /cxs/jsonSchema/validateEvent
- Using debug logs when sending events using the usual ways (using the /context.json or /eventcollector endpoints)

Note that in both cases it helps to activate the debug logs, that may be activated either:

- Through the ssh Karaf console command : log:set DEBUG org.apache.unomi.schema.impl.SchemaServiceImpl
- Using the UNOMI_LOGS_JSONSCHEMA_LEVEL=DEBUG environment variable and then restarting Apache Unomi. This is especially useful when using Docker Containers.

Once the debug logs are active, you will see detailed error messages if your events are not matched with any deployed JSON schema.

Note that it is currently not possible to modify or surcharge an existing system-deployed JSON schema via the REST API. It is however possible to deploy new schemas and manage them through the REST API on the /cxs/jsonSchema endpoint. If you are currently using custom properties on an Apache Unomi-provided event type, you will need to either change to use a new custom eventType and create the corresponding schema or to create a Unomi schema extension. You can find more details in the JSON Schema section of this documentation.

You can use, as a source of inspiration for creating new schemas, Apache Unomi 2.0 schema located at: extensions/json-schema/services/src/main/resources/META-INF/cxs/schemas.

Finally, and although it is technically feasible, we recommend against creating permissive JSON Schemas allowing any event payload. This requires making sure that you don't allow undeclared properties by setting JSON schema keywords such as unevaluated properties to false.

9.18. Migrating your existing data

9.18.1. Elasticsearch version and capacity

While still using Unomi 1.6, the first step will be to upgrade your Elasticsearch to 7.17.5. Documentation is available on Elasticsearch's website.

Your Elasticsearch cluster must have enough capacity to handle the migration. At a minimum, the required capacity storage capacity must be greater than the size of the dataset in production + the size of the largest index. Any other settings should at least be as big as the source setup (preferably higher).

9.18.2. Migrate custom data

Apache Unomi 2.0 knows how to migrate its own data from the new model to the old one, but it does not know how to migrate custom events you might be using in your environment.

It relies on a set of groovy scripts to perform its data migration, located in tools/shell-commands/src/main/resources/META-INF/cxs/migration, these scripts are sorted alphabetically and executed sequentially when migration is started. You can use these scripts as a source of inspiration for

creating your own.

In most cases, migration steps consist of an Elasticsearch painless script that will handle the data changes.

Depending of the volume of data, migration can be lengthy. By paying attention to when re-indexation is happening (triggered in the groovy scripts by MigrationUtils.reIndex()), you can find the most appropriate time for your scritps to be executed and avoid re-indexing the same indices multiple times.

For example if you wanted to update profiles with custom data (currently migrated by migrate-2.0.0-10-profileReindex.groovy), you could create a script in position "09" that would only contain painless scripts without a reindexing step. The script in position "10" will introduce its own painless script, then trigger the re-indexation. This way you don't have to re-index the same indices twice.

You can find existing painless scripts in tools/shell-commands/src/main/resources/requestBody/2.0.0

At runtime, and when starting the migration, Unomi 2.0 will take its own scripts, any additional scripts located in data/migration/scripts, will sort the resulting list alphabetically and execute each migration script sequentially.

9.18.3. Perform the migration

Checklist

Before starting the migration, please ensure that:

- You do have a backup of your data
- You did practice the migration in a staging environment, NEVER migrate a production environment without prior validation
- You verified your applications were operational with Apache Unomi 2.0 (JSON schemas created, client applications updated, ...)
- You are running Elasticsearch 7.17.5 (or a later 7.x version)
- Your Elasticsearch cluster has enough capacity to handle the migration
- You are currently running Apache Unomi 1.6 (or a later 1.x version)
- You will be using the same Apache Unomi instance for the entire migration progress. Do not start the migration on one node, and resume an interrupted migration on another node.

Migration process overview

The migration is performed by means of a dedicated Apache Unomi 2.0 node started in a particular migration mode.

In a nutshell, the migration process will consist in the following steps:

- Shutdown your Apache Unomi 1.6 cluster

- Start one Apache Unomi 2.0 node that will perform the migration (upon startup)
- Wait for data migration to complete
- Start your Apache Unomi 2.0 cluster
- (optional) Import additional JSON Schemas

Each migration step maintains its execution state, meaning if a step fails you can fix the issue, and resume the migration from the failed step.

Configuration

The following environment variables are used for the migration:

Environment Variable	Unomi Setting	Default
UNOMI_ELASTICSEARCH_ADDRESSES	org.apache.unomi.elasticsearch.addresses	localhost:9200
UNOMI_ELASTICSEARCH_SSL_ENABLE	org.apache.unomi.elasticsearch.sslEnable	false
UNOMI_ELASTICSEARCH_USERNAME	org.apache.unomi.elasticsearch.username	
UNOMI_ELASTICSEARCH_PASSWORD	org.apache.unomi.elasticsearch.password	
UNOMI_ELASTICSEARCH_SSL_TRUST_ALL_CERTIFICATES	org.apache.unomi.elasticsearch.sslTrustAllCertificates	false
UNOMI_ELASTICSEARCH_INDEXPREFIX	org.apache.unomi.elasticsearch.index.prefix	context
UNOMI_MIGRATION_RECOVER_FROM_HISTORY	org.apache.unomi.migration.recoverFromHistory	true

If there is a need for advanced configuration, the configuration file used by Apache Unomi 2.0 is located in: etc/org.apache.unomi.migration.cfg

Migrate manually

You can migrate manually using the Karaf console.

After having started Apache Unomi 2.0 with the ./karaf command, you will be presented with the Karaf shell.

From there you have two options:

- The necessary configuration variables (see above) have already been set, you can start the migration using the command: unomi:migrate 1.6.0
- Or, you want to provide the configuration settings interactively via the terminal, in that case you

can start the migration in interactive mode using: unomi:migrate 1.6.0

The parameter of the migrate command (1.6.0 in the example above) corresponds to the version you're migrating from.

At the end of the migration, you can start Unomi 2.0 as usual using: unomi:start.

Migrate with Docker

The migration can also be performed using Docker images, the migration itself can be started by passing a specific value to the KARAF_OPTS environment variable.

In the context of this migration guide, we will asssume that:

- Custom migration scripts are located in /home/unomi/migration/scripts/
- Painless scripts, or more generally any migration assets are located in /home/unomi/migration/assets/, these scripts will be mounted under /tmp/assets/ inside the Docker container.

```
docker run \
  -e UNOMI_ELASTICSEARCH_ADDRESSES=localhost:9200 \
  -e KARAF_OPTS="-Dunomi.autoMigrate=1.6.0" \
  --v /home/unomi/migration/scripts/:/opt/apache-unomi/data/migration/scripts \
  --v /home/unomi/migration/assets/:/tmp/assets/ \
  apache/unomi:2.0.0
```

You might need to provide additional variables (see table above) depending of your environment.

If the migration fails, you can simply restart this command.

Using the above command, Unomi 2.0 will not start automatically at the end of the migration. You can start Unomi automatically at the end of the migration by passing: -e KARAF_OPTS="-Dunomi.autoMigrate=1.6.0 -Dunomi.autoStart=true".

Step by step migration with Docker

Once your cluster is shutdown, performing the migration will be as simple as starting a dedicated docker container.

Post Migration

Once the migration has been executed, you will be able to start Apache Unomi 2.0

Remember you still need to submit JSON schemas corresponding to your events, you can do so using the API.

9.19. From version 1.5 to 1.6

Migration from Unomi 1.5x to 1.6x does not require any particular steps, simply restart your cluster in the new version.

9.20. From version 1.4 to 1.5

9.20.1. Migration from Apache Unomi 1.4 to 1.5

Data model and ElasticSearch 7

Since Apache Unomi version 1.5.0 we decided to upgrade the supported ElasticSearch version to the 7.4.2.

To be able to do so, we had to rework the way the data was stored inside ElasticSearch.

Previously every items was stored inside the same ElasticSearch index but this is not allowed anymore in recent ElasticSearch versions.

Since Apache Unomi version 1.5.0 every type of items (see section: Item) is now stored in a dedicated separated index.

API changes

To be able to handle the multiple indices the Persistence API implementation (ElasticSearchPersistenceServiceImpl) have been adapted and simplified.

The good news is that there is no API changes, the persistence API interface didn't changed.

Any custom Apache Unomi plugins or extensions should continue to work on Apache Unomi 1.5.0.

The only notable changes are located at the ElasticSearchPersistenceServiceImpl Java class. This class should not be use directly, instead you should use OSGI service dependency injection using the interface PersistenceService.

But if you are interested in the implementation changes:

1. The property index.name have been renamed to index.prefix. Previously used for the single one index name, now every index is prefixed using this property. (context- by default)
2. We removed the property index.names originally used to create additional indices (used by the geonames DB for exemple). This property is not needed anymore because the index is automatically created by the peristence service when the mapping configuration is loaded. Example of mapping configuration file: (geoname index mapping)

Because of this changes the geonames DB index name is now respecting the index naming with prefix like any other item type. Previously named: geonames is now using the index name context-geonameentry (see: Documentation about geonames extension).

Migration steps

In order to migrate the data from ElasticSearch 5 to 7, Unomi provides a migration tool that is directly integrated.

In this migration the following is assumed:

- the ElasticSearch 5 cluster installation is referred to as the source
- the ElasticSearch 7 cluster installation is referred to as the target
- the Unomi 1.4 cluster installation is completely stopped
- the Unomi 1.5 cluster installation has never been started (just uncompressed)
- the Unomi 1.5 cluster installation has been configured to connect to the target (ElasticSearch 7) cluster

It is HIGHLY RECOMMENDED to perform a full cluster backup/snapshot of the source clusters (including ElasticSearch and Unomi clusters), and ideally to perform the migration on a restored snapshot of the source cluster. For more information on ElasticSearch 5 snapshots and restore you can find it here:

```
https://www.elastic.co/guide/en/elasticsearch/reference/5.6/modules-snapshots.html
```

The way the migration works is that both ElasticSearch 5 AND an ElasticSearch 7 clusters (or just single nodes) will be started at the same time, and data will be migrated from the ES 5 to the ES 7 cluster. Note that it is possible to use a single node for both the source and the target clusters to - for example - perform the migration on a single machine. If you choose to do that you will have to adjust port numbers on either the source or target cluster node. Changing ports requires a restart of the ES cluster you are modifying. In this example we will illustrate how to migrate by modifying the source cluster node ports.

So in the source 's ElasticSearch 5 config/elasticsearch.yml file we have modified the default ports to:

```
transport.tcp.port: 9310
http.port: 9210
```

Make SURE you change the ports out of the default 9200-9205 and 9300-9305 range (or whatever your cluster uses) otherwise both clusters will attempt to merge!

On the target ElasticSearch 7 cluster configuration you will need to add the following setting in the config/elasticsearch.yml:

```
reindex.remote.whitelist: "localhost:9210"
```

Replace "localhost:9210" which whatever location your source cluster is available at. Restart or start your target ElasticSearch 7 cluster.

Important: Make sure you haven't started Apache Unomi before (using the unomi:start command or the autostart command line parameter) otherwise you will need to restart your Apache Unomi installation from scratch. The best way to be sure of that is to start a new Unomi install by uncompressing the archive and not launching it.

You can then start both instances of ElasticSearch 5 and ElasticSearch 7 and finally start Apache Unomi using:

```
./karaf
```

Once in the console launch the migration using the following command:

```
migrate 1.4.0
```

Note: the 1.4.0 version is the starting version. If you are starting from a different version (for example a fork), make sure that you know what official version of Apache Unomi it corresponds to and you can use the official version number as a start version for the migration.

Follow the instructions and answer the prompts. If you used the above configuration as an example you can simply use the default values.

Be careful because the first address that the tool will ask for is the target (ElasticSearch 7) cluster, not the ES 5 one.

Note that it is also possible to change the index prefix to be different from the default context value so that you could host multiple Apache Unomi instances on the same ElasticSearch cluster.

Important note: only the data that Apache Unomi manages will be migrated. If you have any other data (for example Kibana or ElasticSearch monitoring indices) they will not be migrated by this migration tool.

Once the migration has completed, you can start the new Unomi instance using:

```
unomi:start
```

You should then validate that all the data has been properly migrated. For example you could issue a command to list the profiles:

```
profile-list
```

9.21. Important changes in public servlets since version 1.5.5 and 2.0.0

What used to be dedicated servlets are now part of the REST endpoints. Prior to version 1.5.5 the following servlets were used:

- /context.js /context.json
- /eventcollector
- /client

In version 2.0.0 and 1.5.5 and later you have to use the new cxs REST endpoints:

- /cxs/context.js /cxs/context.json
- /cxs/eventcollector
- /cxs/client

The old servlets have been deprecated and will be removed in a future major version, so make sure to update your client applications.

Chapter 10. Conditions

10.1. Conditions guide

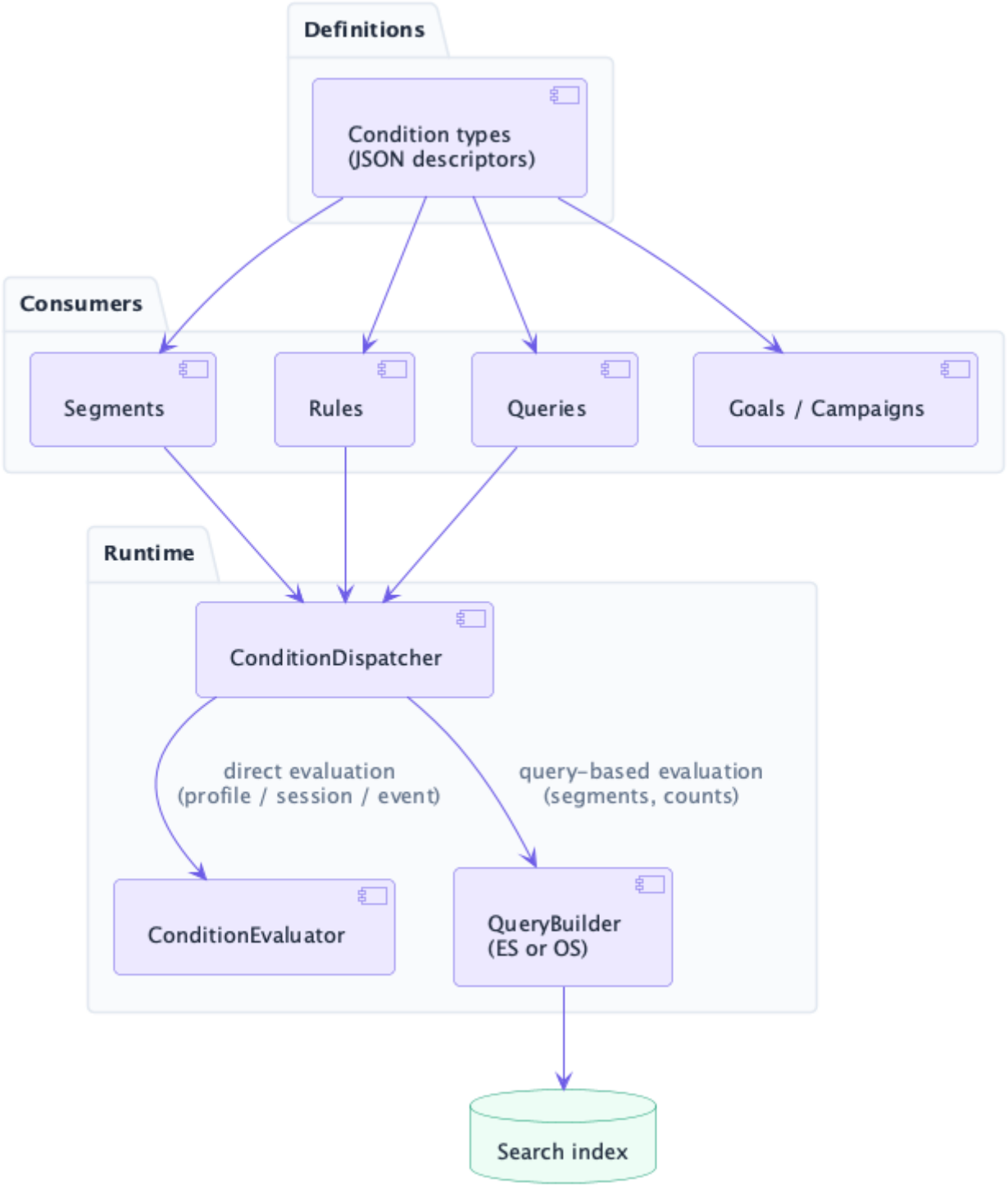
Conditions are the building blocks of segmentation, personalization, rules, queries, campaigns, and goals in Apache Unomi. This guide walks from simple property checks to advanced behavioral conditions with diagrams and REST examples.

See also:

- Built-in condition types — full catalog
- Condition validation — save-time checks and parameter rules
- Condition evaluation system — runtime architecture
- Past event conditions — deep dive on behavioral conditions
- Queries and aggregations — using conditions in search APIs
- Debugging conditions — tracing and troubleshooting

10.1.1. Concepts

- **Condition type** — a reusable definition (JSON descriptor) registered at startup, for example profilePropertyCondition.
- **Condition instance** — a JSON object with type and parameterValues used in segments, rules, or queries.
- **Evaluation context** — profile, session, and/or current event, depending on where the condition runs.



10.1.2. Condition instance JSON shape

Every condition instance uses the same structure:

```
{
  "type": "CONDITION_TYPE_ID",
  "parameterValues": {
    "parameterName": "value"
  }
}
```

Nested conditions use the Condition parameter type (for example subConditions on booleanCondition).

10.1.3. Comparison operators

Property conditions (profilePropertyCondition, sessionPropertyCondition, eventPropertyCondition) accept a comparisonOperator parameter. Common values include:

equals, notEquals, lessThan, greaterThan, lessThanOrEqualTo, greaterThanOrEqualTo, between, exists, missing, contains, notContains, startsWith, endsWith, matchesRegex, in, notIn, all, hasSomeOf, hasNoneOf, isDay, isNotDay, distance.

Use the typed value parameter that matches your data (propertyValue, propertyValueInteger, propertyValues, and so on).

10.1.4. Level 1 — Profile property check

Match profiles where a property equals a value. Used in segments and query counts.

```
{
  "type": "profilePropertyCondition",
  "parameterValues": {
    "propertyName": "properties.country",
    "comparisonOperator": "equals",
    "propertyValue": "US"
  }
}
```

Test with the query count API:

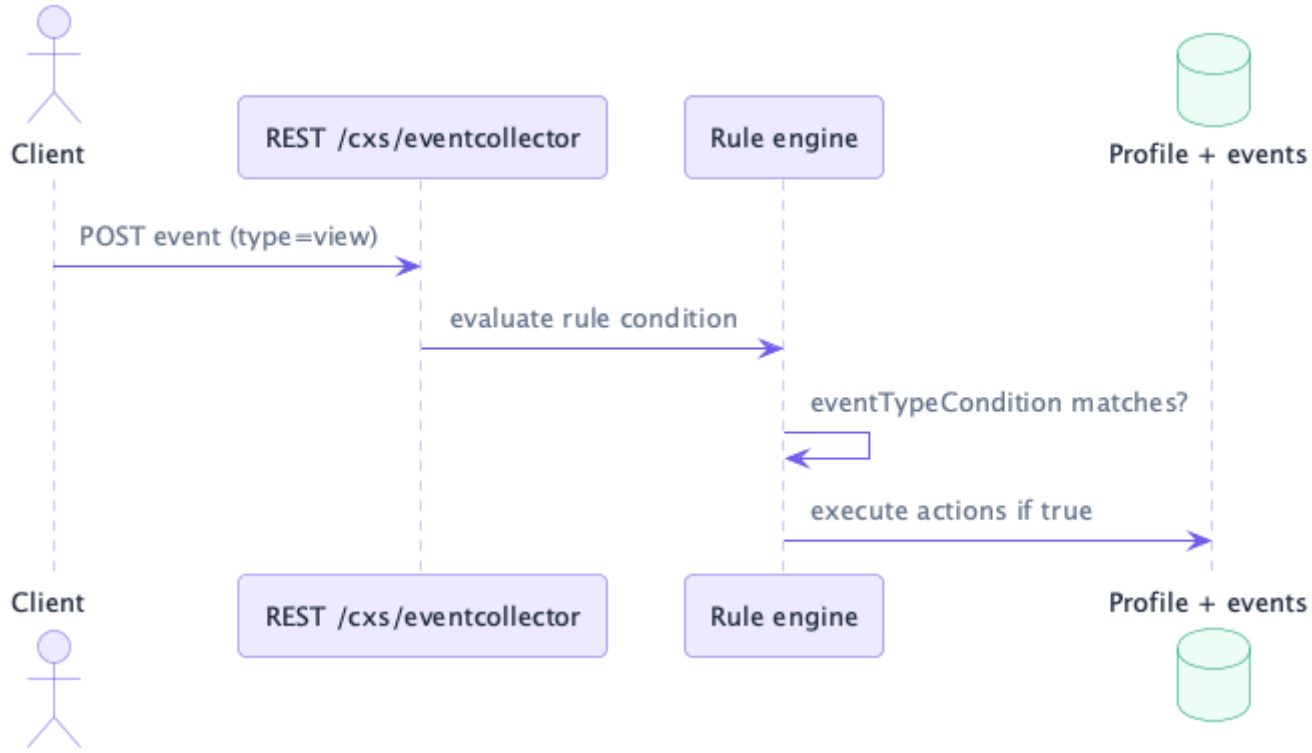
```
curl -X POST http://localhost:8181/cxs/query/profile/count \
  --user "TENANT_ID:PRIVATE_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "type": "profilePropertyCondition",
    "parameterValues": {
      "propertyName": "properties.country",
      "comparisonOperator": "equals",
      "propertyValue": "US"
    }
  }'
```

10.1.5. Level 2 — Event type in a rule

Rules evaluate **event conditions** against the incoming event. eventTypeCondition is a shortcut over eventPropertyCondition on eventType.

```
{
  "type": "eventTypeCondition",
  "parameterValues": {
    "eventTypeId": "view"
  }
}
```

Use this in a rule condition to fire when a view event arrives.



10.1.6. Level 3 — Boolean AND / OR

Combine conditions with booleanCondition. This is the main way to build segment definitions.

```
{
 "type": "booleanCondition",
 "parameterValues": {
  "operator": "and",
  "subConditions": [
   {
    "type": "profilePropertyCondition",
    "parameterValues": {
     "propertyName": "properties.nbOfVisits",
     "comparisonOperator": "greaterThan",
     "propertyValueInteger": 1
    }
   },
   {
    "type": "profilePropertyCondition",
    "parameterValues": {
     "propertyName": "properties.country",
     "comparisonOperator": "equals",
     "propertyValue": "US"
    }
   }
  ]
 }
}
```

An empty subConditions list matches every item in a search query. In-memory evaluation, as in rules, matches every item for an empty AND and none for an empty OR. To select every item, use matchAllCondition rather than an empty list. A sub-condition whose parameter:: reference is unset is left out before the operator applies; see Writing plugins for how condition types use these references.

Negate a sub-condition with notCondition:

```
{
 "type": "notCondition",
 "parameterValues": {
  "subCondition": {
   "type": "profileSegmentCondition",
   "parameterValues": {
    "segments": ["opt-out-segment"],
    "matchType": "in"
   }
  }
 }
}
```

10.1.7. Level 4 — Segment membership and session properties

profileSegmentCondition checks whether a profile belongs to other segments (useful for segment hierarchies).

```
{
  "type": "profileSegmentCondition",
  "parameterValues": {
    "segments": ["high-value-customers"],
    "matchType": "in"
  }
}
```

sessionPropertyCondition evaluates session-scoped properties during event processing:

```
{
  "type": "sessionPropertyCondition",
  "parameterValues": {
    "propertyName": "properties.referringCampaign",
    "comparisonOperator": "exists"
  }
}
```

10.1.8. Level 5 — Past events (behavioral)

pastEventCondition checks whether events occurred in a time window. See Past event conditions for caching, auto-rules, and performance notes.

```
{
  "type": "pastEventCondition",
  "parameterValues": {
    "numberOfDays": 30,
    "operator": "eventsOccurred",
    "minimumEventCount": 3,
    "eventCondition": {
      "type": "eventTypeCondition",
      "parameterValues": {
        "eventTypeId": "purchase"
      }
    }
  }
}
```

10.1.9. Level 6 — Scoring, geo, and nested data

Scoring — compare a profile score from a scoring plan:

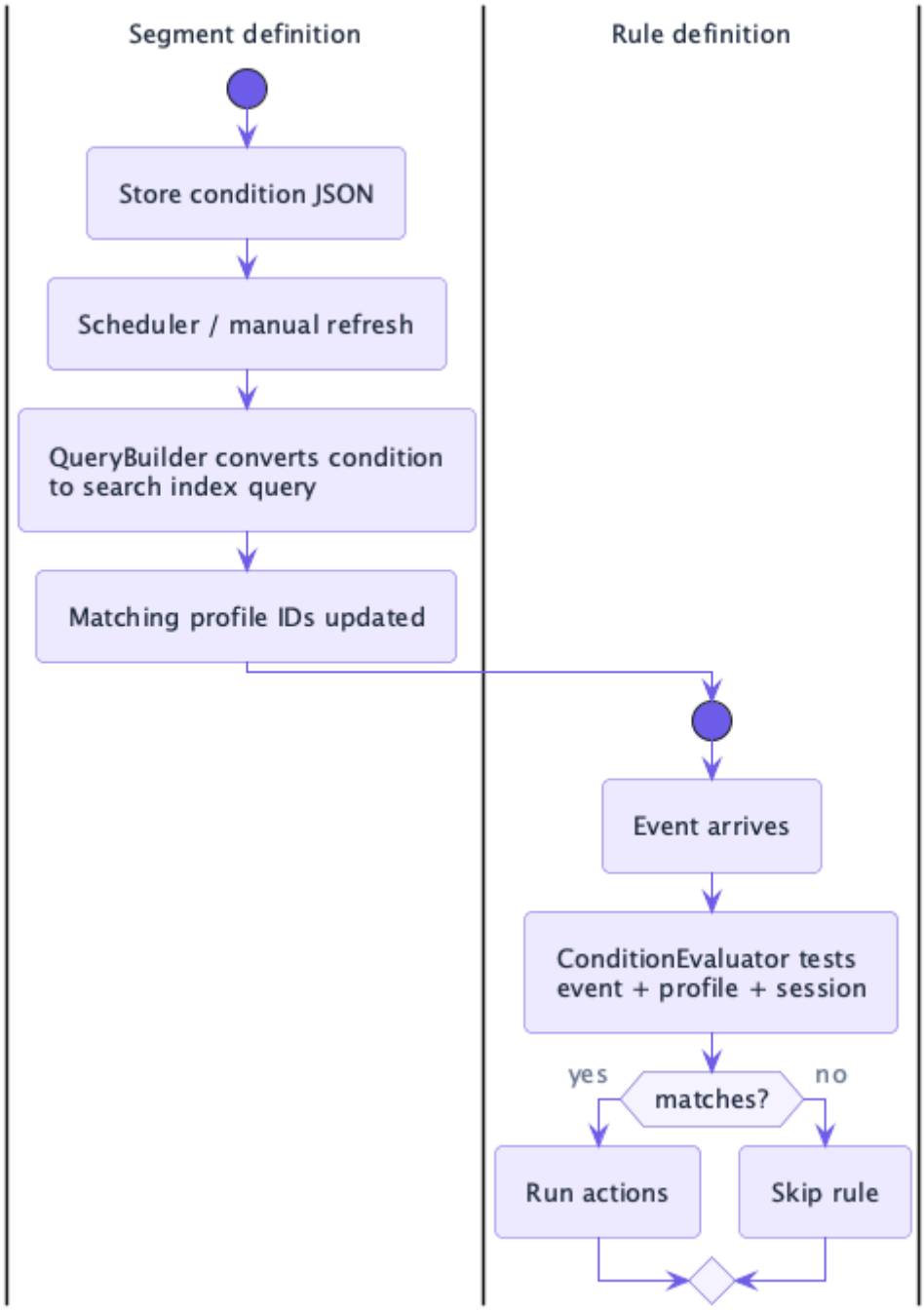
```
{
  "type": "scoringCondition",
  "parameterValues": {
    "scoringPlanId": "engagement",
    "comparisonOperator": "greaterThan",
    "scoreValue": 50
  }
}
```

Geo (session) — geoLocationSessionCondition filters on country/region/city; geoLocationByPointSessionCondition uses map coordinates.

Nested properties — nestedCondition evaluates a sub-condition on a nested path (for example list items inside profile properties).

10.1.10. Using conditions in segments and rules

Segments store a condition on the segment definition. Rules store a condition that is evaluated for each incoming event.



10.1.11. Testing conditions

Query count — verify how many profiles match (see Level 1 example).

Segment membership — after creating a segment with your condition, test a profile:

```
curl -X GET "http://localhost:8181/cxs/segments/SEGMENT_ID/match/PROFILE_ID" \
  --user "TENANT_ID:PRIVATE_KEY"
```

10.1.12. Discovering available condition types

List registered condition types (admin or tenant private key):

```
curl -X GET http://localhost:8181/cxs/definitions/conditions \
  --user "TENANT_ID:PRIVATE_KEY"
```

Each entry includes parameter definitions, evaluators, and query builder IDs (Unomi 3.0+ uses engine-neutral *QueryBuilder names).

10.1.13. Custom condition types

To add new condition types, implement a ConditionEvaluator and search-engine-specific query builders. See Writing plugins and plugin registration example.

Add validation metadata on parameters so invalid instances are rejected at save time — see Condition validation.

10.1.14. Debugging conditions

When a segment or rule does not behave as expected, use these approaches in order.

Test segment membership

Check whether a specific profile matches a segment condition:

```
curl -X GET "http://localhost:8181/cxs/segments/SEGMENT_ID/match/PROFILE_ID" \
  --user "TENANT_ID:PRIVATE_KEY"
```

Count matching profiles

Use a query with your condition and read the total hit count (see Level 1 in this guide).

Request tracing with explain

For runtime debugging — which rules fired, which conditions matched, how long each step took — add explain=true to context or event-collector requests. You need an ADMINISTRATOR or TENANT_ADMINISTRATOR role.

```
curl -X POST "http://localhost:8181/cxs/context.json?sessionId=1234&explain=true" \
  -H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
  -H "Content-Type: application/json" \
  -d @- <<'EOF'
{
  "source": { "itemId": "homepage", "itemType": "page", "scope": "mydigital" },
  "requireSegments": true
}
EOF
```

The response includes a requestTracing tree with nodes such as condition-evaluation, rule-evaluation, and event-validation. Validation steps during processing appear as condition-validation or rule-

condition-validation children.

Full documentation, sample trace JSON, and sequence diagrams:

- Using the explain parameter for request tracing



Use explain only when debugging. It adds overhead and exposes internal processing detail to admin users.

Fix save-time validation errors

If creating or updating a segment or rule fails with Invalid ... condition, the condition JSON does not match its type definition. Read the error list in the response, fix the parameter values, and retry.

Common causes:

- Missing required parameter
- Wrong value type (string instead of integer)
- Two parameters set in the same exclusiveGroup
- Nested condition missing a required tag (for example eventCondition inside pastEventCondition)

See Condition validation for the full rule set and plugin author guide.

10.2. Condition validation

Apache Unomi validates condition instances against their condition type definitions when you save segments, rules, queries, goals, and related items. This catches missing parameters, wrong value types, and invalid nested conditions **before** they reach runtime evaluation.



This is separate from JSON Schema event validation. JSON schemas validate incoming events on public endpoints. Condition validation checks the structure of condition JSON used in segments, rules, and queries.

See also:

- Conditions guide — hands-on introduction
- Built-in condition types — descriptor format and catalog
- Request tracing with explain — see validation results in trace output

10.2.1. When validation runs

Validation runs on the server when a condition is saved or resolved, not on every profile lookup or event.

Typical entry points:

- **Segments** — saving or updating a segment condition (SegmentServiceImpl)
- **Rules** — saving or updating a rule condition (RulesServiceImpl)
- **Queries** — saving queries used in profiles, events, goals, and campaigns
- **Goals** — start and target event conditions
- **Condition type resolution** — when DefinitionsService resolves a condition tree (deprecated path still used in some flows)

There is **no** standalone REST endpoint such as /cxs/conditions/validate. To test a condition, save it on a segment or rule and read the error response, or use debugging with explain when the condition is evaluated at runtime.

10.2.2. Validation metadata on condition types

Each parameter in a condition type descriptor can include a validation object. The base plugin ships examples in plugins/baseplugin/src/main/resources/META-INF/cxs/conditions/*.json.

Supported fields:

Field	Purpose	Example use
required	Parameter must have a literal value at save time	propertyName on profilePropertyCondition
recommended	Parameter is optional but advised; produces a warning, not a blocking error	minimumEventCount on pastEventCondition
allowedValues	String value must be one of the listed values	operator on pastEventCondition (eventsOccurred, eventsNotOccurred)
exclusive + exclusiveGroup	Only one parameter in the group may have a value	propertyValue vs propertyValueInteger on profilePropertyCondition
allowedConditionTags	Nested condition must carry at least one of these system tags	eventCondition tag required for eventCondition on pastEventCondition
disallowedConditionTypes	Nested condition type ID must not be in this list	Restrict which sub-conditions a composite type accepts

Example from pastEventCondition:

```
{
 "id": "eventCondition",
 "type": "Condition",
 "validation": {
  "required": true,
  "allowedConditionTags": ["eventCondition"]
 }
}
```

Example of mutually exclusive value parameters from profilePropertyCondition:

```
{
 "id": "propertyValue",
 "type": "string",
 "validation": {
  "exclusive": true,
  "exclusiveGroup": "propertyValue"
 }
},
{
 "id": "propertyValueInteger",
 "type": "integer",
 "validation": {
  "exclusive": true,
  "exclusiveGroup": "propertyValue"
 }
}
```

10.2.3. Parameter type checks

Before rule-specific checks, the validation service verifies that each literal value matches the parameter type and multivalued flag.

Built-in type validators cover:

- string, integer, long, float, double, boolean, date
- condition — nested condition instances (validated recursively)
- comparisonOperator — custom validator for comparison operators

Plugins can register additional ValueTypeValidator OSGi services for custom parameter types.

10.2.4. Contextual parameters (partial validation)

Parameters whose values start with parameter:: or script:: are resolved at evaluation time, not at save time. The validation service skips most checks on these values because the final value is unknown when the item is saved.

Exceptions:

- A **required** parameter that uses a contextual reference still produces a **warning** (MISSING_RECOMMENDED_PARAMETER) advising that static validation was skipped.
- Nested conditions inside contextual values are not validated until runtime.

This lets rules and segments use dynamic values while still validating fully literal condition trees.

10.2.5. Error types

Validation returns a list of ValidationError objects. Each error has a type, a human-readable message, the parameterName, and optional context (location, allowed values, actual value).

Type	Meaning	Blocks save?
MISSING_REQUIRED_PARAMETER	Required parameter is absent or an empty collection	Yes
INVALID_VALUE	Value has wrong type, shape, or is not in allowedValues	Yes
INVALID_CONDITION_TYPE	Unknown condition type, or nested condition fails tag/type rules	Yes
EXCLUSIVE_PARAMETER_VIOLATION	More than one parameter in an exclusive group has a value	Yes
MISSING_RECOMMENDED_PARAMETER	Recommended parameter is missing, or required parameter uses a contextual reference	No (logged as warning)

10.2.6. Save-time behavior

On save, services separate **errors** from **warnings**:

- **Errors** — any type except MISSING_RECOMMENDED_PARAMETER — cause the save to fail with an IllegalArgumentException or domain-specific exception (for example BadSegmentConditionException for segments).
- **Warnings** — MISSING_RECOMMENDED_PARAMETER only — are written to the server log but do not block the save.

Example error message when saving a segment with a missing required parameter:

```
Invalid segment condition:
- Required parameter is missing (parameter: propertyName, condition: profilePropertyCondition, location: condition[profilePropertyCondition].propertyName)
```

REST clients receive a generic badRequest response when the save fails. The detailed text above is written to the server log.

10.2.7. Rules and invalid conditions

By default, a rule with validation errors cannot be saved. During bundle deployment or when a rule references a missing plugin, Unomi may save the rule with allowInvalidRules semantics: validation errors are logged and the rule is marked invalid instead of rejected. This supports gradual plugin rollout without blocking server startup.

10.2.8. Validation in request tracing

When explain=true is set on context or event-collector requests, validation steps appear in the requestTracing tree. Look for operation types such as:

- condition-validation — validation during condition type resolution
- segment-condition-validation — segment save path (when triggered in that flow)
- rule-condition-validation — rule condition checks

Trace nodes include validation messages and whether validation passed. See Using the explain parameter for request tracing for full examples.

10.2.9. Adding validation to custom condition types

When you define a new condition type in a plugin:

1. Declare parameters with correct type and multivalued flags.
2. Add a validation block on parameters that need constraints.
3. Use allowedConditionTags on nested Condition parameters to restrict which sub-types users can pick.
4. Use exclusive / exclusiveGroup when several parameters represent alternative value shapes (string vs integer vs date).
5. Register a custom ValueTypeValidator OSGi service if you introduce a new parameter type ID.

Condition type descriptors are loaded at startup. After deployment, list types to confirm your validation metadata is present:

```
curl -X GET http://localhost:8181/cxs/definitions/conditions \
  --user "TENANT_ID:PRIVATE_KEY"
```

Each condition type entry in the response includes parameter definitions with their validation objects, which UIs and integrators can use to guide authors before save.

10.2.10. Relation to JSON Schema validation

	Condition validation	JSON Schema validation
What is validated	Condition instances in segments, rules, queries	Event payloads on /cxs/context.json and /cxs/eventcollector
When	Save time (server-side)	Request time (public endpoints)
Defined in	Condition type JSON descriptors (META-INF/cxs/conditions/)	JSON schema definitions (META-INF/cxs/json-schema/)
REST test endpoint	None (save the parent item instead)	/cxs/jsonSchema/validateEvent

Both systems can appear in the same explain=true trace: event schema validation under event-validation, condition checks under condition-evaluation or condition-validation.

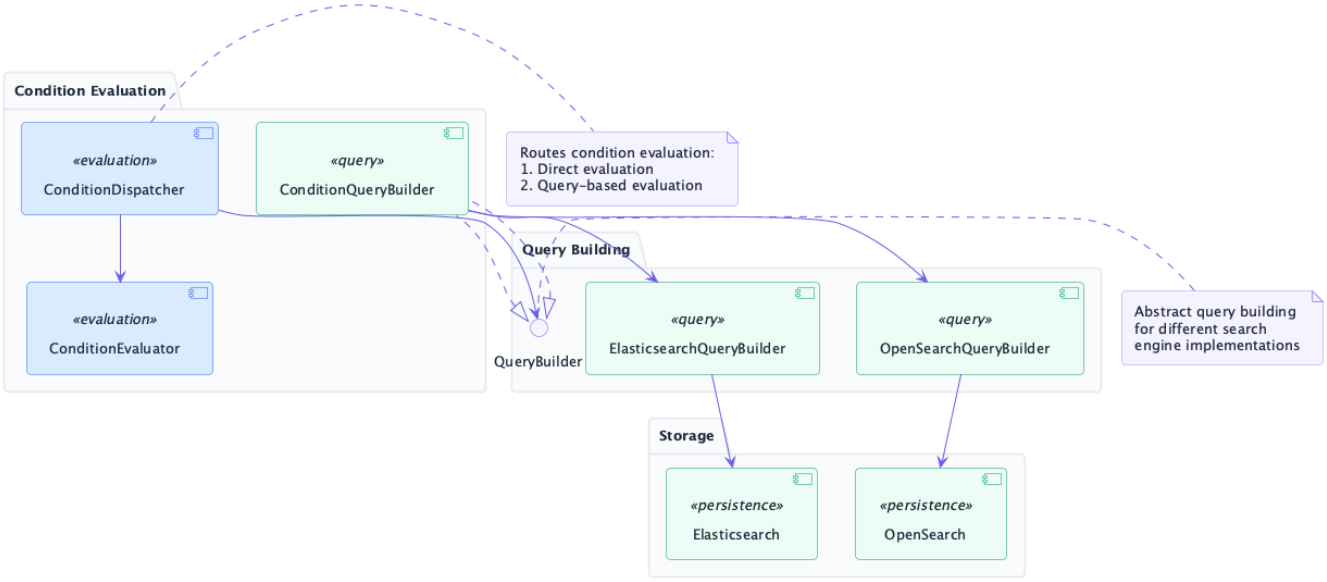
10.3. Condition Evaluation System

For examples and progressive tutorials, see Conditions guide.

The condition evaluation system in Apache Unomi provides flexible and efficient evaluation of conditions across different storage backends.

10.3.1. Architecture Overview

10.4. Condition Evaluation Architecture



10.4.1. Key Components

1. **ConditionDispatcher**
    - Routes conditions to appropriate evaluators
    - Determines evaluation strategy (direct vs query-based)
    - Handles caching and optimization

2. **ConditionEvaluator**
    - Performs direct condition evaluation
    - Handles boolean logic and property comparisons
    - Evaluates nested conditions
3. **QueryBuilders**
    - Convert conditions to search engine queries
    - Handle search engine specific syntax
    - Optimize query performance

10.4.2. Evaluation Strategies

1. **Direct Evaluation**
    - Used for simple conditions
    - Evaluates against in-memory data
    - No storage queries required
    - Fastest evaluation path
2. **Query-Based Evaluation**
    - Used for complex conditions
    - Converts conditions to storage queries
    - Supports different search engines
    - Handles large data sets efficiently

10.4.3. Query Builder Implementation

Query builders provide storage-specific implementations:

1. **Elasticsearch Implementation**
    - Optimized for ES query syntax
    - Handles ES-specific mappings
    - Supports ES versioning
2. **OpenSearch Implementation**
    - Compatible with OS API
    - Maintains OS-specific features
    - Handles OS security model

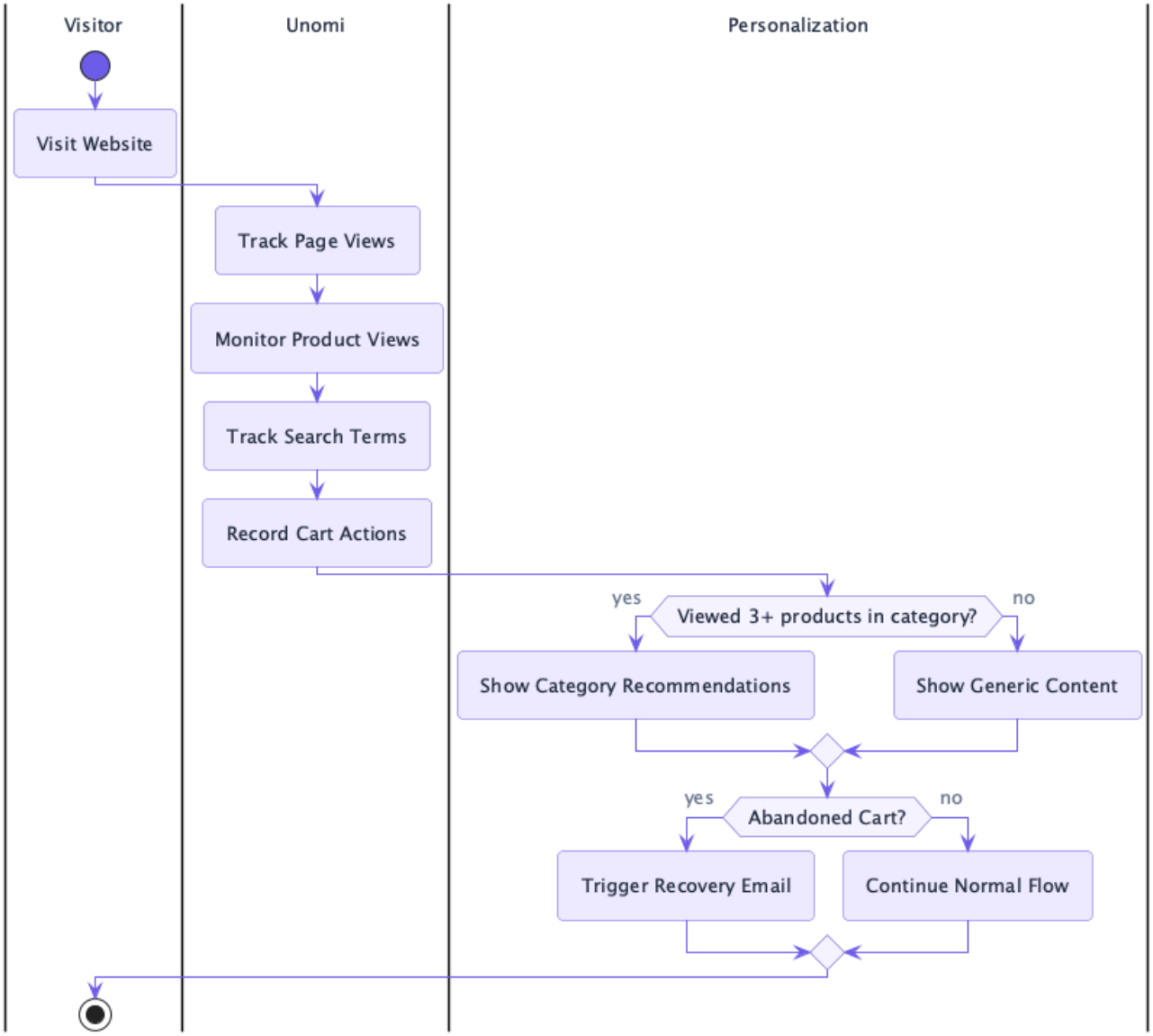
10.4.4. Best Practices

1. **Performance Optimization**
    - Use direct evaluation when possible
    - Leverage caching for frequent queries
    - Consider query complexity impact
2. **Implementation Guidelines**
    - Implement storage-specific query builders
    - Handle version compatibility
    - Consider security implications

10.5. Past Event Conditions in Apache Unomi

10.5.1. Introduction

Past event conditions are a powerful feature in Apache Unomi that allows both segments and rules to be defined based on events that occurred within a specific time window. This document explains how past event conditions work, their different usage patterns, and performance implications.


Visitor
Unomi
Personalization
Visit Website
Track Page Views
Monitor Product Views
Track Search Terms
Record Cart Actions
yes
Viewed 3+ products in category?
no
Show Category Recommendations
Show Generic Content
yes
Abandoned Cart?
no
Trigger Recovery Email
Continue Normal Flow

Example: Product Interest Segment

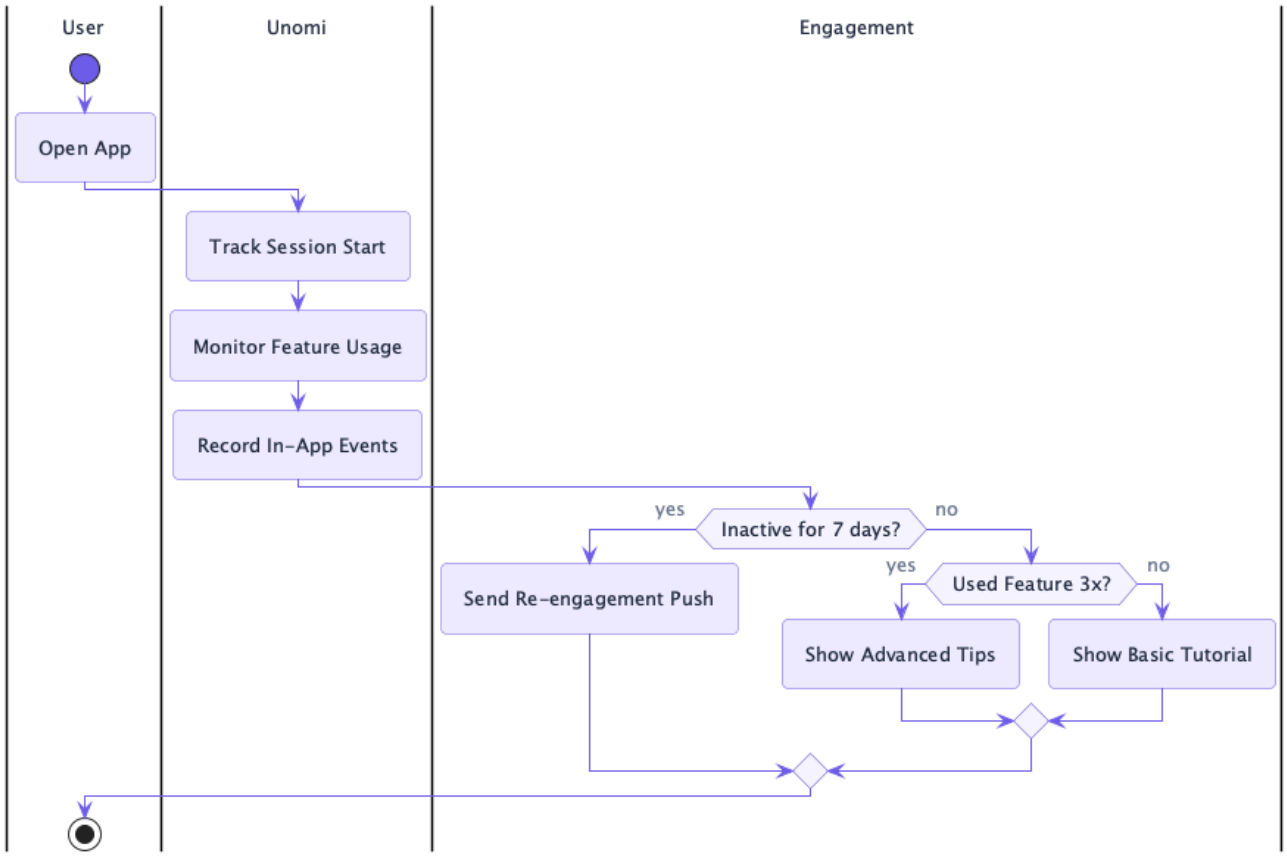
```
{
 "metadata": {
  "id": "electronics_enthusiast",
  "name": "Electronics Category Interest"
 },
 "condition": {
  "type": "pastEventCondition",
  "parameterValues": {
   "eventType": "view",
   "numberOfDays": 30,
   "minimumCount": 3,
   "propertyConditions": {
    "type": "propertyCondition",
    "parameterValues": {
     "propertyName": "target.properties.category",
     "comparisonOperator": "equals",
     "propertyValue": "electronics"
    }
   }
  }
 }
}
```

Common website scenarios include:

1. **Shopping Cart Abandonment**
    - Track cart additions and checkout starts
    - Identify abandoned carts after time threshold
    - Trigger recovery campaigns
2. **Content Engagement**
    - Monitor article views and time spent
    - Track topic interests
    - Personalize content recommendations

```
{
 "type": "pastEventCondition",
 "parameterValues": {
  "eventType": "articleView",
  "numberOfDays": 30,
  "minimumCount": 5,
  "propertyConditions": {
   "type": "propertyCondition",
   "parameterValues": {
    "propertyName": "topic",
    "comparisonOperator": "equals",
    "propertyValue": "technology"
   }
  }
 }
}
```

10.5.2. Mobile Application Engagement



Common mobile scenarios include:

1. **User Activation**
   - Track feature discovery and usage
   - Identify activation milestones
   - Guide user onboarding

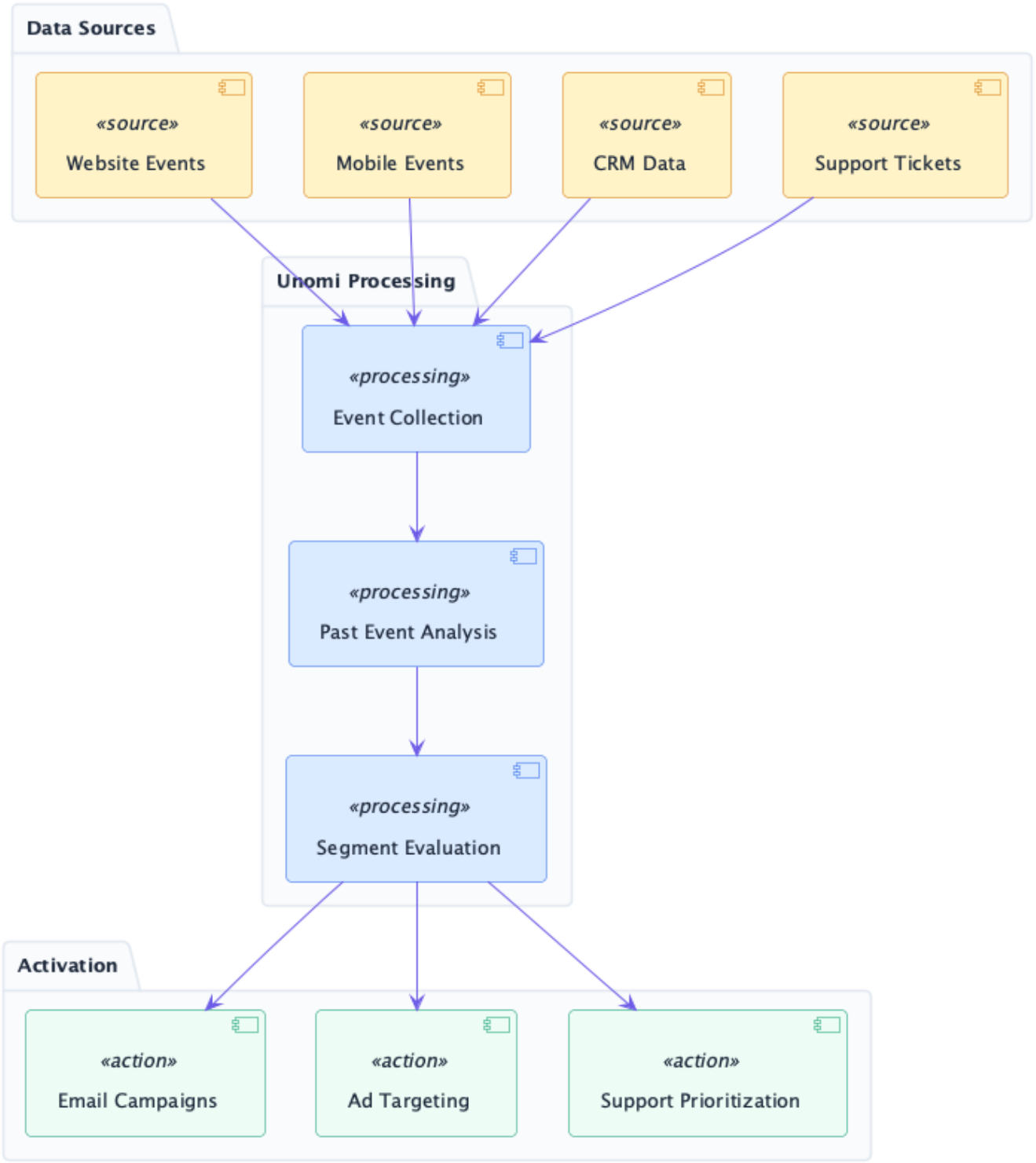
2. **Retention Campaigns**

    - Monitor app opens and session frequency
    - Detect usage patterns
    - Trigger re-engagement actions

```
{
 "type": "pastEventCondition",
 "parameterValues": {
  "eventType": "featureUsed",
  "numberOfDays": 7,
  "minimumCount": 3,
  "propertyConditions": {
   "type": "propertyCondition",
   "parameterValues": {
    "propertyName": "featureId",
    "comparisonOperator": "equals",
    "propertyValue": "core_feature"
   }
  }
 }
}
```

```
{
 "type": "pastEventCondition",
 "parameterValues": {
  "eventType": "appOpen",
  "numberOfDays": 7,
  "maximumCount": 0,
  "operator": "eventsNotOccurred"
 }
}
```

10.5.3. Customer Data Platform (CDP) Integration



Common CDP scenarios include:

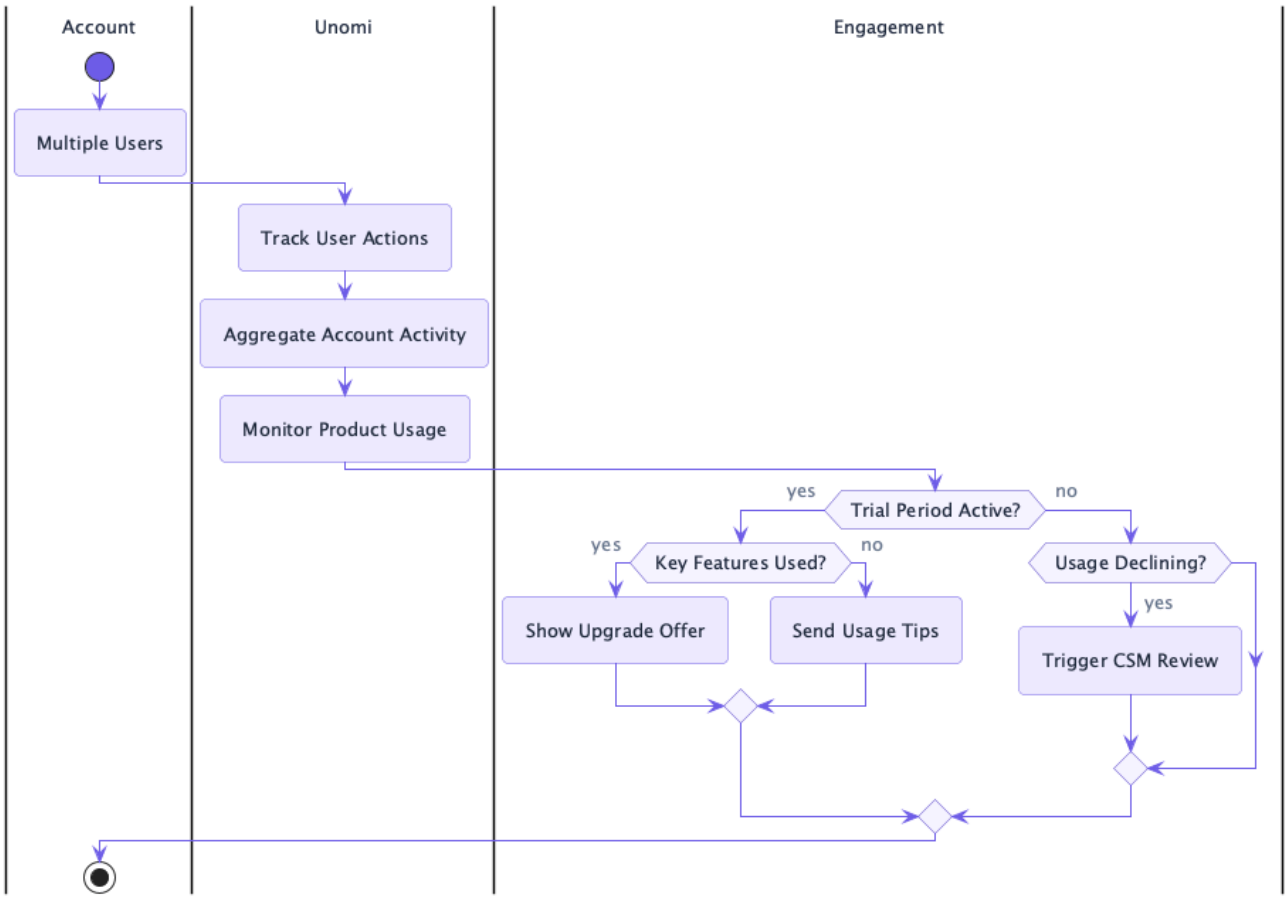
1. **Cross-Channel Customer Journey**
    - Track interactions across touchpoints
    - Build unified customer view
    - Trigger omnichannel campaigns
2. **Customer Support Optimization**
    - Monitor support interactions

- Track issue resolution
- Identify high-value customers

```
{
  "type": "booleanCondition",
  "parameterValues": {
    "operator": "and",
    "subConditions": [
      {
        "type": "pastEventCondition",
        "parameterValues": {
          "eventType": "websiteView",
          "numberOfDays": 30,
          "minimumCount": 1
        }
      },
      {
        "type": "pastEventCondition",
        "parameterValues": {
          "eventType": "mobileAppOpen",
          "numberOfDays": 30,
          "minimumCount": 1
        }
      }
    ]
  }
}
```

```
{
  "type": "pastEventCondition",
  "parameterValues": {
    "eventType": "supportTicket",
    "numberOfDays": 90,
    "minimumCount": 3,
    "propertyConditions": {
      "type": "propertyCondition",
      "parameterValues": {
        "propertyName": "priority",
        "comparisonOperator": "equals",
        "propertyValue": "high"
      }
    }
  }
}
```

10.5.4. B2B Use Cases



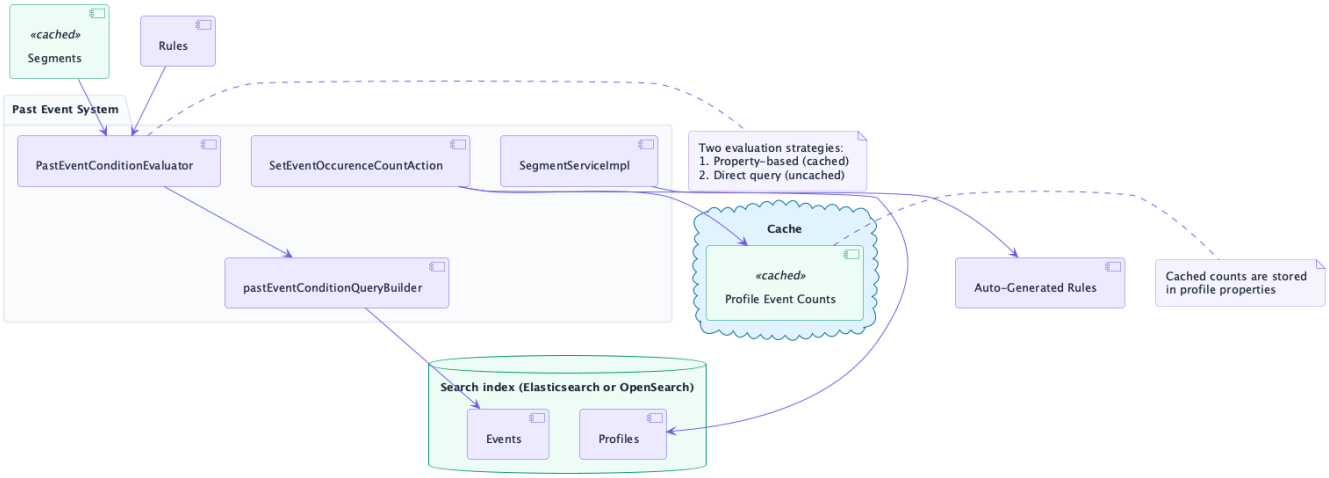
Common B2B scenarios include:

1. **Account Health Monitoring**
    - Track usage across account users
    - Monitor feature adoption
    - Predict churn risk
2. **Product Adoption Tracking**
    - Monitor feature usage frequency
    - Track user onboarding progress
    - Identify power users

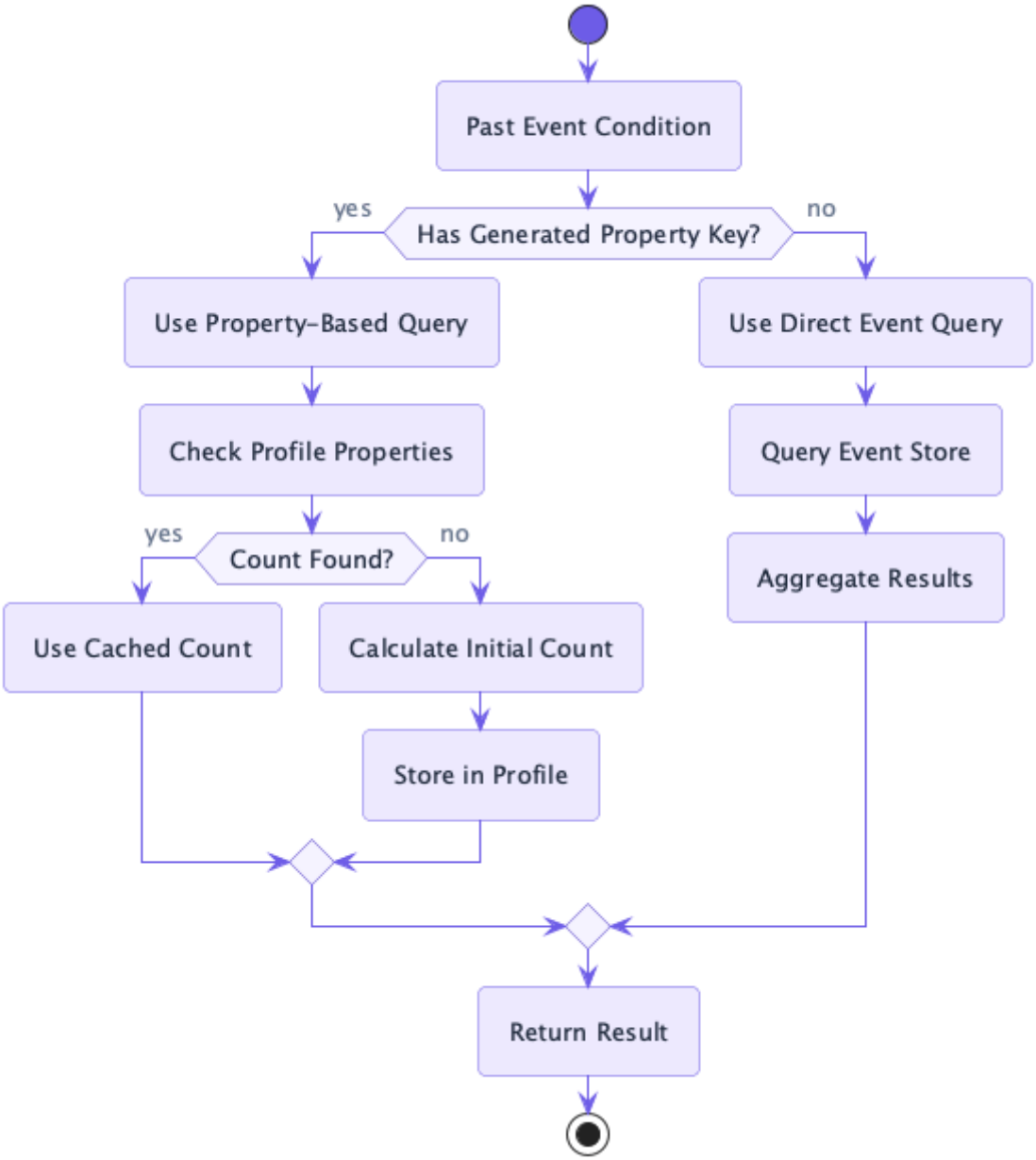
```
{
 "type": "pastEventCondition",
 "parameterValues": {
  "eventType": "login",
  "numberOfDays": 30,
  "maximumCount": 5,
  "propertyConditions": {
   "type": "propertyCondition",
   "parameterValues": {
    "propertyName": "accountId",
    "comparisonOperator": "exists"
   }
  }
 }
}
```

```
{
 "type": "pastEventCondition",
 "parameterValues": {
  "eventType": "featureUsed",
  "numberOfDays": 30,
  "minimumCount": 10,
  "propertyConditions": {
   "type": "propertyCondition",
   "parameterValues": {
    "propertyName": "featureTier",
    "comparisonOperator": "equals",
    "propertyValue": "premium"
   }
  }
 }
}
```

10.5.5. Architecture Overview



Query Strategy Selection

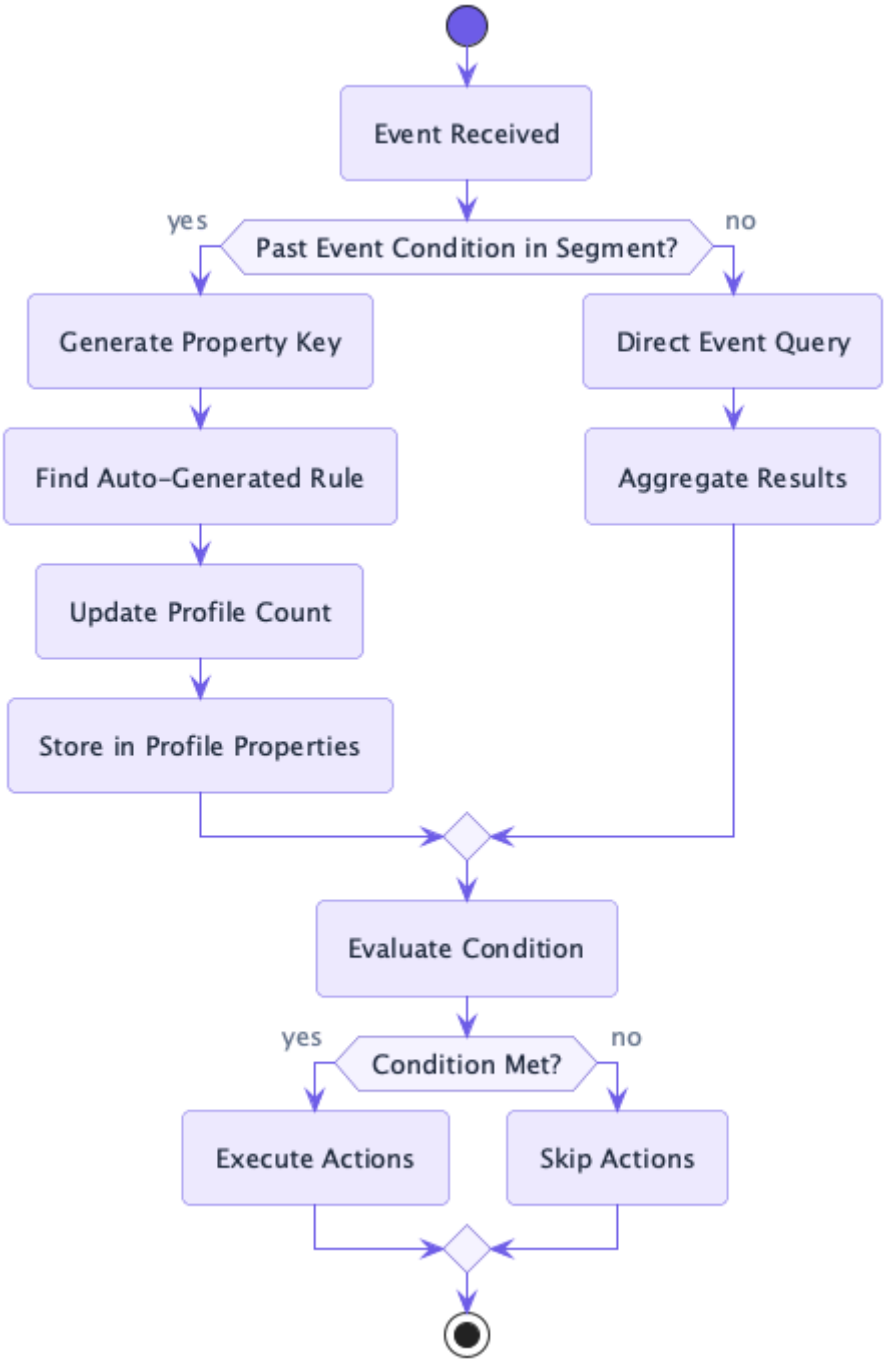


Performance Implications

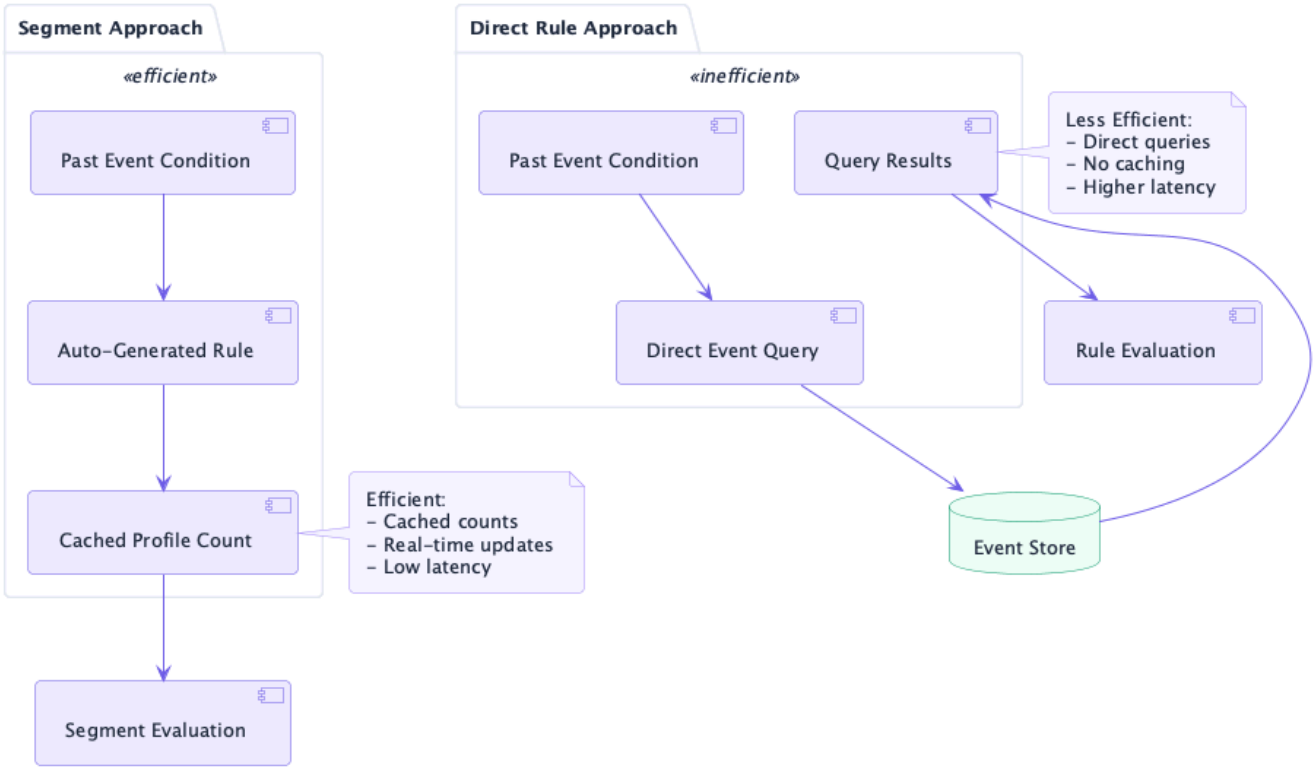
1. **Property-Based Evaluation**
   - Uses cached counts from profile properties
   - Minimal query overhead
   - Suitable for high-frequency conditions
   - Recommended for segments
2. **Direct Query Evaluation**
   - Queries event store directly
   - Higher latency and resource usage
   - No caching mechanism

- Use with caution in rules

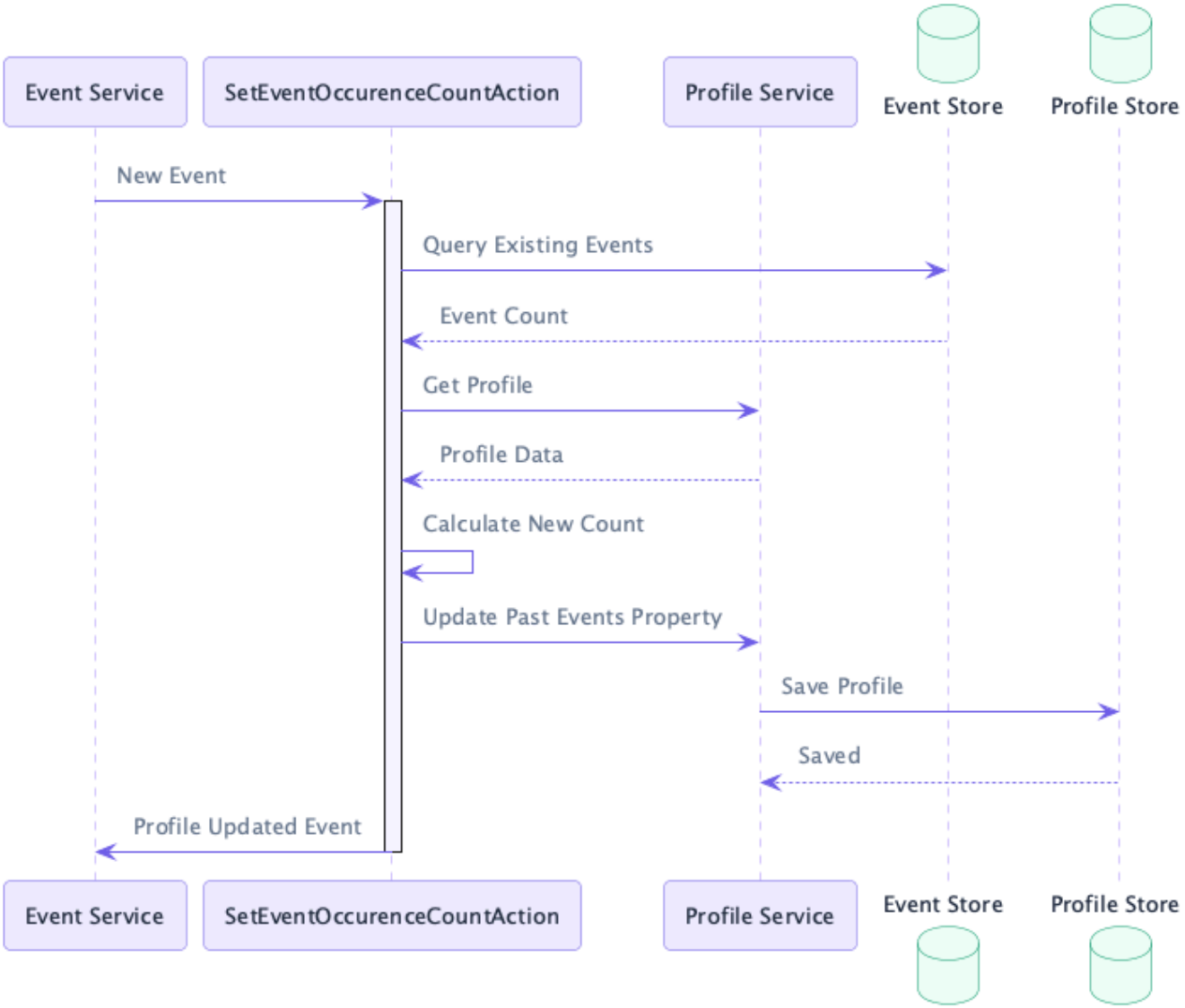
10.5.6. Event Processing Flow



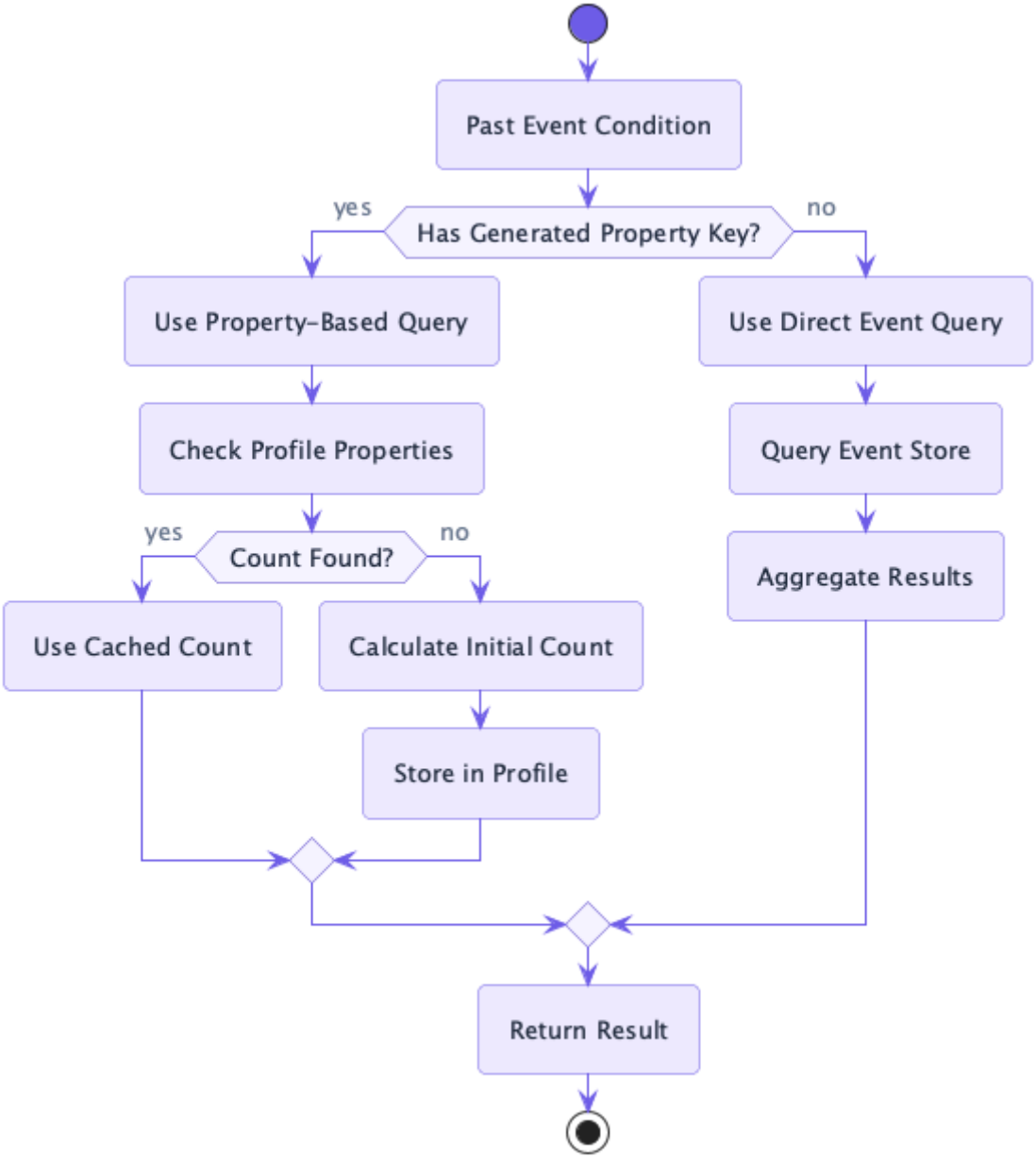
10.5.7. Segment vs Direct Rule Comparison



10.5.8. Cache Update Process



10.5.9. Query Strategy Selection



10.5.10. Usage Patterns

In Segments

When used in segments, past event conditions automatically generate optimization rules that maintain cached event counts on profiles. This is the recommended approach for optimal performance.

Example: Segment with Past Event Condition

```
{
 "metadata": {
  "id": "activeUsers",
  "name": "Active Users",
  "scope": "systemscope"
 },
 "condition": {
  "type": "pastEventCondition",
  "parameterValues": {
   "eventType": "view",
   "numberOfDays": 30,
   "minimumCount": 10
  }
 }
}
```

When this segment is created: 1. The system generates a unique property key for the condition 2. An auto-generated rule is created to maintain event counts 3. Initial profile counts are calculated and stored 4. Subsequent events update the cached counts in real-time

In Rules

Past event conditions can also be used directly in rules, but with important performance considerations:

Example: Rule with Past Event Condition

```
{
 "metadata": {
  "id": "directPastEventRule",
  "name": "Direct Past Event Rule"
 },
 "condition": {
  "type": "pastEventCondition",
  "parameterValues": {
   "eventType": "purchase",
   "numberOfDays": 7,
   "minimumCount": 1
  }
 },
 "actions": [
  {
   "type": "sendEventAction",
   "parameterValues": {
    "eventType": "directRuleTriggered"
   }
  }
 ]
}
```

Performance Implications: 1. No automatic optimization rules are generated 2. Each evaluation requires a direct query to the event store 3. No caching of event counts occurs 4. Higher latency and

resource usage 5. Not recommended for high-frequency conditions

10.5.11. Auto-Generated Rules

The system automatically generates rules for past event conditions through SegmentServiceImpl:

1. **Rule Creation**
    - Generated when segments with past event conditions are created/updated
    - Uses evaluateProfileSegments.json template
    - Includes EvaluateProfileSegmentsAction for segment membership updates
2. **Rule Structure**

```
{
  "metadata": {
    "id": "evaluateProfileSegments",
    "name": "Evaluate segments",
    "description": "Evaluate segments when a profile is modified",
    "readOnly": true
  },
  "condition": {
    "type": "profileUpdatedEventCondition"
  },
  "actions": [
    {
      "type": "evaluateProfileSegmentsAction"
    }
  ]
}
```

+ 3. **Profile Storage Format**

```
{
 "properties": {
  "pastEvents": {
   "past_event_count_abc123_30d_v1": {
    "count": 15,
    "lastUpdated": "2024-03-15T10:30:00Z"
   }
  }
 }
}
```

10.5.12. Core Components

The past event condition system consists of four main components:

1. SetEventOccurenceCountAction - Handles real-time event processing and updates profile event counts

2. PastEventConditionEvaluator - Manages condition evaluation with optimized caching
3. pastEventConditionQueryBuilder - Constructs optimized search index queries
4. SegmentServiceImpl - Manages segments and auto-generated rules

10.5.13. Condition Parameters

Past event conditions support the following parameters as defined in pastEventCondition.json:

1. **Time Window Parameters**
   - numberOfDays (integer) - Rolling window of N days from current time
   - fromDate (date) - Explicit start date for the time window
   - toDate (date) - Explicit end date for the time window
2. **Count Parameters**
   - minimumEventCount (integer) - Minimum number of events required
   - maximumEventCount (integer) - Maximum number of events allowed
   - operator (string) - Either "eventsOccurred" or "eventsNotOccurred"
3. **Event Filter**
   - eventCondition (Condition) - The condition defining which events to count



The eventCondition parameter can **only** accept conditions that are specifically tagged with eventCondition. These are conditions specifically designed to match events. It is NOT allowed to use any other type of condition, including booleanCondition, even if that condition contains valid event conditions.

If you need to combine multiple event conditions, you have two options:

1. Use multiple pastEventCondition`s at a higher level and combine them with a `booleanCondition (as shown in previous examples).
2. Define a custom condition type that uses a booleanCondition in its parentCondition. This is a valid pattern because the condition type itself will be tagged with eventCondition. Example:

```
{
 "metadata": {
  "id": "pageViewEventCondition",
  "name": "pageViewEventCondition",
  "systemTags": [
   "eventCondition",  // This tag makes it valid for use in pastEventCondition
   "event",
   "condition"
  ]
 },
 "parentCondition": {
  "type": "booleanCondition",  // Valid here because it's part of condition type
definition
  "parameterValues": {
   "subConditions": [
    {
     "type": "eventTypeCondition",
     "parameterValues": {
      "eventTypeId": "view"
     }
    },
    {
     "type": "eventPropertyCondition",
     "parameterValues": {
      "propertyName": "target.properties.pageInfo.pagePath",
      "propertyValue": "parameter::pagePath",
      "comparisonOperator": "equals"
     }
    }
   ],
   "operator": "and"
  }
 },
 "parameters": [
  {
   "id": "pagePath",
   "type": "string",
   "multivalued": false
  }
 ]
}
```

This pattern works because the condition type system is hierarchical - a condition type can define a parentCondition that will be evaluated as part of the condition evaluation. The restriction on using booleanCondition applies only to direct usage in the eventCondition parameter, not to the internal structure of properly tagged condition types.

10.5.14. Event Processing System

Real-Time Event Processing

When an event occurs in Unomi, the SetEventOccurenceCountAction processes it in real-time through the following steps:

1. Extracts the past event condition from the action parameters
2. Builds a compound condition for event matching that includes:
    - The original event condition
    - Time window constraints
    - Profile filters
3. Queries existing matching events using optimized search index queries
4. Updates the profile's past event count in system properties
5. Triggers profile update events if configured

The action is defined in setEventOccurenceCountAction.json with the following metadata:

```
{
  "metadata": {
    "id": "setEventOccurenceCountAction",
    "systemTags": [
      "profileTags",
      "demographic"
    ]
  }
}
```

Time Window Processing

The SetEventOccurenceCountAction implements time window processing using the following logic:

```
private boolean inTimeRange(LocalDateTime eventTime, Integer numberOfDays,
                    LocalDateTime fromDate, LocalDateTime toDate) {
   boolean inTimeRange = true;

   if (numberOfDays != null) {
      LocalDateTime now = LocalDateTime.now(ZoneId.of("UTC"));
      if (eventTime.isAfter(now)) {
         inTimeRange = false;
      }
      long daysDiff = Duration.between(eventTime, now).toDays();
      if (daysDiff > numberOfDays) {
         inTimeRange = false;
      }
   }
   if (fromDate != null && fromDate.isAfter(eventTime)) {
      inTimeRange = false;
   }
   if (toDate != null && toDate.isBefore(eventTime)) {
      inTimeRange = false;
   }

   return inTimeRange;
}
```

Profile Storage Format

Past event counts are stored efficiently in the profile's system properties under a pastEvents list. Each entry contains:

- key - A generated unique identifier for the condition using MD5 hashing
- count - The number of matching events for that condition

The update process is handled by:

```
private boolean updatePastEvents(Event event, String generatedPropertyKey, long count) {
    List<Map<String, Object>> existingPastEvents =
        (List<Map<String, Object>>) event.getProfile().getSystemProperties().get("pastEvents");
    if (existingPastEvents == null) {
        existingPastEvents = new ArrayList<>();
        event.getProfile().getSystemProperties().put("pastEvents", existingPastEvents);
    }

    // Update or add count
    for (Map<String, Object> pastEvent : existingPastEvents) {
        if (generatedPropertyKey.equals(pastEvent.get("key"))) {
            if (pastEvent.containsKey("count") && pastEvent.get("count").equals(count)) {
                return false;
            }
            pastEvent.put("count", count);
            return true;
        }
    }

    Map<String, Object> newPastEvent = new HashMap<>();
    newPastEvent.put("key", generatedPropertyKey);
    newPastEvent.put("count", count);
    existingPastEvents.add(newPastEvent);
    return true;
}
```

Evaluation Process

The evaluation process involves multiple components working together to efficiently evaluate past event conditions:

1. **Condition Evaluator** (PastEventConditionEvaluator)
    - Primary evaluation logic for past event conditions
    - Two-phase evaluation strategy:
        - First checks profile properties for cached counts using generatedPropertyKey
        - Falls back to direct event queries only if cached data is unavailable
    - Count validation logic:
        - For eventsOccurred: Verifies count is between minimumEventCount and maximumEventCount
        - For eventsNotOccurred: Verifies count is exactly 0
2. **Query Builder** (pastEventConditionQueryBuilder)
    - Responsible for constructing optimized search index queries
    - Supports two query strategies:
        - Property-based queries: Uses cached counts from profile properties

- Event-based queries: Aggregates events directly when needed

- Partitioned processing support:
  - Configurable through pastEventsDisablePartitions
  - Uses aggregateQueryBucketSize for partition sizing
  - Handles large datasets efficiently

3. **Action Executor** (ActionExecutorDispatcherImpl)
   - Manages the execution of past event condition actions
   - Features:
     - Contextual parameter resolution for dynamic values
     - Performance metrics collection for monitoring
     - Script execution support for complex logic
     - Error handling and logging
   - Execution flow:
     1. Resolves action parameters using context
     2. Validates action configuration
     3. Executes action with performance tracking
     4. Handles results and updates profiles
   - Implementation details:

```
public boolean eval(Condition condition, Item item, Map<String, Object> context,
                    ConditionEvaluatorDispatcher dispatcher) {
    final Map<String, Object> parameters = condition.getParameterValues();
    long count;

    // Try to get count from profile properties first
    if (parameters.containsKey("generatedPropertyKey")) {
        String key = (String) parameters.get("generatedPropertyKey");
        Profile profile = (Profile) item;
        List<Map<String, Object>> pastEvents =
            (ArrayList<Map<String, Object>>) profile.getSystemProperties().get("pastEvents");
        if (pastEvents != null) {
            Number l = (Number) pastEvents
                    .stream()
                    .filter(pastEvent -> pastEvent.get("key").equals(key))
                    .findFirst()
                    .map(pastEvent -> pastEvent.get("count")).orElse(0L);
            count = l.longValue();
        } else {
            count = 0;
        }
    } else {
        // Legacy fallback: direct event query
        count = persistenceService.queryCount(
            pastEventConditionPersistenceQueryBuilder.getEventCondition(
                condition, context, item.getItemId(), definitionsService, scriptExecutor
            ),
            Event.ITEM_TYPE
        );
    }

    // Evaluate count against condition parameters
    boolean eventsOccurred = pastEventConditionPersistenceQueryBuilder
        .getStrategyFromOperator((String) condition.getParameter("operator"));
    if (eventsOccurred) {
        int minimumEventCount = parameters.get("minimumEventCount") == null ?
            0 : (Integer) parameters.get("minimumEventCount");
        int maximumEventCount = parameters.get("maximumEventCount") == null ?
            Integer.MAX_VALUE : (Integer) parameters.get("maximumEventCount");
        return count > 0 && (count >= minimumEventCount && count <= maximumEventCount);
    } else {
        return count == 0;
    }
}
```

10.5.15. Rule Generation

Auto-Generated Rules

The system automatically generates rules for past event conditions through SegmentServiceImpl:

1. **Rule Creation**
    - Generated when segments with past event conditions are created/updated

- Uses evaluateProfileSegments.json template
- Includes EvaluateProfileSegmentsAction for segment membership updates

2. **Rule Structure**

```
{
  "metadata": {
    "id": "evaluateProfileSegments",
    "name": "Evaluate segments",
    "description": "Evaluate segments when a profile is modified",
    "readOnly": true
  },
  "condition": {
    "type": "profileUpdatedEventCondition"
  },
  "actions": [
    {
      "type": "evaluateProfileSegmentsAction"
    }
  ]
}
```

Performance Configuration

Key configuration parameters from SegmentServiceImpl:

```
# Maximum number of IDs to return in a single query
maximumIdsQueryCount = 5000

# Size of each partition for aggregation queries
aggregateQueryBucketSize = 5000

# Whether to disable partitioned processing
pastEventsDisablePartitions = false

# Segment update batch size
segmentUpdateBatchSize = 1000

# Maximum retries for profile segment updates
maxRetriesForUpdateProfileSegment = 3

# Delay between retries (seconds)
secondsDelayForRetryUpdateProfileSegment = 1

# Whether to send profile update events
sendProfileUpdateEventForSegmentUpdate = true
```

10.5.16. Best Practices

Implementation Guidelines

1. **Time Window Selection**
    - Use numberOfDays for rolling windows
    - Use explicit dates for fixed periods
    - Consider data retention policies
    - Account for timezone handling (UTC)
2. **Performance Optimization**
    - Enable partitioning for large datasets
    - Configure appropriate batch sizes
    - Monitor memory usage
    - Use property-based evaluation when possible
3. **Error Handling**
    - Configure appropriate retries
    - Monitor failed updates
    - Implement proper logging
    - Plan for recovery scenarios

Example Configurations

1. **Rolling Window**

```
{
  "type": "pastEventCondition",
  "parameterValues": {
    "eventCondition": {
      "type": "eventTypeCondition",
      "parameterValues": {
        "eventTypeId": "purchase"
      }
    },
    "numberOfDays": 30
  }
}
```

+ 2. **Date Range**

```
{
  "type": "pastEventCondition",
  "parameterValues": {
    "eventCondition": {
      "type": "eventTypeCondition",
      "parameterValues": {
        "eventTypeId": "campaign-click"
      }
    },
    "fromDate": "2024-01-01T00:00:00Z",
    "toDate": "2024-12-31T23:59:59Z"
  }
}
```

10.5.17. Common Use Cases and Examples

Basic Past Event Condition

Example: Count page views in the last 7 days

```
{
  "type": "pastEventCondition",
  "parameterValues": {
    "eventType": "view",
    "numberOfDays": 7,
    "minimumCount": 5,
    "maximumCount": null
  }
}
```

Time-Based Conditions

Example: Count purchases between specific dates

```
{
  "type": "pastEventCondition",
  "parameterValues": {
    "eventType": "purchase",
    "fromDate": "2024-01-01T00:00:00Z",
    "toDate": "2024-03-31T23:59:59Z",
    "minimumCount": 1
  }
}
```

Complex Event Conditions

Example: Count specific product category views with property constraints

```
{
  "type": "pastEventCondition",
  "parameterValues": {
    "eventType": "view",
    "numberOfDays": 30,
    "minimumCount": 3,
    "propertyConditions": {
      "type": "booleanCondition",
      "parameterValues": {
        "operator": "and",
        "subConditions": [
          {
            "type": "propertyCondition",
            "parameterValues": {
              "propertyName": "target.properties.category",
              "comparisonOperator": "equals",
              "propertyValue": "electronics"
            }
          },
          {
            "type": "propertyCondition",
            "parameterValues": {
              "propertyName": "target.properties.price",
              "comparisonOperator": "greaterThan",
              "propertyValue": 100
            }
          }
        ]
      }
    }
  }
}
```

Segment Definition

Example: Segment for active users based on past events

```
{
  "metadata": {
    "id": "activeUsers",
    "name": "Active Users",
    "scope": "systemscope"
  },
  "condition": {
    "type": "booleanCondition",
    "parameterValues": {
      "operator": "and",
      "subConditions": [
        {
          "type": "pastEventCondition",
          "parameterValues": {
            "eventType": "view",
            "numberOfDays": 30,
            "minimumCount": 10
          }
        },
        {
          "type": "pastEventCondition",
          "parameterValues": {
            "eventType": "purchase",
            "numberOfDays": 90,
            "minimumCount": 1
          }
        }
      ]
    }
  }
}
```

Auto-Generated Rule

Example: Auto-generated rule for past event counting

```
{
  "metadata": {
    "id": "auto_past_event_count_rule_12345",
    "name": "Past Event Count Rule for Segment activeUsers",
    "hidden": true
  },
  "condition": {
    "type": "eventTypeCondition",
    "parameterValues": {
      "eventTypeId": "view"
    }
  },
  "actions": [
    {
      "type": "setEventOccurenceCountAction",
      "parameterValues": {
        "pastEventCondition": {
          "eventType": "view",
          "numberOfDays": 30
        }
      }
    }
  ]
}
```

Property Storage Format

Example: Profile property storage format for past event counts

```
{
  "properties": {
    "pastEvents": {
      "past_event_count_abc123_30d_v1": {
        "count": 15,
        "lastUpdated": "2024-03-15T10:30:00Z"
      },
      "past_event_count_def456_90d_v1": {
        "count": 3,
        "lastUpdated": "2024-03-15T10:30:00Z"
      }
    }
  }
}
```

Integration Examples

REST API Usage

Example: Query past event counts via REST API

```
curl -X POST http://localhost:8181/cxs/profiles/query \
 -H 'Content-Type: application/json' \
 -d '{
  "condition": {
   "type": "pastEventCondition",
   "parameterValues": {
    "eventType": "view",
    "numberOfDays": 7,
    "minimumCount": 5
   }
  }
 }'
```

Batch Update Configuration

Example: Configure batch updates for past event counts

```
# Batch processing configuration
segment.pastEventCondition.batchSize=1000
segment.pastEventCondition.threadsCount=4
segment.pastEventCondition.timeoutInSeconds=3600

# Monitoring configuration
segment.pastEventCondition.monitoringEnabled=true
segment.pastEventCondition.alertThreshold=100000
```

10.5.18. Troubleshooting

Common Issues

1. **Performance Problems**
    - Partition Size Issues
        - Symptom: Slow query performance or high memory usage
        - Check aggregateQueryBucketSize configuration (default: 5000)
        - Monitor search engine (Elasticsearch or OpenSearch) heap usage during queries
        - Consider enabling partitioned processing for large datasets
    - Query Optimization
        - Verify proper index settings for event timestamps
        - Check if property-based queries are being used when possible
        - Monitor query execution times through metrics
        - Analyze search engine query patterns
2. **Incorrect Counts**

- Time Window Configuration
  - Verify UTC timezone handling in date calculations
  - Check numberOfDays calculation logic
  - Validate fromDate/toDate format (ISO-8601)
- Event Processing
  - Verify events are being properly persisted
  - Check event type matching in conditions
  - Validate profile ID associations
  - Monitor event indexing status
- Profile Updates
  - Verify pastEvents property structure
  - Check property key generation
  - Monitor profile update events
  - Validate segment evaluation timing

Debugging Tips

1. **Query Verification**
   - Search engine query analysis
     - Use the Elasticsearch or OpenSearch _explain API to analyze queries
     - Monitor query performance through metrics
     - Check query routing and shard distribution
     - Verify index mappings for event fields
2. **Cache Validation**
   - Property Storage
     - Verify pastEvents list structure in profiles
     - Check property key consistency
     - Monitor cache hit/miss ratios
     - Validate property update timestamps
   - Profile Updates
     - Monitor profile update events
     - Check segment evaluation triggers
     - Verify rule execution flow
     - Validate condition evaluation results

10.5.19. Integration Points

Event Flow

The complete event processing flow for past event conditions:

1. **Event Reception**
    - Event arrives through REST API or other channels
    - Event is validated and enriched
    - Profile association is verified
    - Event type is matched against conditions
2. **Real-time Processing**
    - SetEventOccurenceCountAction is triggered
    - Event conditions are evaluated
    - Time windows are calculated
    - Counts are updated in profile
3. **Profile Update**
    - Profile properties are updated
    - Cache entries are invalidated if needed
    - Update events are triggered if configured
    - Optimistic locking handles concurrent updates
4. **Segment Evaluation**
    - Profile segments are re-evaluated
    - New segment memberships are calculated
    - Profile is updated with new segments
    - Related rules are triggered

Segment Evaluation Flow

The detailed segment evaluation process:

1. **Condition Evaluation**
    - Past event conditions are extracted
    - Property-based evaluation is attempted first
    - Falls back to event queries if needed
    - Results are cached when possible

2. **Query Building**
    - Appropriate query strategy is selected
    - Time constraints are applied
    - Partitioning is configured if enabled
    - Optimizations are applied based on context
3. **Result Aggregation**
    - Event counts are aggregated
    - Results are partitioned if needed
    - Memory usage is optimized
    - Results are validated
4. **Profile Update**
    - Profile properties are updated atomically
    - Segment memberships are recalculated
    - Update events are triggered
    - Changes are persisted to storage

10.5.20. Security Considerations

Data Access Controls

1. **Profile Data Protection**
    - Event data access is controlled through permissions
    - Profile properties are protected by scope definitions
    - Sensitive data is filtered from event properties
    - Access to past event counts follows profile permissions
2. **Query Security**
    - Search engine queries are sanitized
    - Input parameters are validated
    - Time ranges are bounded
    - Resource limits are enforced

Resource Protection

1. **Query Limits**
    - Maximum time window restrictions

- Aggregation bucket size limits
- Query timeout configurations
- Memory usage thresholds

2. **Rate Limiting**
   - Profile update frequency limits
   - Event processing rate controls
   - Segment evaluation throttling
   - Cache invalidation controls

Data Integrity

1. **Event Data**
   - Event timestamps are validated
   - Event type verification
   - Profile association validation
   - Property value sanitization
2. **Profile Updates**
   - Optimistic locking for concurrent updates
   - Transaction boundaries
   - Update validation
   - Rollback procedures

Audit Trail

1. **Event Processing**
   - Event processing logs
   - Profile update tracking
   - Segment evaluation history
   - Rule execution records
2. **Error Handling**
   - Failed update logging
   - Security violation tracking
   - Resource limit violations
   - Data integrity issues

10.5.21. Configuration Parameters

Core Settings

```
# Segment Service Configuration
segment.pastEventCondition.aggregateQueryBucketSize=5000
segment.pastEventCondition.maxRetriesForUpdateProfileSegment=3
segment.pastEventCondition.updateBatchSize=1000

# Event Processing
event.pastCondition.maxTimeWindow=365
event.pastCondition.defaultTimeUnit=day

# Query Optimization
query.pastEvent.usePropertyBasedEvaluation=true
query.pastEvent.enablePartitioning=true
query.pastEvent.partitionSize=10000
```

Performance Tuning

```
# Cache Configuration
cache.pastEventCount.timeToLiveInSeconds=3600
cache.pastEventCount.maxEntries=10000

# Query Timeouts
query.pastEvent.timeout=30
query.pastEvent.scrollTimeout=60
query.pastEvent.scrollSize=1000

# Batch Processing
batch.pastEventUpdate.threadPoolSize=4
batch.pastEventUpdate.queueSize=1000
```

Monitoring Configuration

```
# Metrics Collection
metrics.pastEventCondition.enabled=true
metrics.pastEventCondition.detailed=false

# Logging
logging.pastEventCondition.level=INFO
logging.pastEventCondition.detailed=false

# Alerts
alerts.pastEventCondition.slowQueryThreshold=5000
alerts.pastEventCondition.errorThreshold=100
```

10.5.22. Best Practices

Performance Optimization

1. **Query Optimization**
    - Use property-based evaluation when possible
    - Enable partitioning for large datasets
    - Configure appropriate cache sizes
    - Monitor and tune query timeouts
2. **Resource Management**
    - Configure appropriate batch sizes
    - Set reasonable time windows
    - Monitor memory usage
    - Use partitioned processing

Maintenance

1. **Regular Tasks**
    - Monitor cache hit rates
    - Review query performance
    - Check error logs
    - Validate configuration
2. **Troubleshooting**
    - Use detailed logging when needed
    - Monitor metrics
    - Review audit trails
    - Check resource usage

Scaling Considerations

1. **Horizontal Scaling**
    - Configure cluster settings
    - Balance load across nodes
    - Monitor node health
    - Optimize resource allocation
2. **Vertical Scaling**
    - Tune JVM settings
    - Configure memory limits

- Optimize thread pools
- Monitor CPU usage

10.5.23. Evaluation Strategies

The system uses different evaluation strategies depending on how the past event condition is used:

Property-Based Evaluation

Used when a past event condition is part of a segment:

1. **Evaluation Process**
   - Checks for cached count in profile's pastEvents property
   - Uses the generated property key to locate the count
   - Compares count against condition parameters
   - No direct event store query needed

Example Code Flow:

```
if (parameters.containsKey("generatedPropertyKey")) {
   String key = (String) parameters.get("generatedPropertyKey");
   Profile profile = (Profile) item;
   List<Map<String, Object>> pastEvents =
      (ArrayList<Map<String, Object>>) profile.getSystemProperties().get("pastEvents");
   if (pastEvents != null) {
      Number count = (Number) pastEvents
            .stream()
            .filter(pastEvent -> pastEvent.get("key").equals(key))
            .findFirst()
            .map(pastEvent -> pastEvent.get("count")).orElse(0L);
      // Evaluate count against condition parameters
   }
}
```

Direct Event Query Evaluation

Used when a past event condition is used directly in a rule:

1. **Evaluation Process**
   - Constructs a search engine query for matching events
   - Queries event store directly
   - Aggregates results
   - No caching occurs

Example Query Flow:

```
// Direct event store query (less efficient)
count = persistenceService.queryCount(
   pastEventConditionPersistenceQueryBuilder.getEventCondition(
      condition, context, item.getItemId(), definitionsService, scriptExecutor
   ),
   Event.ITEM_TYPE
);
```

10.5.24. Performance Considerations

Segment vs Direct Rule Usage

1. **Segment Usage (Recommended)**
   - Cached event counts in profile properties
   - Real-time updates through auto-generated rules
   - Minimal query overhead during evaluation
   - Efficient for high-frequency conditions
   - Suitable for large-scale segment memberships
2. **Direct Rule Usage (Use with Caution)**
   - No caching mechanism
   - Direct event store queries on each evaluation
   - Higher latency and resource usage
   - Can impact system performance
   - Suitable only for low-frequency rules

Query Optimization

1. **Property-Based Queries**
   - Used when generatedPropertyKey is available
   - Fast profile property lookups
   - Minimal resource usage
   - Example:

```
if (generatedPropertyKey != null) {
   // Use property-based query (efficient)
   return dispatcher.buildFilter(
      getProfileConditionForCounter(
         generatedPropertyKey,
         minimumEventCount,
         maximumEventCount,
         eventsOccurred
      ),
      context
   );
}
```

+ 2. **Event-Based Queries** * Used for direct rule conditions * Requires event store access * Higher resource usage * Example:

```
// Fall back to event-based query (less efficient)
Condition eventCondition = getEventCondition(condition, context, null);
Set<String> ids = getProfileIdsMatchingEventCount(
   eventCondition, minimumEventCount, maximumEventCount
);
```

Resource Usage

1. **Memory Impact**
   - Segment usage: Minimal (only stores count in profile)
   - Direct rule usage: Higher (query results in memory)
2. **CPU Impact**
   - Segment usage: Low (simple property lookup)
   - Direct rule usage: Higher (query execution and aggregation)
3. **Storage Impact**
   - Segment usage: Small profile property overhead
   - Direct rule usage: No additional storage

Best Practices for Performance

1. **Use Segments When Possible**
   - Prefer segments over direct rule usage
   - Let the system generate optimization rules
   - Benefit from built-in caching
2. **Optimize Time Windows**
   - Use reasonable time windows

- Consider data retention policies
- Balance accuracy vs performance

3. **Query Optimization**
   - Enable partitioning for large datasets
   - Configure appropriate batch sizes
   - Monitor query performance
4. **Cache Management**
   - Monitor cache hit rates
   - Configure appropriate cache sizes
   - Regular cache maintenance

Resolution Steps

1. **Initial Analysis**
   - Verify configuration
   - Check query patterns
   - Analyze resource usage
   - Review error logs
2. **Detailed Investigation**
   - Analyze query execution plans
   - Review cache statistics
   - Check system metrics
   - Examine log patterns
3. **Solution Implementation**
   - Apply configuration changes
   - Optimize queries
   - Adjust cache settings
   - Update monitoring rules
4. **Validation**
   - Verify improvements
   - Monitor performance
   - Check error rates
   - Document changes

10.5.25. Maintenance and Monitoring

Regular Maintenance Tasks

1. **Event Count Recalculation**
    - Periodic recalculation of past event counts
    - Updates stale or incorrect counts
    - Example:

```
// Recalculate past event conditions
segmentService.recalculatePastEventConditions();
```

+ 2. **Cache Management** * Monitor cache hit rates * Clean up expired entries * Optimize cache sizes

+ 3. **Performance Monitoring** * Track query execution times * Monitor resource usage * Identify bottlenecks

Monitoring Metrics

1. **Query Performance**
    - Average query execution time
    - Query count by type (property vs direct)
    - Cache hit/miss ratios
2. **Resource Usage**
    - Memory consumption
    - CPU utilization
    - Storage growth

Troubleshooting

1. **Common Issues**
    - Slow query performance
    - High memory usage
    - Incorrect event counts
    - Cache inconsistencies
2. **Diagnostic Tools**
    - Search engine query analysis
    - Performance metrics

- Log analysis
- Cache statistics

3. **Resolution Steps**
   - Verify configuration
   - Check query patterns
   - Analyze resource usage
   - Review error logs

Chapter 11. Queries and aggregations

Apache Unomi contains a query endpoint that is quite powerful. It provides ways to perform queries that can quickly get result counts, apply metrics such as sum/min/max/avg or even use powerful aggregations.

Conditions used in queries follow the same JSON shape described in Conditions guide.

In this section we will show examples of requests that may be built using this API.

11.1. Query counts

Query counts are highly optimized queries that will count the number of objects that match a certain condition without retrieving the results. This can be used for example to quickly figure out how many objects will match a given condition before actually retrieving the results. It uses search engine optimizations (Elasticsearch/OpenSearch) to avoid the cost of loading all the resulting objects.

Here's an example of a query:

```
curl -X POST http://localhost:8181/cxs/query/profile/count \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "parameterValues": {
    "subConditions": [
      {
        "type": "profilePropertyCondition",
        "parameterValues": {
          "propertyName": "systemProperties.isAnonymousProfile",
          "comparisonOperator": "missing"
        }
      },
      {
        "type": "profilePropertyCondition",
        "parameterValues": {
          "propertyName": "properties.nbOfVisits",
          "comparisonOperator": "equals",
          "propertyValueInteger": 1
        }
      }
    ],
    "operator": "and"
  },
  "type": "booleanCondition"
}
EOF
```

The above result will return the profile count of all the profiles

Result will be something like this:

```
2084
```

11.2. Metrics

Metric queries make it possible to apply functions to the resulting property. The supported metrics are:

- sum
- avg
- min
- max

These metrics are supported by both Elasticsearch and OpenSearch backends.

It is also possible to request more than one metric in a single request by concatenating them with a "/" in the URL. Here's an example request that uses the sum and avg metrics:

```
curl -X POST http://localhost:8181/cxs/query/session/profile.properties.nbOfVisits/sum/avg \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "parameterValues": {
    "subConditions": [
      {
        "type": "sessionPropertyCondition",
        "parameterValues": {
          "comparisonOperator": "equals",
          "propertyName": "scope",
          "propertyValue": "digitall"
        }
      }
    ],
    "operator": "and"
  },
  "type": "booleanCondition"
}
EOF
```

The result will look something like this:

```
{
   "_avg":1.0,
   "_sum":9.0
}
```

11.3. Aggregations

Aggregations are a very powerful way to build queries in Apache Unomi that will collect and aggregate data by filtering on certain conditions.

Aggregations are composed of: - an object type and a property on which to aggregate - an aggregation setup (how data will be aggregated, by date, by numeric range, date range or ip range) - a condition (used to filter the data set that will be aggregated)

11.3.1. Aggregation types

Aggregations may be of different types. They are listed here below.

Date

Date aggregations make it possible to automatically generate "buckets" by time periods. The format is compatible with both Elasticsearch and OpenSearch. For more information about the format, you can refer to: - Elasticsearch documentation: Date histogram aggregation (Elasticsearch current) - OpenSearch documentation: Date histogram aggregation (OpenSearch) - OpenSearch documentation: https://opensearch.org/docs/latest/aggregations/bucket/datehistogram/

Here’s an example of a request to retrieve a histogram of by day of all the session that have been create by newcomers (nbOfVisits=1)

```
curl -X POST http://localhost:8181/cxs/query/session/timeStamp \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "aggregate": {
    "type": "date",
    "parameters": {
      "interval": "1d",
      "format": "yyyy-MM-dd"
    }
  },
  "condition": {
    "type": "booleanCondition",
    "parameterValues": {
      "operator": "and",
      "subConditions": [
        {
          "type": "sessionPropertyCondition",
          "parameterValues": {
            "propertyName": "scope",
            "comparisonOperator": "equals",
            "propertyValue": "acme"
          }
        },
        {
          "type": "sessionPropertyCondition",
          "parameterValues": {
            "propertyName": "profile.properties.nbOfVisits",
            "comparisonOperator": "equals",
            "propertyValueInteger": 1
          }
        }
      ]
    }
  }
}
EOF
```

The above request will produce a similar that looks like this:

```
{
  "_all": 8062,
  "_filtered": 4085,
  "2018-10-02": 3,
  "2018-10-03": 17,
  "2018-10-04": 18,
  "2018-10-05": 19,
  "2018-10-06": 23,
  "2018-10-07": 18,
  "2018-10-08": 20
}
```

You can see that we retrieve the count of newcomers aggregated by day.

The interval parameter accepts the following values, in short or named form. It defaults to a month (1M).

Short form	Named form
1s	Second
1m	Minute
1h	Hour
1d	Day
1w	Week
1M	Month
1q	Quarter
1y	Year

Other values, such as 2d or day, are not supported. The optional format parameter sets how the date of each bucket is written in the result keys, as in the yyyy-MM-dd keys above.

Date range

Date ranges make it possible to "bucket" dates, for example to regroup profiles by their birth date as in the example below:

```
curl -X POST http://localhost:8181/cxs/query/profile/properties.birthDate \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "aggregate": {
    "property": "properties.birthDate",
    "type": "dateRange",
    "dateRanges": [
      {
        "key": "After 2009",
        "from": "now-10y/y",
        "to": null
      },
      {
        "key": "Between 1999 and 2009",
        "from": "now-20y/y",
        "to": "now-10y/y"
      },
      {
        "key": "Between 1989 and 1999",
        "from": "now-30y/y",
        "to": "now-20y/y"
      },
      {
        "key": "Between 1979 and 1989",
        "from": "now-40y/y",
        "to": "now-30y/y"
      },
      {
        "key": "Between 1969 and 1979",
        "from": "now-50y/y",
        "to": "now-40y/y"
      },
      {
        "key": "Before 1969",
        "from": null,
        "to": "now-50y/y"
      }
    ]
  },
  "condition": {
    "type": "matchAllCondition",
    "parameterValues": {}
  }
}
EOF
```

The resulting JSON response will look something like this:

```
{
   "_all":4095,
   "_filtered":4095,
   "Before 1969":2517,
   "Between 1969 and 1979":353,
   "Between 1979 and 1989":336,
   "Between 1989 and 1999":337,
   "Between 1999 and 2009":35,
   "After 2009":0,
   "_missing":517
}
```


Each range includes its from date and excludes its to date. Set from or to to null for a range that is open on that side. The same rules apply to numeric and IP ranges.

You can find more information about the date range formats here: - Elasticsearch documentation: Date range aggregation (Elasticsearch current) - OpenSearch documentation: Date range aggregation (OpenSearch) - OpenSearch documentation: https://opensearch.org/docs/latest/aggregations/bucket/daterange/

Numeric range

Numeric ranges make it possible to use "buckets" for the various ranges you want to classify.

Here’s an example of a using numeric range to regroup profiles by number of visits:

```
curl -X POST http://localhost:8181/cxs/query/profile/properties.nbOfVisits \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "aggregate": {
    "property": "properties.nbOfVisits",
    "type": "numericRange",
    "numericRanges": [
      {
        "key": "Less than 5",
        "from": null,
        "to": 5
      },
      {
        "key": "Between 5 and 10",
        "from": 5,
        "to": 10
      },
      {
        "key": "Between 10 and 20",
        "from": 10,
        "to": 20
      },
      {
        "key": "Between 20 and 40",
        "from": 20,
        "to": 40
      },
      {
        "key": "Between 40 and 80",
        "from": 40,
        "to": 80
      },
      {
        "key": "Greater than 100",
        "from": 100,
        "to": null
      }
    ]
  },
  "condition": {
    "type": "matchAllCondition",
    "parameterValues": {}
  }
}
EOF
```

This will produce an output that looks like this:

```
{
  "_all":4095,
  "_filtered":4095,
  "Less than 5":3855,
  "Between 5 and 10":233,
  "Between 10 and 20":7,
  "Between 20 and 40":0,
  "Between 40 and 80":0,
  "Greater than 100":0
}
```


Ranges may overlap: a value is counted in every range it falls in.

IP range

IP ranges make it possible to "bucket" IP addresses, for example to tell internal visitors from external ones. IPv4 and IPv6 addresses are both supported. The property must be mapped as an ip field in the search engine before its data is indexed. No property is mapped that way by default, so add that mapping yourself for the property you aggregate on.

Here's an example that groups sessions by a custom properties.remoteIp property:

```
curl -X POST http://localhost:8181/cxs/query/session/properties.remoteIp \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "aggregate": {
    "property": "properties.remoteIp",
    "type": "ipRange",
    "ipRanges": [
      {
        "key": "Internal",
        "from": "10.0.0.0",
        "to": "11.0.0.0"
      },
      {
        "key": "Loopback",
        "from": "127.0.0.0",
        "to": "128.0.0.0"
      },
      {
        "from": "192.168.0.0",
        "to": "192.169.0.0"
      }
    ]
  },
  "condition": {
    "type": "matchAllCondition",
    "parameterValues": {}
  }
}
EOF
```

This will produce an output that looks like this:

```
{
   "_all":2310,
   "_filtered":2310,
   "Internal":1207,
   "Loopback":12,
   "192.168.0.0-192.169.0.0":340,
   "_missing":64
}
```

A range without a key is reported as from-to, as for the last range above.

11.3.2. Reading the results

The result of an aggregate query is a JSON object that maps each bucket key to the number of items in that bucket. A few keys starting with an underscore hold counts of their own:

_all

The number of items of the requested type in the current tenant. Counting every item takes extra time, so it is not returned when the optimizedQuery=true query parameter is set (as in POST /cxs/query/profile/properties.nbOfVisits?optimizedQuery=true), nor for GET aggregate queries.

_filtered

The number of items that match the condition. Without optimizedQuery=true, it is only returned when the request has a condition.

_missing

The number of matching items that have no value for the aggregated property. It is only returned when there are such items.

The bucket keys depend on the aggregation type:

- When the request has no aggregate, or one without a type, the keys are the values of the property.
- For a date aggregation, the keys are the dates of the buckets, written with the format parameter when it is set. Periods with no items between the first and the last bucket are included, with a count of 0.
- For a range aggregation, the keys are the key of each range. A range without a key is reported as from-to, with for an open side, for example -5.0 or 100.0-*. Every range is reported, even with a count of 0.

A bucket whose key is _all, _filtered or _missing is dropped, because those keys are reserved for the counts above.

11.4. Scroll Queries

Scroll queries are useful when you need to retrieve large result sets efficiently. Instead of retrieving all results at once, which could be memory intensive, scroll queries allow you to retrieve results in batches while maintaining consistency between requests.

11.4.1. Initial Scroll Query

To start a scroll query, use the search endpoint with the following parameters:

```
# Initial scroll query
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "condition": {                  // Optional filtering condition
    "type": "matchAllCondition",
    "parameterValues": {}
  },
  "offset": 0,                    // Starting position
  "limit": 50,                    // Number of items per batch
  "sortby": "property:asc",       // Optional sorting
  "scrollTimeValidity": "1m",     // How long to keep scroll context alive
  "forceRefresh": false           // Whether to force index refresh
}
EOF
```

The response will include: - scrollIdentifier: A unique identifier for continuing the scroll - list: The first batch of results - totalSize: Total number of matching items - offset: Current position in the result set - pageSize: Size of the current batch

11.4.2. Continue Scroll Query

To retrieve subsequent batches, only the scroll identifier and validity are needed:

```
# Retrieve next batch
curl -X POST http://localhost:8181/cxs/profiles/search \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "scrollIdentifier": "YOUR_SCROLL_ID",    // Required: the scroll ID from previous response
  "scrollTimeValidity": "1m"               // Required: extends scroll context lifetime
}
EOF
```



When continuing a scroll query, you do not need to resend other query parameters (condition, sortby, etc.). The scroll context maintains the original query state.

11.4.3. Query Parameters

Initial query parameters: - condition: The filtering condition (optional) - offset: Starting position in results - limit: Number of results per batch - sortby: Property and direction for sorting (optional) - scrollTimeValidity: How long to keep the scroll context alive (e.g., "1m") - forceRefresh: Whether to force an index refresh before query (default: false)

Continuation query parameters: - scrollIdentifier: The scroll ID returned from the previous request - scrollTimeValidity: How long to extend the scroll context

Chapter 12. Profile import & export

The profile import and export feature in Apache Unomi is based on configurations and consumes or produces CSV files that contain profiles to be imported and exported.

12.1. Importing profiles

Only ftp, sftp, ftps and file are supported in the source path. For example:

```
file:///tmp/?fileName=profiles.csv&move=.done&consumer.delay=25s
```

Where:

- fileName Can be a pattern, for example include=.*.csv instead of fileName=... to consume all CSV files. By default the processed files are moved to .camel folder you can change it using the move option.
- consumer.delay Is the frequency of polling in milliseconds. For example, 20000 milliseconds is 20 seconds. This frequency can also be 20s. Other possible format are: 2h30m10s = 2 hours and 30 minutes and 10 seconds.

See http://camel.apache.org/ftp.html and http://camel.apache.org/file2.html to build more complex source path. Also be careful with FTP configuration as most servers no longer support plain text FTP and you should use SFTP or FTPS instead, but they are a little more difficult to configure properly. It is recommended to test the connection with an FTP client first before setting up these source paths to ensure that everything works properly. Also on FTP connections most servers require PASSIVE mode so you can specify that in the path using the passiveMode=true parameter.

Here are some examples of FTPS and SFTP source paths:

```
sftp://USER@HOST/PATH?password=PASSWORD&include=.*.csv
ftps://USER@HOST?password=PASSWORD&fileName=profiles.csv&passiveMode=true
```

Where:

- USER is the user name of the SFTP/FTPS user account to login with
- PASSWORD is the password for the user account
- HOST is the host name (or IP address) of the host server that provides the SFTP / FTPS server
- PATH is a path to a directory inside the user's account where the file will be retrieved.

12.1.1. Import API

Apache Unomi provides REST endpoints to manage import configurations:

```
GET /cxs/importConfiguration
GET /cxs/importConfiguration/{configId}
POST /cxs/importConfiguration
DELETE /cxs/importConfiguration/{configId}
```

This is how a oneshot import configuration looks like:

```
{
   "itemId": "importConfigId",
   "itemType": "importConfig",
   "name": "Import Config Sample",
   "description": "Sample description",
   "configType": "oneshot",       //Config type can be 'oneshot' or 'recurrent'
   "properties": {
      "mapping": {
         "email": 0,              //<Apache Unomi Property Id> : <Column Index In the CSV>
         "firstName": 2,
         ...
      }
   },
   "columnSeparator": ",",        //Character used to separate columns
   "lineSeparator": "\\n",        //Character used to separate lines (\n or \r)
   "multiValueSeparator": ";",    //Character used to separate values for multivalued columns
   "multiValueDelimiter": "[]",   //Character used to wrap values for multivalued columns
   "status": "SUCCESS",           //Status of last execution
   "executions": [                //(RETURN) Last executions by default only last 5 are returned
      ...
   ],
   "mergingProperty": "email",        //Apache Unomi Property Id used to check duplicates
   "overwriteExistingProfiles": true,  //Overwrite profiles that have duplicates
   "propertiesToOverwrite": "firstName, lastName, ...",      //If last is set to true, which property to
overwrite, 'null' means overwrite all
   "hasHeader": true,                //CSV file to import contains a header line
   "hasDeleteColumn": false          //CSV file to import doesn't contain a TO DELETE column (if it
contains, will be the last column)
}
```

A recurrent import configuration is similar to the previous one with some specific information to add to the JSON like:

```
{
  ...
  "configType": "recurrent",
  "properties": {
    "source":
"ftp://USER@SERVER[:PORT]/PATH?password=xxx&fileName=profiles.csv&move=.done&consumer.
delay=20000",
            // Only 'ftp', 'sftp', 'ftps' and 'file' are supported in the 'source' path
            // eg. file:///tmp/?fileName=profiles.csv&move=.done&consumer.delay=25s
            // 'fileName' can be a pattern eg 'include=.*.csv' instead of 'fileName=...' to consume all CSV
files
            // By default the processed files are moved to '.camel' folder you can change it using the
'move' option
            // 'consumer.delay' is the frequency of polling. '20000' (in milliseconds) means 20 seconds.
Can be also '20s'
            // Other possible format are: '2h30m10s' = 2 hours and 30 minutes and 10 seconds
    "mapping": {
      ...
    }
  },
  ...
  "active": true,  //If true the polling will start according to the 'source' configured above
  ...
}
```

12.2. Exporting profiles

Only ftp, sftp, ftps and `file are supported in the source path. For example:

```
file:///tmp/?fileName=profiles-export-${date:now:yyyyMMddHHmm}.csv&fileExist=Append)
sftp://USER@HOST/PATH?password=PASSWORD&binary=true&fileName=profiles-export-
${date:now:yyyyMMddHHmm}.csv&fileExist=Append
ftps://USER@HOST?password=PASSWORD&binary=true&fileName=profiles-export-
${date:now:yyyyMMddHHmm}.csv&fileExist=Append&passiveMode=true
```

As you can see in the examples above, you can inject variables in the produced file name ${date:now:yyyyMMddHHmm} is the current date formatted with the pattern yyyyMMddHHmm. fileExist option put as Append will tell the file writer to append to the same file for each execution of the export configuration. You cam omit this option to write a profile per file.

See http://camel.apache.org/ftp.html and http://camel.apache.org/file2.html to build more complex destination path.

12.2.1. Export API

Apache Unomi provides REST endpoints to manage export configurations:

```
GET /cxs/exportConfiguration
GET /cxs/exportConfiguration/{configId}
POST /cxs/exportConfiguration
DELETE /cxs/exportConfiguration/{configId}
```

This is how a oneshot export configuration looks like:

```
{
  "itemId": "exportConfigId",
  "itemType": "exportConfig",
  "name": "Export configuration sample",
  "description": "Sample description",
  "configType": "oneshot",
  "properties": {
    "period": "2m30s",
    "segment": "contacts",
    "mapping": {
      "0": "firstName",
      "1": "lastName",
      ...
    }
  },
  "columnSeparator": ",",
  "lineSeparator": "\\n",
  "multiValueSeparator": ";",
  "multiValueDelimiter": "[]",
  "status": "RUNNING",
  "executions": [
    ...
  ]
}
```

A recurrent export configuration is similar to the previous one with some specific information to add to the JSON like:

```
{
   ...
   "configType": "recurrent",
   "properties": {
   "destination": "sftp://USER@SERVER:PORT/PATH?password=XXX&fileName=profiles-export-
${date:now:yyyyMMddHHmm}.csv&fileExist=Append",
   "period": "2m30s",      //Same as 'consumer.delay' option in the import source path
   "segment": "contacts",  //Segment ID to use to collect profiles to export
   "mapping": {
      ...
      }
   },
   ...
   "active": true,         //If true the configuration will start polling upon save until the user deactivate
it
   ...
}
```

12.3. Configuration in details

First configuration you need to change would be the configuration type of your import / export feature (code name router) in the etc/unomi.custom.system.properties file (creating it if necessary):

```
#Configuration Type values {'nobroker', 'kafka'}
org.apache.unomi.router.config.type=nobroker
```

By default the feature is configured (as above) to use no external broker, which means to handle import/export data it will use in memory queues (In the same JVM as Apache Unomi). If you are clustering Apache Unomi, most important thing to know about this type of configuration is that each Apache Unomi will handle the import/export task by itself without the help of other nodes (No Load-Distribution).

Changing this property to kafka means you have to provide the Apache Kafka configuration, and in the opposite of the nobroker option import/export data will be handled using an external broker (Apache Kafka), this will lighten the burden on the Apache Unomi machines.

You may use several Apache Kafka instance, 1 per N Apache Unomi nodes for better application scaling.

To enable using Apache Kafka you need to configure the feature as follows:

```
#Configuration Type values {'nobroker', 'kafka'}
org.apache.unomi.router.config.type=kafka
```

Uncomment and update Kafka settings to use Kafka as a broker

```
#Kafka
org.apache.unomi.router.kafka.host=localhost
org.apache.unomi.router.kafka.port=9092
org.apache.unomi.router.kafka.import.topic=import-deposit
org.apache.unomi.router.kafka.export.topic=export-deposit
org.apache.unomi.router.kafka.import.groupId=unomi-import-group
org.apache.unomi.router.kafka.export.groupId=unomi-import-group
org.apache.unomi.router.kafka.consumerCount=10
org.apache.unomi.router.kafka.autoCommit=true
```

There is couple of properties you may want to change to fit your needs, one of them is the **import.oneshot.uploadDir which** will tell Apache Unomi where to store temporarily the CSV files to import in Oneshot mode, it's a technical property to allow the choice of the convenient disk space where to store the files to be imported. It defaults to the following path under the Apache Unomi Karaf (It is recommended to change the path to a more convenient one).

```
#Import One Shot upload directory
org.apache.unomi.router.import.oneshot.uploadDir=${karaf.data}/tmp/unomi_oneshot_import_conf
igs/
```

Next two properties are max sizes for executions history and error reports, for some reason you don't want Apache Unomi to report all the executions history and error reports generated by the executions of an import/export configuration. To change this you have to change the default values of these properties.

```
#Import/Export executions history size
org.apache.unomi.router.executionsHistory.size=5
```

```
#errors report size
org.apache.unomi.router.executions.error.report.size=200
```

Next one is about the allowed endpoints you can use when building the source or destionation path, as mentioned above we can have a path of type file, ftp, ftps, sftp. You can make it less if you want to omit some endpoints (eg. you don't want to permit the use of non secure FTP).

```
#Allowed source endpoints
org.apache.unomi.router.config.allowedEndpoints=file,ftp,sftp,ftps
```

Final ones say where a file endpoint may resolve. A recurrent import source or export destination using the file scheme is refused unless it resolves inside one of these base directories, under the directory named after its tenant ({baseDir}/{tenantId}), at any depth, and so is any of its options that names a path (fileName, move, moveFailed, ...). The same applies to localWorkDirectory, whatever the scheme, since that one names a local directory even on a remote endpoint. Each property accepts a comma-separated list. A tenant cannot point a file endpoint at another tenant's directory. An endpoint that carries a Camel property placeholder ({{...}}) is refused whatever its scheme, since Camel expands it after the endpoint

has been validated.

```
#Base directories a file endpoint may resolve into
org.apache.unomi.router.config.import.baseDir=${karaf.data}/router/import/
org.apache.unomi.router.config.export.baseDir=${karaf.data}/router/export/
```

Import and export are kept apart on purpose: with a single shared directory, an export writing a .csv could be picked up by an import route polling the same place. Point both properties at the same directory only if that is what you mean. A configuration that resolves outside them builds no route and is marked INVALID_ENDPOINT; restoring the directories clears that mark on its own at the next rebuild.

Chapter 13. Consent management

13.1. Consent API

Starting with Apache Unomi 1.3, a new API for consent management is now available. This API is designed to be able to store/retrieve/update visitor consents in order to comply with new privacy regulations such as the GDPR.

13.1.1. Profiles with consents

Visitor profiles now contain a new Consent object that contains the following information:

- a scope
- a type identifier for the consent. This can be any key to reference a consent. Note that Unomi does not manage consent definitions, it only stores/retrieves consents for each profile based on this type
- a status : GRANT, DENY or REVOKED
- a status date (the date at which the status was updated)
- a revocation date, in order to comply with GDPR this is usually set at two years

Consents are stored as a sub-structure inside a profile. To retrieve the consents of a profile you can simply retrieve a profile with the following request:

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d @- <<'EOF'
{
   "source": {
      "itemId":"homepage",
      "itemType":"page",
      "scope":"example"
   }
}
EOF
```

Here is an example of a response with a Profile with a consent attached to it:

```
{
  "profileId": "18afb5e3-48cf-4f8b-96c4-854cfaadf889",
  "sessionId": "1234",
  "profileProperties": null,
  "sessionProperties": null,
  "profileSegments": null,
  "filteringResults": null,
  "personalizations": null,
  "trackedConditions": [],
  "anonymousBrowsing": false,
  "consents": {
    "example/newsletter": {
      "scope": "example",
      "typeIdentifier": "newsletter",
      "status": "GRANTED",
      "statusDate": "2018-05-22T09:27:09Z",
      "revokeDate": "2020-05-21T09:27:09Z"
    }
  }
}
```


It is of course possible to have multiple consents defined for a single visitor profile.

13.1.2. Consent type definitions

Apache Unomi does not manage consent definitions, it leaves that to an external system (for example a CMS) so that it can handle user-facing UIs to create, update, internationalize and present consent definitions to end users.

The only thing that is import to Apache Unomi to manage visitor consents is a globally unique key, that is called the consent type.

13.1.3. Creating / update a visitor consent

A new built-in event type called "modifyConsent" can be sent to Apache Unomi to update a consent for the current profile.

Here is an example of such an event:

```
{
 "events": [
  {
   "scope": "example",
   "eventType": "modifyConsent",
   "source": {
    "itemType": "page",
    "scope": "example",
    "itemId": "anItemId"
   },
   "target": {
    "itemType": "anyType",
    "scope": "example",
    "itemId": "anyItemId"
   },
   "properties": {
    "consent": {
     "typeIdentifier": "newsletter",
     "scope": "example",
     "status": "GRANTED",
     "statusDate": "2018-05-22T09:27:09.473Z",
     "revokeDate": "2020-05-21T09:27:09.473Z"
    }
   }
  }
 ]
}
```

You could send it using the following curl request:

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d @- <<'EOF'
{
   "source":{
      "itemId":"homepage",
      "itemType":"page",
      "scope":"example"
   },
   "events": [
      {
         "scope":"example",
         "eventType":"modifyConsent",
         "source":{
            "itemType":"page",
            "scope":"example",
            "itemId":"anItemId"
         },
         "target":{
            "itemType":"anyType",
            "scope":"example",
            "itemId":"anyItemId"},
         "properties":{
            "consent":{
               "typeIdentifier":"newsletter",
               "scope":"example",
               "status":"GRANTED",
               "statusDate":"2018-05-22T09:27:09.473Z",
               "revokeDate":"2020-05-21T09:27:09.473Z"
            }
         }
      }
   ]
}
EOF
```

13.1.4. How it works (internally)

Upon receiving this event, Apache Unomi will trigger the modifyAnyConsent rule that has the following definition:

```
{
 "metadata" : {
  "id": "modifyAnyConsent",
  "name": "Modify any consent",
  "description" : "Modify any consent and sets the consent in the profile",
  "readOnly":true
 },

 "condition" : {
  "type": "modifyAnyConsentEventCondition",
  "parameterValues": {
  }
 },

 "actions" : [
  {
   "type": "modifyConsentAction",
   "parameterValues": {
   }
  }
 ]

}
```

As we can see this rule is pretty simple it will simply execute the modifyConsentAction that is implemented by the ModifyConsentAction Java class

This class will update the current visitor profile to add/update/revoke any consents that are included in the event.

Chapter 14. Privacy management

Apache Unomi provides an endpoint to manage visitor privacy. You will find in this section information about what it includes as well as how to use it.

14.1. Setting up access to the privacy endpoint

The privacy endpoint is a bit special, because despite being protected by basic authentication as the rest of the REST API is is actually designed to be available to end-users.

So in effect it should usually be proxied so that public internet users can access the endpoint but the proxy should also check if the profile ID wasn't manipulated in some way.

Apache Unomi doesn't provide (for the moment) such a proxy, but basically it should do the following:

1. check for potential attack activity (could be based on IDS policies or even rate detection), and at the minimum check that the profile ID cookie seems authentic (for example by checking that it is often coming from the same IP or the same geographic location)
2. proxy to /cxs/privacy

14.2. Anonymizing a profile

It is possible to anonymize a profile, meaning it will remove all "identifying" property values from the profile. Basically all properties with the tag personalIdentifierProperties will be purged from the profile.



In Unomi 4.0+, /cxs/privacy/* endpoints require tenant or administrator authentication. The examples below use a tenant private key; you may also proxy these endpoints and validate the profile cookie server-side as described above.

Here's an example of a request to anonymize a profile:

```
curl -X POST http://localhost:8181/cxs/privacy/profiles/{profileID}/anonymize?scope=ASCOPE \
  --user "TENANT_ID:PRIVATE_KEY"
```

where {profileID} must be replaced by the actual identifier of a profile and ASCOPE must be replaced by a scope identifier.

14.3. Downloading profile data

It is possible to download the profile data of a user. This will only download the profile for a user using the specified ID as a cookie value.

Warning: this operation can also be sensitive so it would be better to protected with a proxy that can perform some validation on the requests to make sure no one is trying to download a profile using some kind of "guessing" of profile IDs.

```
curl -X GET http://localhost:8181/cxs/client/myprofile/json \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
--cookie "context-profile-id=PROFILE-ID"
```

where the last path segment is the format (json, csv, yaml or text), PROFILE-ID is the profile identifier for which to download the profile, and YOUR_PUBLIC_API_KEY is the tenant public API key (required on Unomi 4.0 public /cxs/client/* endpoints unless single-tenant compatibility mode is enabled).

In single-tenant compatibility mode a plain link on the website to this endpoint also works, as a top-level navigation; see Websites on other origins.

14.4. Deleting a profile

It is possible to delete a profile, but this works a little differently than you might expect. In all cases the data contained in the profile will be completely erased. If the withData optional flag is set to true, all past event and session data will also be detached from the current profile and anonymized.

```
curl -X DELETE http://localhost:8181/cxs/privacy/profiles/{profileID}?withData=false \
--user "TENANT_ID:PRIVATE_KEY"
```



Replace TENANT_ID and PRIVATE_KEY with your actual tenant ID and private API key. Only the Tenant API (/cxs/tenants) uses system administrator authentication (the karaf user and your configured UNOMI_ROOT_PASSWORD).

where {profileID} must be replaced by the actual identifier of a profile and the withData specifies whether the data associated with the profile must be anonymized or not

14.5. Related

You might also be interested in the Consent API section that describe how to manage profile consents.

Chapter 15. Cluster setup

15.1. Cluster setup

Apache Unomi runs as one or more Karaf nodes that share the same Elasticsearch or OpenSearch backend. Since Unomi 3.0, cluster coordination uses the **persistence layer** (UNOMI-877): each node registers a ClusterNode record, publishes periodic heartbeats, and reads peer state from the search index. Karaf Cellar / Hazelcast are no longer required.

This design also supports the task scheduler (cluster-wide locks, executor nodes) and the health-check **cluster** provider.

15.1.1. Required settings

Configure each node in etc/unomi.custom.system.properties (or via environment variables):

```
org.apache.unomi.cluster.public.address=${env:UNOMI_CLUSTER_PUBLIC_ADDRESS:-
http://localhost:8181}
org.apache.unomi.cluster.internal.address=${env:UNOMI_CLUSTER_INTERNAL_ADDRESS:-
https://localhost:9443}
org.apache.unomi.cluster.nodeId=${env:UNOMI_CLUSTER_NODEID:-}
org.apache.unomi.cluster.nodeName=${env:UNOMI_CLUSTER_NODENAME:-}
org.apache.unomi.cluster.nodeStatisticsUpdateFrequency=${env:UNOMI_CLUSTER_NODESTATISTIC
S_UPDATEFREQUENCY:-10000}
```

- **public.address** — URL other nodes and operators use to reach this node (HTTP, port 8181 by default).
- **internal.address** — HTTPS URL for the private REST API (port 9443 by default).
- **nodeId** — Optional. Identifies the node in the cluster and in the scheduler; when it is not set, Unomi generates one (see Node identity). When you set it, it **must be unique** in the cluster.
- **nodeName** — Optional user friendly name, shown next to the id. Defaults to the host name.
- **nodeStatisticsUpdateFrequency** — Interval in milliseconds between heartbeat / statistics updates (default 10 seconds).

15.1.2. Node identity

Every node needs an id that no other node uses and that it keeps across restarts: the id owns the node's cluster record and its scheduler task locks. Nothing has to be configured for this.

- On its first start a node generates an id made of its name and a random suffix, for example web-1-3f9a2c1b, and stores it with its name in etc/org.apache.unomi.node.cfg. Later starts reuse it. A second node, container or copy of the image generates its own, so a single node grows into a cluster without any change.
- UNOMI_CLUSTER_NODEID (org.apache.unomi.cluster.nodeId) sets the id explicitly. It replaces the stored id and is stored in turn, so the node keeps it after the variable is removed. Unomi does not

check that an id you set is unique: two nodes with the same id share one cluster record and the same task locks.

- The name comes from UNOMI_CLUSTER_NODENAME (org.apache.unomi.cluster.nodeName), or from the host name. It is read again at every start: renaming a node does not change its id.
- The scheduler uses the same id.

The stored id lives in the node's etc/ directory and is lost with it, for example when a container is recreated. The node then joins under a new id; the old cluster record is removed once its heartbeat is stale and its task locks expire. To give a node its previous id back, start it once with UNOMI_CLUSTER_NODEID set to that id (it is in the node's log at startup and in the cluster REST endpoint). To make a node take a new id, unset UNOMI_CLUSTER_NODEID, blank nodeId in etc/org.apache.unomi.node.cfg and restart the node; do not delete the file, Unomi could then no longer save the id there.

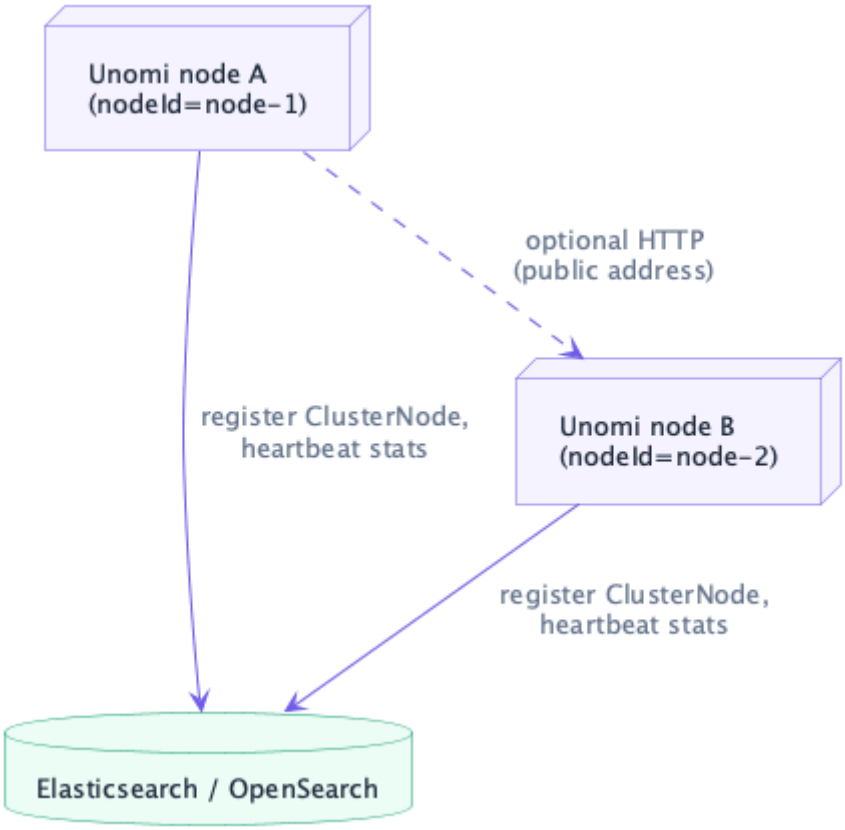
Copying an installation together with its etc/ directory (a cloned virtual machine, an image built from a node that already ran, an etc/ volume shared by several containers) copies the id too: blank it on the copies.

15.1.3. How it works

On startup the cluster service:

1. Registers this node in persistence (ClusterNode item).
2. Schedules a recurring task to update CPU, load, and uptime statistics. It is the node's heartbeat and runs on every node, including those with scheduler.executorNode=false; the record also says whether the node runs scheduled tasks.
3. Schedules stale-node cleanup (nodes with no heartbeat for roughly three times the update interval are removed).

Other nodes see the same registry through the shared search backend. The health-check extension reports cluster size and per-node status when you call /health/check (see Health Check extension).



15.1.4. Scheduler and executor nodes

Background tasks can be marked **persistent** (stored in the index, visible on all nodes) or **local** (single node only). Only nodes that run the task executor process scheduled work.

- Use the Karaf command unomi:task-executor to see or toggle executor mode on a node.
- See Task scheduler for REST and shell operations on tasks.

In a typical cluster you run multiple Unomi nodes with **distinct nodeId values**, all pointing at the same search backend, and enable the executor on every node that should run background jobs (or on a subset for dedicated worker nodes).

15.1.5. Operations

- **Inspect nodes** — GET /cxs/cluster (private API, system administrator credentials). Listed in Useful Apache Unomi URLs.
- **Health** — GET /health/check includes a cluster provider when the health-check feature is installed.
- **Stale nodes** — Removed automatically when heartbeats stop; adjust nodeStatisticsUpdateFrequency if your network is slow or nodes are under heavy load.

15.1.6. Migration from Unomi 2.x / Cellar

If you previously used Karaf Cellar for clustering, plan a full migration to persistence-based clustering as part of your Unomi 3 upgrade. See JIRA reference (UNOMI-877) and Karaf upgrade for platform constraints.

Chapter 16. Task scheduler

16.1. Cluster-aware task scheduler

Apache Unomi 4.0 introduces a built-in **task scheduler** for background work that must run reliably in single-node and clustered deployments.

Use it when you need to:

- Run maintenance jobs on a timer (cache refresh, purge, segment recalculation).
- Offload heavy work from a request or rule action (for example profile merge follow-up).
- Coordinate work across cluster nodes with locks, recovery, and optional persistence.
- Inspect, retry, or cancel tasks through the REST API or Karaf shell.

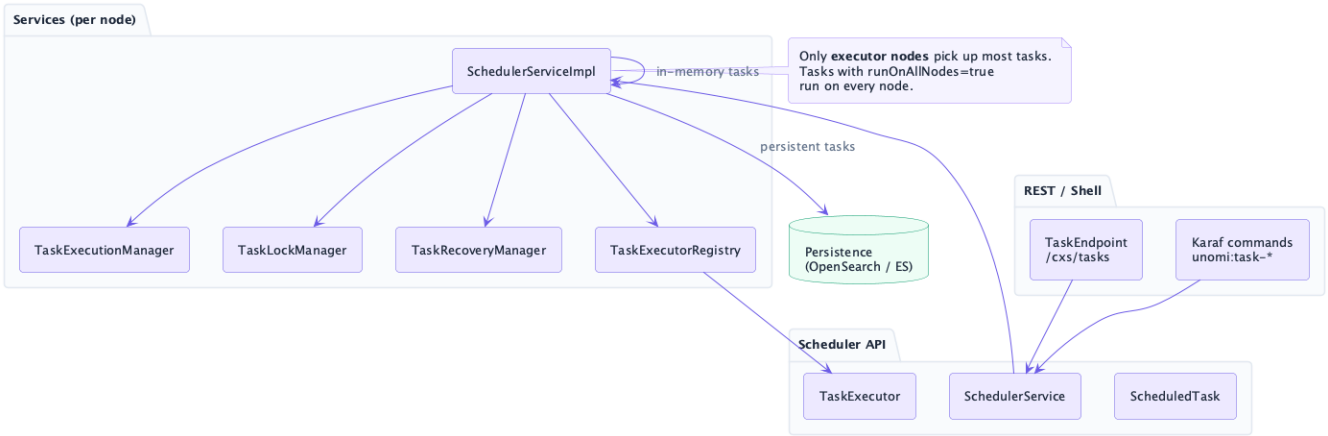
Tasks are identified by a **task type** string. Each type is handled by a registered TaskExecutor implementation (in core services or in a plugin).

16.1.1. Quick example: periodic cache refresh

Core services already use the scheduler for in-memory cache refresh. The pattern looks like this:

```
schedulerService.newTask("cache-refresh-segment")
    .nonPersistent()                // local to this node; not stored in OpenSearch/Elasticsearch
    .withPeriod(1, TimeUnit.SECONDS)
    .withFixedDelay()               // wait for completion before next run
    .disallowParallelExecution()    // skip overlapping runs
    .withSimpleExecutor(() -> refreshSegmentCache())
    .schedule();
```

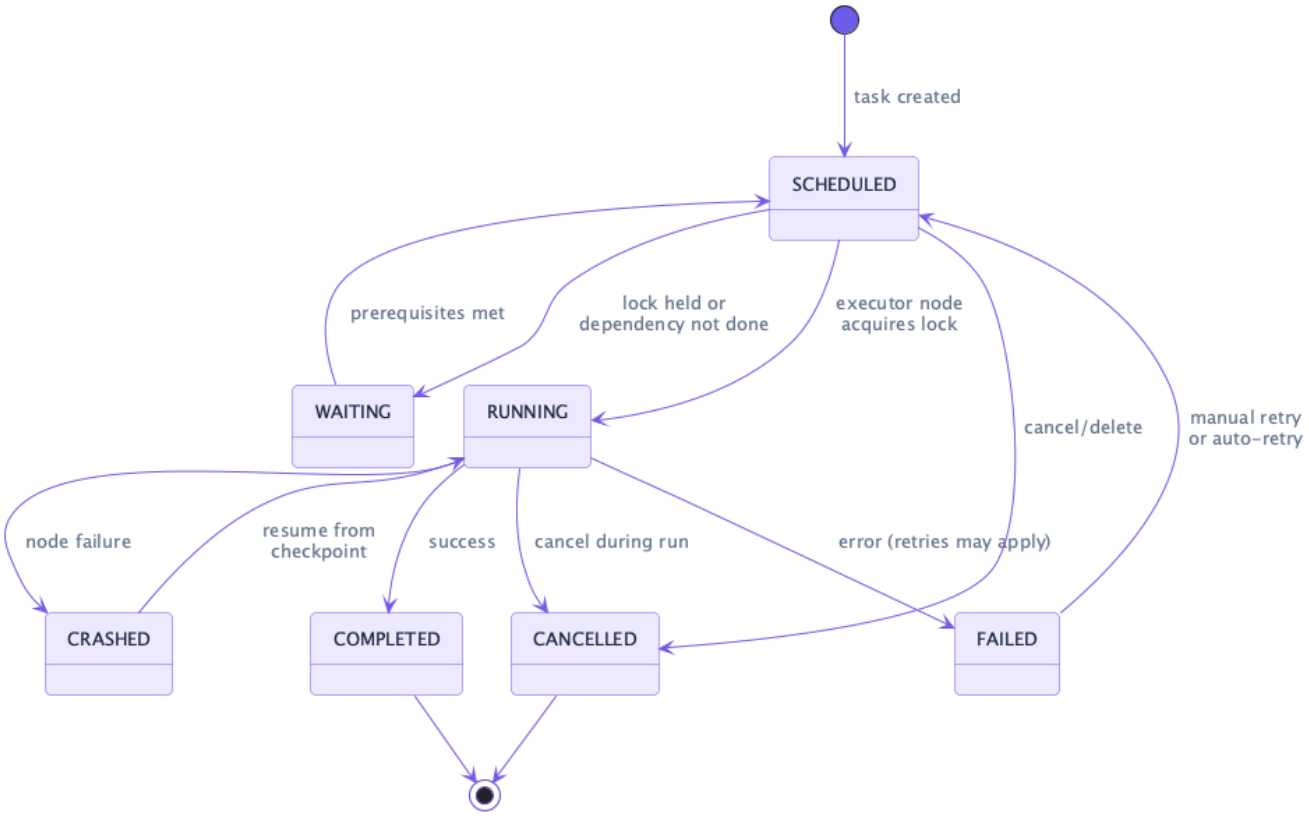
16.1.2. Architecture overview



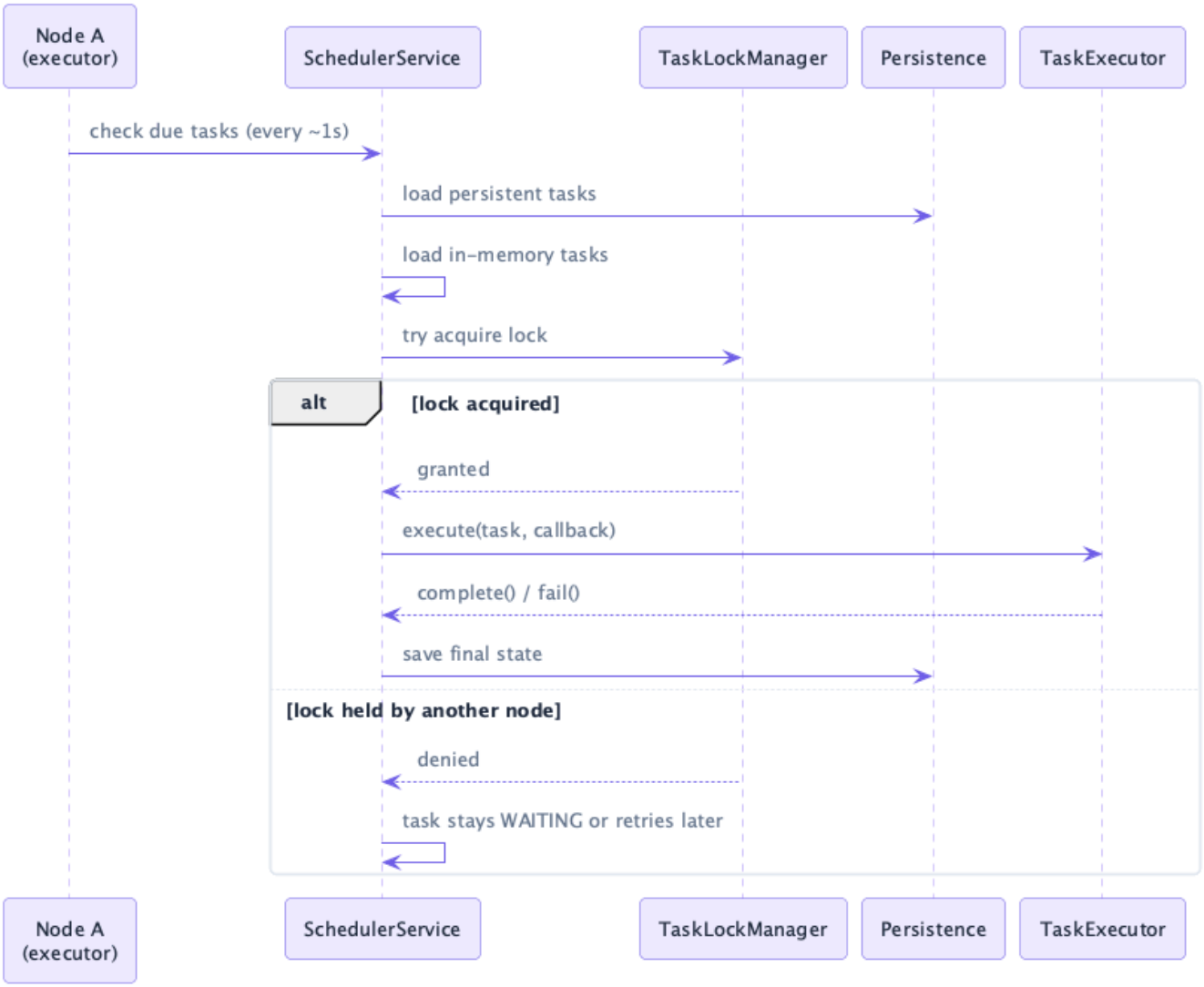
16.1.3. Task lifecycle

A ScheduledTask moves through well-defined states. Operators usually see SCHEDULED, RUNNING,

COMPLETED, FAILED, CANCELLED, or CRASHED.



16.1.4. Execution flow in a cluster

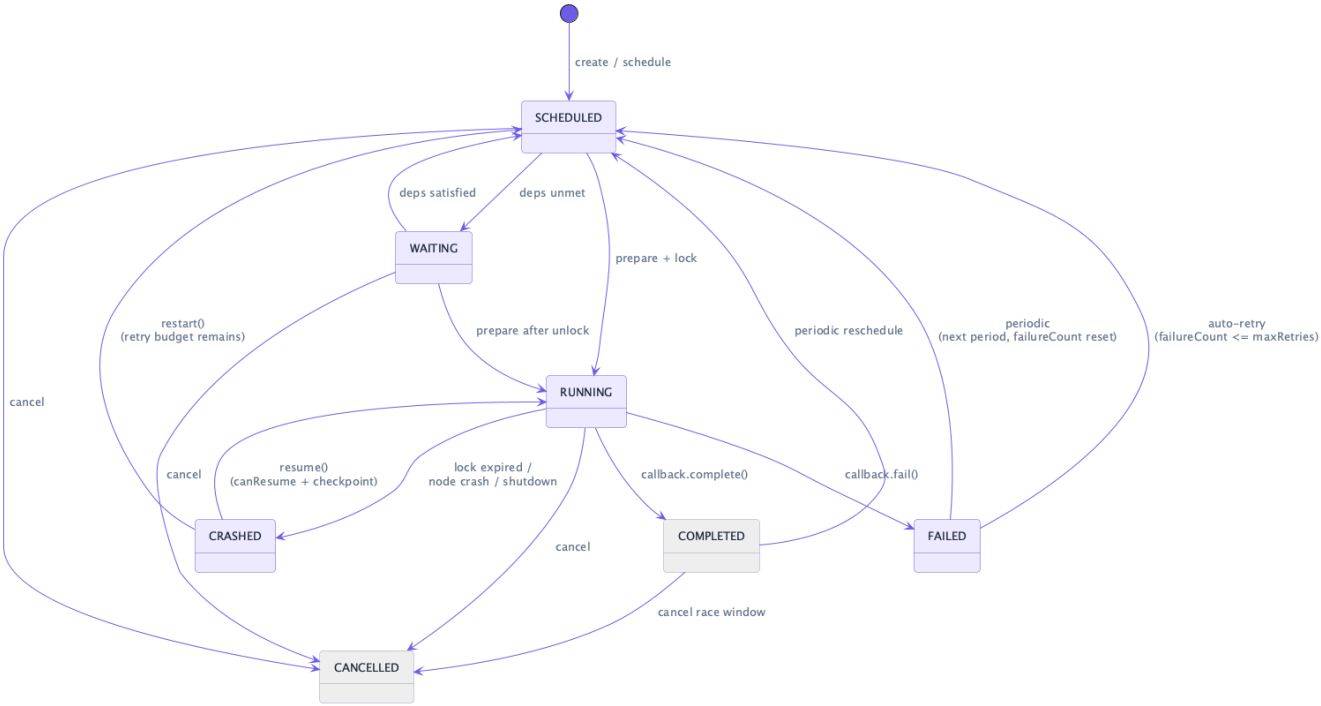


16.1.5. Task states in detail

Each persistent task document carries a status field. Transitions are validated by TaskStateManager (and a stricter check in TaskValidationManager for some API paths).

Status	Meaning	Typical next states
SCHEDULED	Eligible to run when nextScheduledExecution is due (and dependencies are satisfied).	RUNNING, WAITING, CANCELLED, CRASHED
WAITING	Blocked on unmet dependsOn / waitingOnTasks, or waiting for a lock/prerequisite.	SCHEDULED, RUNNING, CANCELLED
RUNNING	An executor node prepared the task, acquired the lock (if exclusive), and is executing (or resuming).	COMPLETED, FAILED, CANCELLED, CRASHED
COMPLETED	Successful terminal state for the current attempt. One-shot tasks are disabled; periodic tasks usually return to SCHEDULED.	SCHEDULED (periodic), CANCELLED (race window)
FAILED	Attempt failed. May return to SCHEDULED for retry while failureCount maxRetries.	SCHEDULED, terminal FAILED when retries exhausted (one-shot)

Status	Meaning	Typical next states
CRASHED	A node failure was detected (expired lock while RUNNING, or local shutdown). Visible to the task checker for recovery.	RUNNING (resume), SCHEDULED (restart)
CANCELLED	Operator or API cancelled the task. Terminal for execution; must not be overwritten by a late complete() / fail().	*(terminal)*



16.1.6. Exclusive vs parallel execution

Builder option	Cluster behaviour
disallowParallelExecution() (recommended for cluster singletons)	Distributed lock with optimistic concurrency (seq_no / primary_term) plus a fencing lockVersion. Only one node runs that task instance.
allowParallelExecution() (opt-in; builder default is false / exclusive)	No exclusive OCC lock — only a non-exclusive lockOwner marker. **Multiple executor nodes may run the same due task.** Use only when duplicate side effects are safe. runOnAllNodes() enables this automatically.
runOnAllNodes()	Every node (including non-executors that still run a checker for these tasks) should execute. Requires allowParallelExecution=true. Typical for per-node cache refresh.



newTask(...) defaults to exclusive locking (allowParallelExecution=false). Call allowParallelExecution() only when duplicates are intentional; runOnAllNodes() turns it on for you.

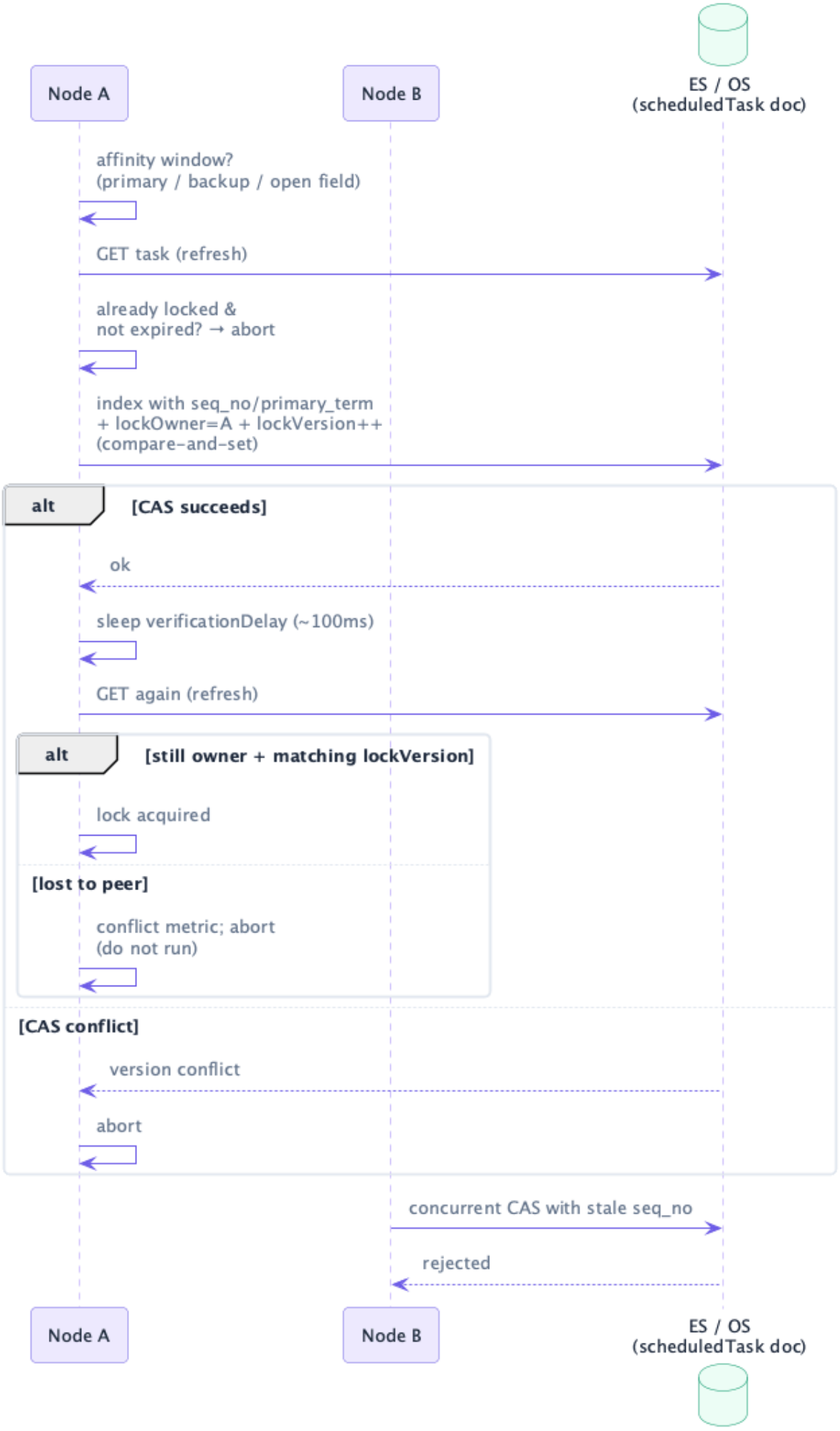
16.1.7. How distributed locks work with Elasticsearch / OpenSearch

Unomi does **not** use a separate lock service. The scheduledTask document **is** the lock. That matters because ES/OS is an asynchronous, eventually-consistent search engine — two nodes can briefly see different versions of the same document unless you design for it.

What is stored on the document

Field	Role
lockOwner	Node ID that currently holds the exclusive lock (or a non-exclusive marker when parallel is allowed).
lockDate	When the lock was taken. Used with scheduler.lockTimeout to detect dead holders.
systemMetadata.seq_no / primary_term	Elasticsearch/OpenSearch optimistic concurrency tokens. Compare-and-set rejects stale writers.
systemMetadata.lockVersion	Monotonic fencing token incremented on each successful lock acquire. Survives brief visibility races.
executingNodeId	Node currently inside the execution wrapper (set after successful prepare). Used to reclaim a premature CRASHED mark if this node is still alive.

Acquire protocol (exclusive persistent tasks)

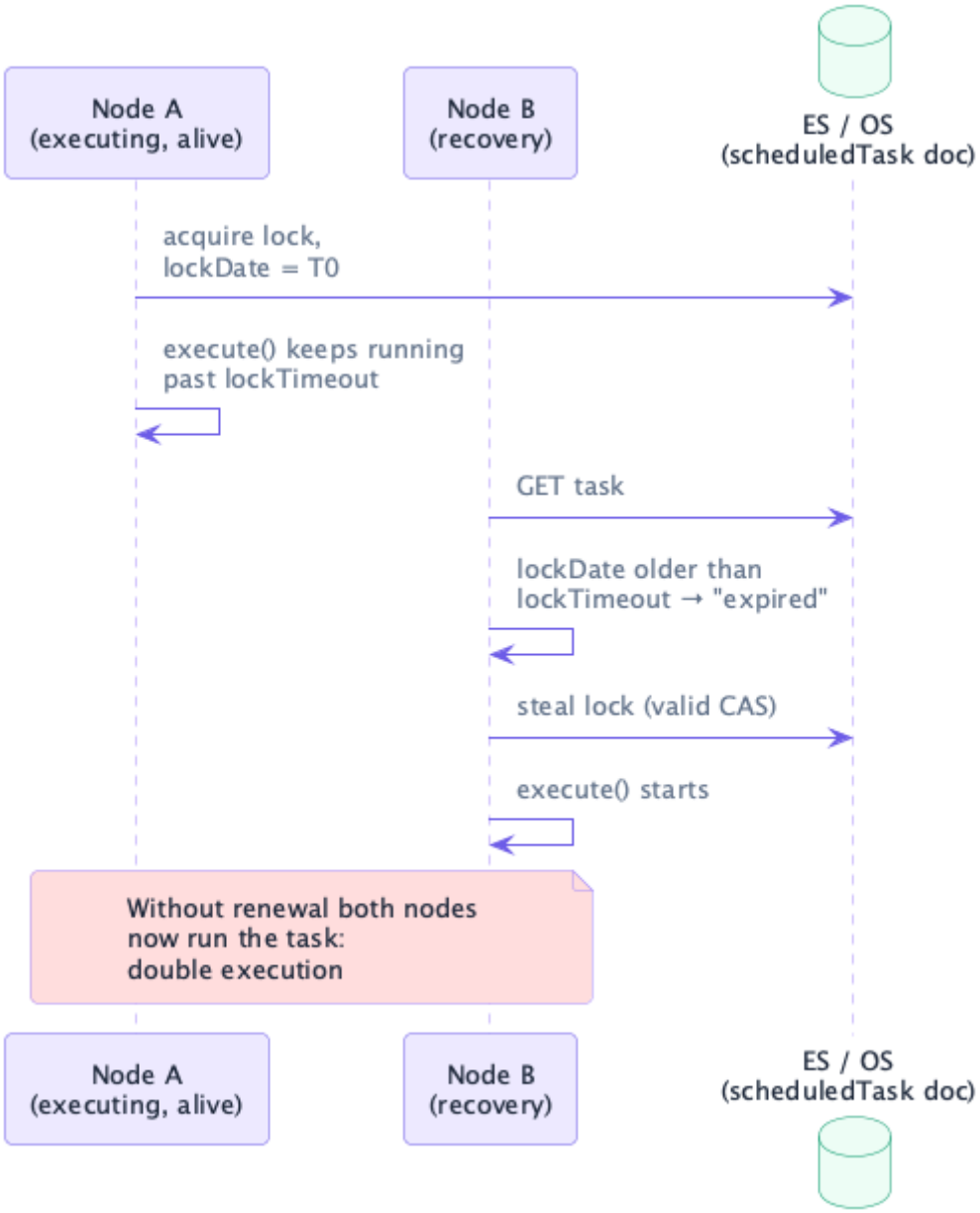

Node A
Node B
ES / OS
(scheduledTask doc)
affinity window?
(primary / backup / open field)
GET task (refresh)
already locked &
not expired? → abort
index with seq_no/primary_term
+ lockOwner=A + lockVersion++
(compare-and-set)
alt
[CAS succeeds]
ok
sleep verificationDelay (~100ms)
GET again (refresh)
alt
[still owner + matching lockVersion]
lock acquired
[lost to peer]
conflict metric; abort
(do not run)
[CAS conflict]
version conflict
abort
concurrent CAS with stale seq_no
rejected
Node A
Node B
ES / OS
(scheduledTask doc)

Key properties:

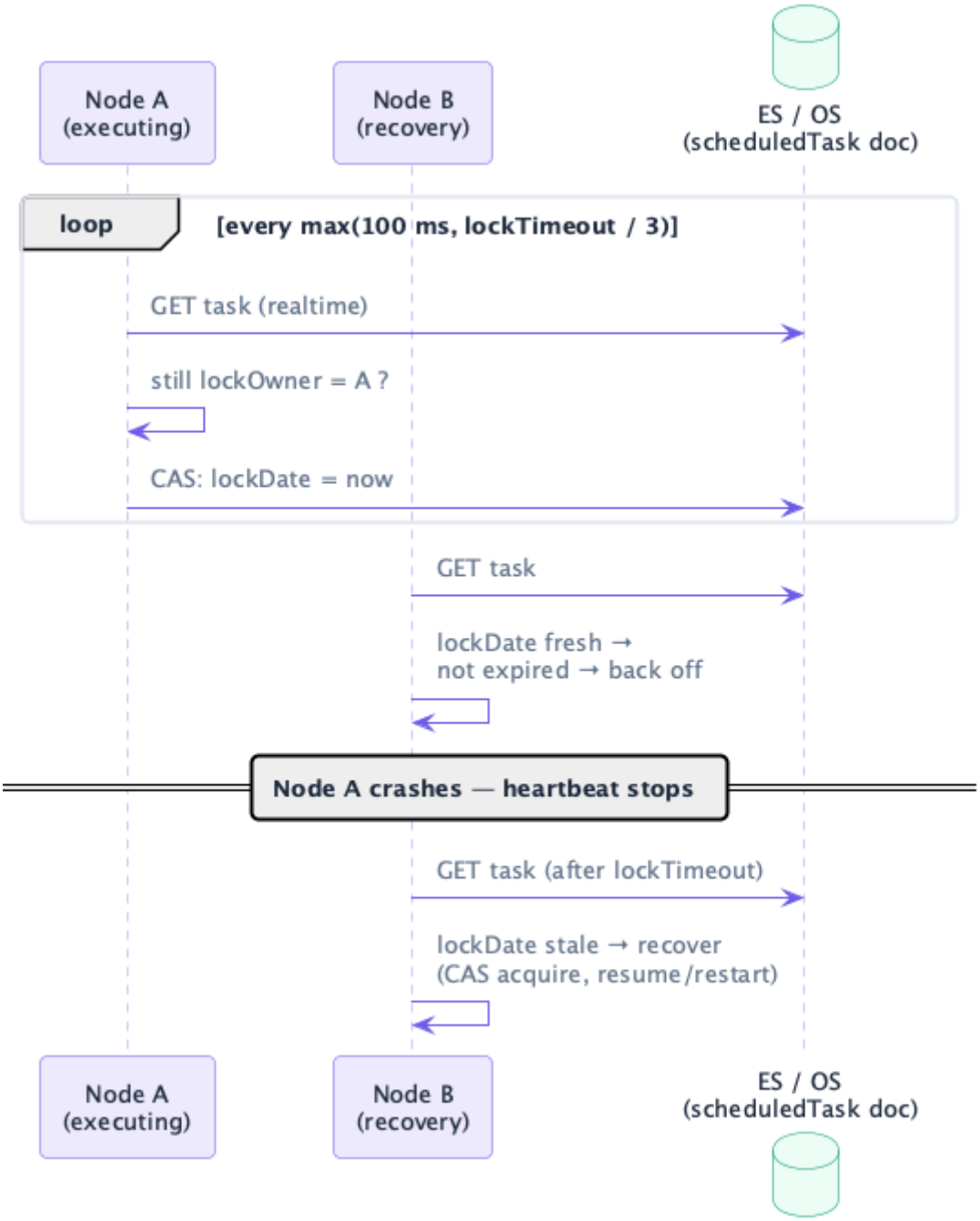
- **Compare-and-set** — two nodes cannot both win the lock write against the same seq_no.
- **WAIT_UNTIL / wait_for refresh** (configured for scheduledTask) — after a successful lock write, subsequent GETs on other nodes are much more likely to see it immediately. This is still not a distributed mutex primitive; the fencing token + verify step covers residual races.
- **Affinity windows** — reduce contention: the hashed primary node gets ~3s exclusivity, then staggered 500ms backup windows, then an **open field** so a dead primary that remains in getActiveNodes() cannot strand the task forever.
- **CRASHED bypasses affinity** — recovery must be able to resume immediately on a survivor.

Lock lease renewal (heartbeating)

A fixed lock timeout alone cannot distinguish **"the owner died"** from **"the execution is simply taking longer than scheduler.lockTimeout`". Without renewal, a long-running execution eventually looks expired to its peers, and crash recovery can then steal the lock — through a perfectly valid compare-and-set — from a node that is still working, producing a double execution. Guarding on `executingNodeId** instead would create the opposite failure: only the owner ever clears that field, so a task whose owner really is dead would be declined by every node forever.



While a node executes a persistent exclusive task it therefore **renews its lease**: a periodic heartbeat (every max(100 ms, lockTimeout / 3)) reloads the document with a realtime GET, verifies it still owns lockOwner, and refreshes lockDate with a compare-and-set write. Lock expiry then only ever means the owner **stopped renewing** — crashed, shut down, or unreachable — never merely "ran long".

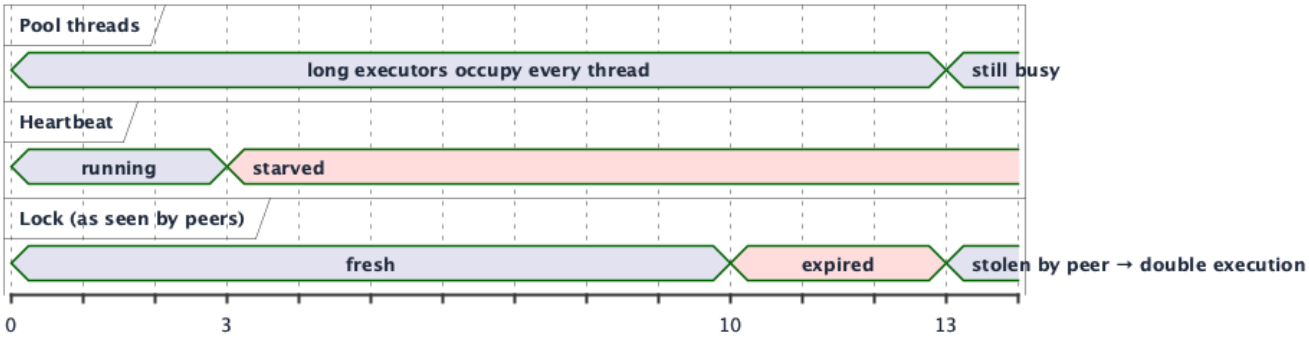


The heartbeat fails closed on any conflict: if a peer legitimately took over (or the document changed between the GET and the write), the CAS is rejected and the heartbeat backs off — it never resurrects a lost lock. Renewal deliberately uses the shutdown-sensitive load/save paths, so heartbeats go inert the moment shutdown begins and held locks age out normally for peers to recover. And just before a terminal COMPLETED / FAILED write, the heartbeat is stopped (waiting out any in-flight tick) so a late renewal cannot invalidate the terminal compare-and-set.

Renewal timing limits with the default 1 s refresh interval

Renewal is safe with the ES/OS default index refresh batching (refresh_interval = 1 s) because none of its building blocks depend on search visibility: compare-and-set is enforced at write time on the primary shard, and ownership checks use realtime GETs that read through the translog. Two operational bounds do follow from that batching, though:

- scheduledTask writes use the wait_for refresh policy, so **each heartbeat write may block up to ~1 s** waiting for the next refresh cycle. The safety margin is roughly lockTimeout − (lockTimeout / 3 + 1 s + scheduling jitter): comfortable at the default scheduler.lockTimeout=10000, but do **not** tune scheduler.lockTimeout below ~5 s in production or the refresh wait consumes the whole margin.
- Heartbeats run on the **same scheduler thread pool** (scheduler.thread.poolSize) as task executions and the task checker. If every pool thread is occupied by long-running blocking executors, heartbeats — like the checker — can starve, the lock expires under a live owner, and the pre-renewal steal scenario returns.



To stay clear of both bounds:

- **Size scheduler.thread.poolSize above the number of concurrently executing long-running exclusive tasks**, so a thread is always free for heartbeats and the checker.
- Or break long work into checkpointed steps with a resume()-capable executor, so no single execution blocks a pool thread for long stretches.

Release protocol (safe unlock)

Unlocking reloads the store document and **re-validates** before clearing lockOwner:

- If this node still owns the lock clear and save.
- If the lock is expired (dead holder) a recoverer may clear it.
- If a **peer holds a non-expired lock do not clear** (prevents a late completer from wiping a stealer's lock after expiry).

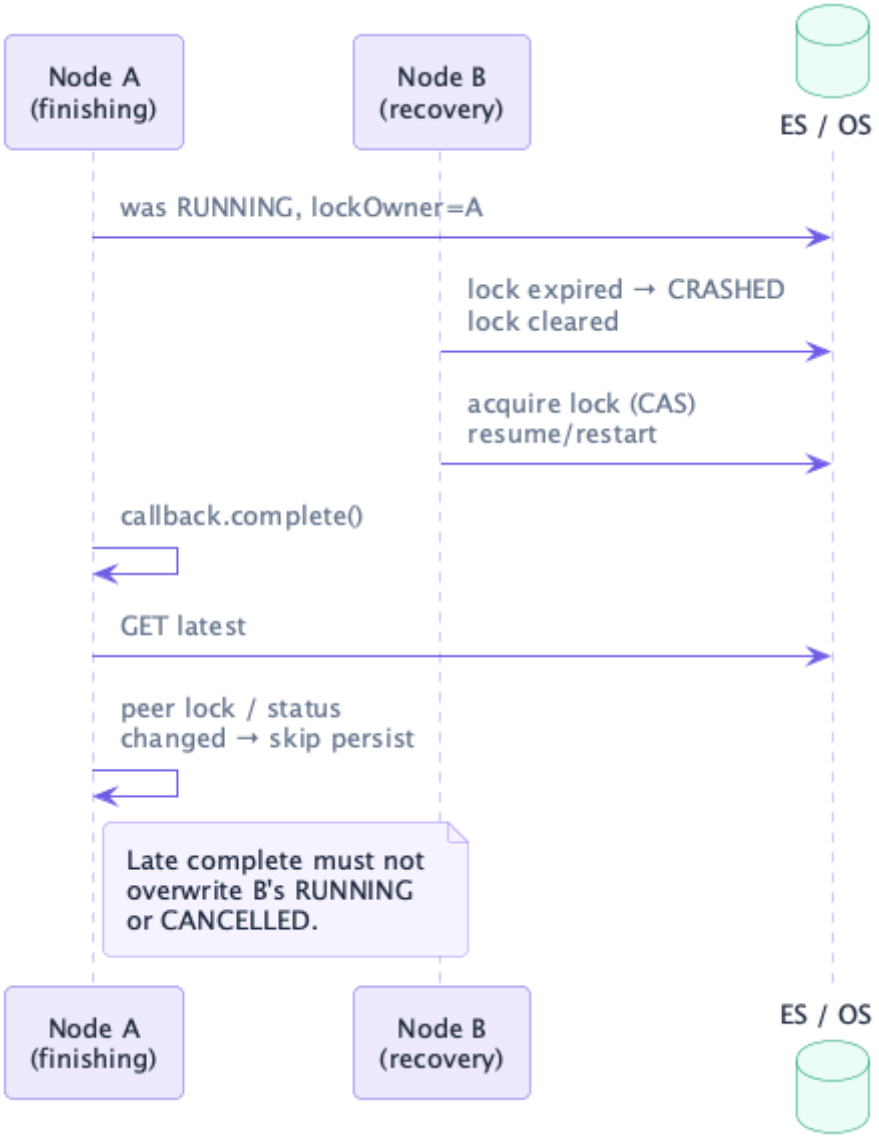
Terminal completion/error paths clear the lock on the **same CAS write** as the status change, so unlock and status stay atomic from the store's point of view.

Terminal complete / fail under concurrency

A node finishing work may be racing with cancel or with crash recovery. Before persisting COMPLETED / FAILED / retry SCHEDULED, the execution manager:

1. Reclaims local CRASHED RUNNING only if executingNodeId is still this node.
2. Reloads the store document.

3. Aborts if store is CANCELLED, or a peer owns executingNodeId / a non-expired lock, or store status is no longer RUNNING/CRASHED (for reclaim).
4. Persists with **compare-and-set** (saveTaskWithRefresh), including cleared lock fields.

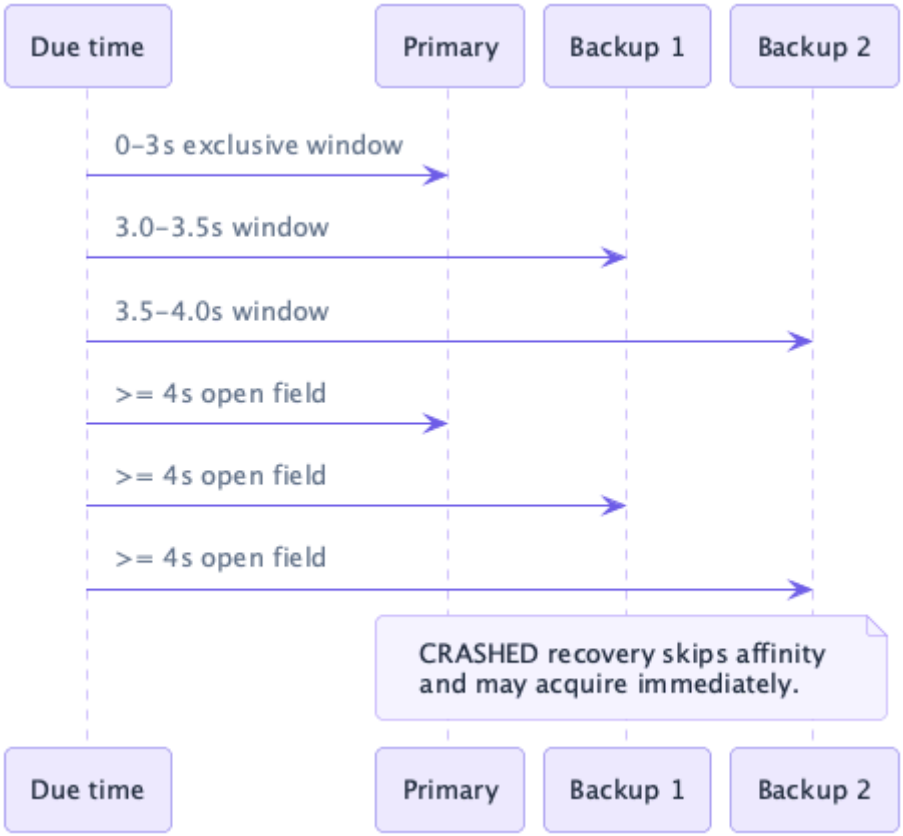


16.1.8. Node affinity timeline

For a due exclusive task with active nodes [N0, N1, N2] (sorted) and primary = hash(taskId) % 3. Active nodes are the nodes that sent a heartbeat in the last 5 minutes, leaving out those with scheduler.executorNode=false, which never run exclusive tasks:

Delay since nextScheduledExecution	Who may acquire	Purpose
0 .. 3000ms	Primary only	Reduce CAS storms; preferred owner runs first
3000 .. 3500ms	1st backup (ring order after primary)	Failover if primary is slow/dead
3500 .. 4000ms	2nd backup	Next staggered window (500ms each)

Delay since nextScheduledExecution	Who may acquire	Purpose
>= 4000ms (= 3000 + (n-1)*500)	Any active node (**open field**)	Avoid permanent stranding when a dead primary remains in getActiveNodes()



16.1.9. Crash recovery

Every checker tick (~1s) on a node runs recovery:

1. **Running tasks with expired locks** → mark CRASHED, clear dead lock, record history, then:
   - → canResume + checkpoint → keep CRASHED and call resume()
   - → else if failureCount < maxRetries → reset to SCHEDULED and execute()
   - → else leave CRASHED
2. **Locked SCHEDULED / WAITING with expired locks** → release lock; if deps allow, reschedule / dispatch.
3. **Never** release locks for RUNNING/CRASHED in the locked-task pass (that races the resume acquire).

executingNodeId lets a still-living node whose lock expired reclaim a false CRASHED when its callback fires (reclaimIfPrematurelyCrashed).

16.1.10. In-memory (non-persistent) tasks

nonPersistent() tasks live only in the creating node's memory map. They use a simple in-process lock (no ES CAS). They are ideal for local cache refresh and are **not** cluster-coordinated.

16.1.11. Operational checklist for clustering

- Each JVM needs a **stable unique** node id. Unomi generates and remembers one per node (see Node identity) and the scheduler uses it.
- Put at least one scheduler.executorNode=true in the cluster for persistent work.
- Use disallowParallelExecution() for singleton side-effecting jobs.
- Keep scheduler.lockTimeout at ~5s or above (default 10s): lease renewal keeps long executions safe automatically, but the heartbeat needs headroom over the wait_for refresh cycle (see Renewal timing limits).
- Size scheduler.thread.poolSize above the number of concurrently executing long-running exclusive tasks, so lock heartbeats and the task checker never starve.
- Remember ES/OS visibility: Unomi uses wait-for refresh + OCC + fencing; do not assume search-after-index without those safeguards.
- On cancel: expect CANCELLED to win over a late complete() from an in-flight thread.

16.2. Core concepts

Concept	Description
taskType	Logical name of the work (for example cache-refresh-segment, merge-profiles-reassign-data). Must match a registered TaskExecutor.
TaskExecutor	Java component that runs the work. Plugins register executors at activation time.
persistent task	Stored in the persistence layer (scheduledTask items). Visible on all cluster nodes. Survives restarts.
In-memory task	Exists only on the node that created it (nonPersistent()). Typical for local cache refresh.
Executor node	A node configured with scheduler.executorNode=true that actually runs tasks. Non-executor nodes can still create and manage tasks.
runOnAllNodes	When true, the task is eligible on every node (useful for per-node cache maintenance).
Lock	For exclusive persistent tasks: document-level OCC lock (lockOwner + seq_no /primary_term + lockVersion) with verify delay and affinity windows. See the distributed-locks section above. Does **not** apply when allowParallelExecution is true.
Checkpoint / resume	Long tasks can save progress and be resumed after a CRASHED state via REST or shell.

16.3. Task model

Tasks are ScheduledTask items (itemType = scheduledTask). Important fields:

Field	Meaning
taskType	Executor lookup key
parameters	JSON-serializable map passed to the executor
status	SCHEDULED, WAITING, RUNNING, COMPLETED, FAILED, CANCELLED, CRASHED
persistent	true = cluster-visible storage; false = node-local memory
oneShot	Run once then stop
period / timeUnit	Recurrence interval (0 means one-shot)
fixedRate	true = interval from start; false = fixed delay from end
allowParallelExecution	Default false (exclusive OCC lock). true = no exclusivity (multi-node duplicates possible)
maxRetries / retryDelay	Automatic retry policy after failure
dependsOn	Task IDs that must complete first
checkpointData / currentStep	Progress for resumable work
executingNodeId / lockOwner	Cluster coordination metadata

Example task JSON (as returned by REST)


```
{
  "itemId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "itemType": "scheduledTask",
  "taskType": "merge-profiles-reassign-data",
  "status": "COMPLETED",
  "persistent": true,
  "oneShot": true,
  "allowParallelExecution": false,
  "fixedRate": false,
  "period": 0,
  "timeUnit": "MILLISECONDS",
  "initialDelay": 1000,
  "failureCount": 0,
  "successCount": 1,
  "maxRetries": 3,
  "retryDelay": 60000,
  "parameters": {
    "tenantId": "mytenant",
    "masterProfileId": "profile-123",
    "mergedProfileIds": ["profile-456", "profile-789"],
    "anonymousBrowsing": false
  },
  "lastExecutedBy": "unomi-node-1",
  "lastExecutionDate": "2026-07-12T10:15:30.000Z"
}
```


16.4. Configuration

Scheduler settings live in etc/org.apache.unomi.services.cfg (OSGi PID org.apache.unomi.services).

Property	Default	Description
scheduler.thread.poolSize	5	Thread pool size for task execution
scheduler.nodeId	the node id	Unique ID for this node in scheduler locks and task metadata. Unomi provides the id of the node (see Node identity); a value set here is ignored. To give the scheduler an id of its own, set the org.apache.unomi.scheduler.nodeId system property, to a value unique per node.
scheduler.executorNode	true	Whether this node runs tasks. Set false on dedicated REST-only nodes if you split roles.
scheduler.lockTimeout	10000	Lock timeout in milliseconds
scheduler.purgeTaskEnabled	true	Enables the built-in periodic purge of old completed tasks
task.purge.completedTaskTtlDays	30	Age in days after which completed tasks may be purged

Example: dedicated executor node in a 3-node cluster

```
# Node 1 and 2: REST + task execution
scheduler.executorNode=true

# Node 3: API only, no task execution
scheduler.executorNode=false
```

After changing configuration, restart the Unomi bundle or the Karaf container.

16.5. REST API

Base path: {unomi}/cxs/tasks

Requires role ROLE_UNOMI_ADMIN or ROLE_UNOMI_TENANT_ADMIN (same as other administrative REST endpoints).



The REST API is for **monitoring and operations** (list, inspect, cancel, retry, resume). **Creating** tasks is done from Java code (SchedulerService) inside services or plugins, not via a public REST POST /tasks endpoint.

16.5.1. List tasks

List all tasks (paginated)

```
curl -s -u "karaf:$UNOMI_ROOT_PASSWORD" \
  "http://localhost:8181/cxs/tasks?offset=0&limit=20"
```

Filter by status

```
curl -s -u "karaf:$UNOMI_ROOT_PASSWORD" \
  "http://localhost:8181/cxs/tasks?status=FAILED&limit=50"
```

Valid status values: SCHEDULED, WAITING, RUNNING, COMPLETED, FAILED, CANCELLED, CRASHED.

Filter by task type

```
curl -s -u "karaf:$UNOMI_ROOT_PASSWORD" \
  "http://localhost:8181/cxs/tasks?type=cache-refresh-segment&limit=10"
```

Example response (truncated)


```
{
  "list": [
    {
      "itemId": "task-uuid-here",
      "taskType": "cache-refresh-segment",
      "status": "SCHEDULED",
      "persistent": false,
      "nextScheduledExecution": "2026-07-12T10:16:01.000Z"
    }
  ],
  "offset": 0,
  "pageSize": 20,
  "totalSize": 42,
  "totalSizeRelation": "EQUAL"
}
```


16.5.2. Get one task

```
curl -s -u "karaf:$UNOMI_ROOT_PASSWORD" \
  "http://localhost:8181/cxs/tasks/TASK_ID"
```

16.5.3. Cancel a task

Cancellation stops future runs and marks the task CANCELLED. The task record remains in storage.

```
curl -s -u "karaf:$UNOMI_ROOT_PASSWORD" -X DELETE \
  "http://localhost:8181/cxs/tasks/TASK_ID"
```

Returns HTTP 204 No Content on success.

16.5.4. Retry a failed task

```
# Retry keeping failure count
curl -s -u "karaf:$UNOMI_ROOT_PASSWORD" -X POST \
  "http://localhost:8181/cxs/tasks/TASK_ID/retry"

# Retry and reset failure count to zero
curl -s -u "karaf:$UNOMI_ROOT_PASSWORD" -X POST \
  "http://localhost:8181/cxs/tasks/TASK_ID/retry?resetFailureCount=true"
```

16.5.5. Resume a crashed task

Use when status is CRASHED and the executor supports checkpoint resume.

```
curl -s -u "karaf:$UNOMI_ROOT_PASSWORD" -X POST \
  "http://localhost:8181/cxs/tasks/TASK_ID/resume"
```

16.6. Karaf shell commands

Connect to the shell (ssh -p 8102 karaf@localhost) and use the unomi:task-* commands.

Command	Description
unomi:task-list	List tasks. Options: -s/--status, -t/--type, --limit
unomi:task-show TASK_ID	Detailed view including parameters and checkpoint data
unomi:task-cancel TASK_ID	Cancel a task
unomi:task-retry TASK_ID	Retry a failed task (-r resets failure count)
unomi:task-purge	Delete old COMPLETED tasks (-d days, -f force)
unomi:task-executor	Show whether this node is an executor node and its nodeId

Examples

```
# List running tasks
unomi:task-list -s RUNNING

# Show tasks created by profile merge follow-up
unomi:task-list -t merge-profiles-reassign-data

# Inspect a single task
unomi:task-show a1b2c3d4-e5f6-7890-abcd-ef1234567890

# Retry after fixing upstream data
unomi:task-retry a1b2c3d4-e5f6-7890-abcd-ef1234567890 --reset

# Purge completed tasks older than 14 days
unomi:task-purge --days 14 --force

# Check executor configuration on this node
unomi:task-executor
```

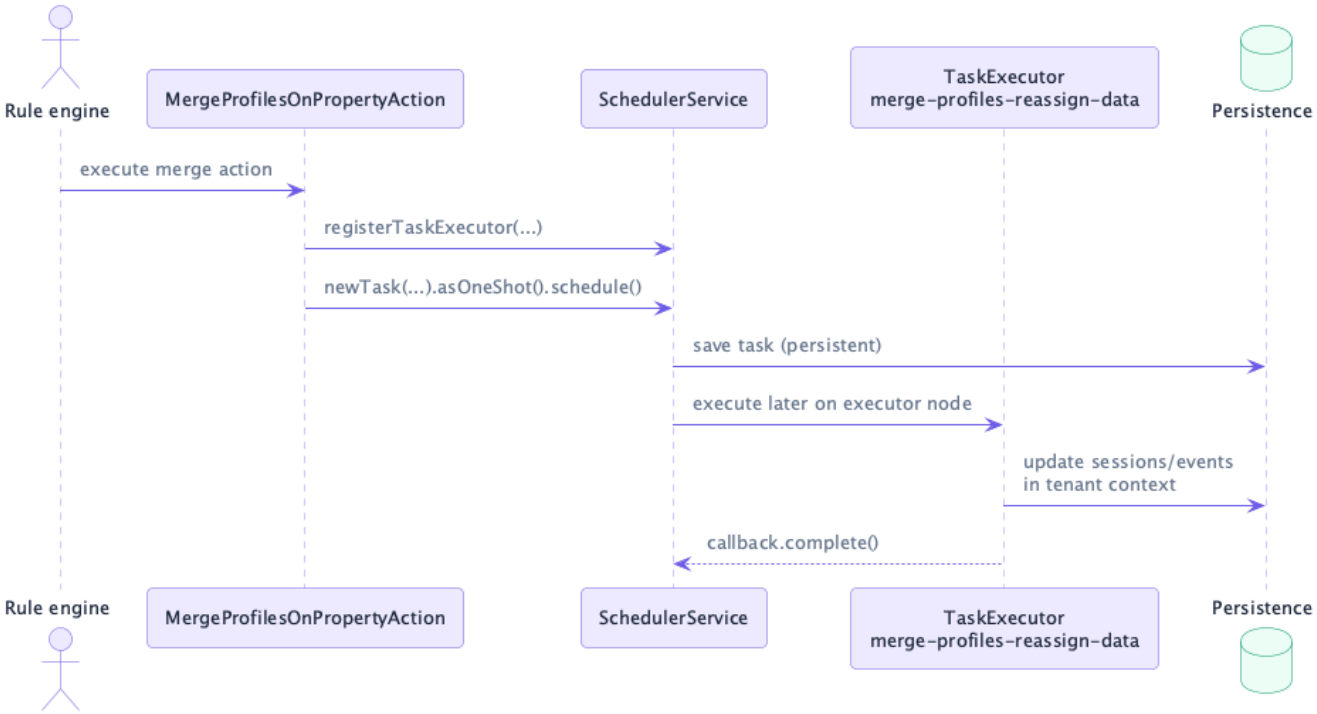
16.7. Using the scheduler in plugins

Plugins schedule work by:

1. Injecting SchedulerService (OSGi reference).
2. Registering a TaskExecutor for a unique taskType.
3. Creating tasks with newTask(...), createRecurringTask(...), or createTask(...).

16.7.1. Pattern A: one-shot async work (real example)

The mergeProfilesOnProperty action in baseplugin reassigns events and sessions in the background after a profile merge:



Simplified plugin code

```
@Reference
private SchedulerService schedulerService;

@Reference
private ExecutionContextManager executionContextManager;

private static final String TASK_TYPE = "merge-profiles-reassign-data";

public void activate() {
   schedulerService.registerTaskExecutor(new TaskExecutor() {
      @Override
      public String getTaskType() {
         return TASK_TYPE;
      }

      @Override
      public void execute(ScheduledTask task, TaskStatusCallback callback) {
         try {
            String tenantId = (String) task.getParameters().get("tenantId");
            executionContextManager.executeAsTenant(tenantId, () -> {
               // ... reassign sessions/events ...
               return null;
            });
            callback.complete();
         } catch (Exception e) {
            callback.fail(e.getMessage());
         }
      }
   });
}

public void scheduleReassign(String tenantId, String masterId, List<String> mergedIds) {
   schedulerService.newTask(TASK_TYPE)
      .withParameters(Map.of(
         "tenantId", tenantId,
         "masterProfileId", masterId,
         "mergedProfileIds", mergedIds,
         "anonymousBrowsing", false
      ))
      .withInitialDelay(1, TimeUnit.SECONDS)
      .asOneShot()
      .disallowParallelExecution()
      .schedule();
}
```



When task parameters include a tenantId, always run the work inside executionContextManager.executeAsTenant(...) (or set the security subject explicitly) so persistence queries respect tenant isolation.

16.7.2. Pattern B: simple recurring maintenance

For straightforward periodic code without a custom executor class:

```
schedulerService.createRecurringTask(
    "my-plugin-daily-cleanup",
    24, TimeUnit.HOURS,
    () -> cleanupStaleLocalCache(),
    false   // non-persistent: each node runs its own copy if executor
);
```

16.7.3. Pattern C: fluent builder (recommended)

The builder covers most scheduling options in one place.

One-shot task with retries

```
schedulerService.newTask("export-tenant-snapshot")
    .withParameters(Map.of("tenantId", "mytenant", "destination", "/tmp/export"))
    .withInitialDelay(5, TimeUnit.SECONDS)
    .asOneShot()
    .withMaxRetries(3)
    .withRetryDelay(2, TimeUnit.MINUTES)
    .disallowParallelExecution()
    .withExecutor(myExportExecutor)
    .schedule();
```

Periodic task with fixed delay

```
schedulerService.newTask("refresh-external-lookup")
    .withPeriod(15, TimeUnit.MINUTES)
    .withFixedDelay()
    .disallowParallelExecution()
    .nonPersistent()
    .withSimpleExecutor(() -> refreshLookupTable())
    .schedule();
```

Periodic task on every cluster node

```
schedulerService.newTask("local-temp-file-cleanup")
    .withPeriod(1, TimeUnit.HOURS)
    .withFixedDelay()
    .runOnAllNodes()
    .nonPersistent()
    .withSimpleExecutor(() -> deleteStaleTempFiles())
    .schedule();
```

Task chain with dependencies

```
ScheduledTask extract = schedulerService.newTask("etl-extract")
    .asOneShot()
    .withExecutor(extractExecutor)
    .schedule();

schedulerService.newTask("etl-load")
    .asOneShot()
    .withDependencies(extract.getItemId())
    .withExecutor(loadExecutor)
    .schedule();
```

16.7.4. Pattern D: resumable long-running executor

For work that may run longer than a node failover window, save checkpoints and implement canResume / resume:

```
public class BulkReindexExecutor implements TaskExecutor {

    @Override
    public String getTaskType() {
        return "bulk-reindex";
    }

    @Override
    public void execute(ScheduledTask task, TaskStatusCallback callback) {
        runFromOffset(task, callback, 0);
    }

    @Override
    public boolean canResume(ScheduledTask task) {
        return task.getCheckpointData() != null
            && task.getCheckpointData().containsKey("lastOffset");
    }

    @Override
    public void resume(ScheduledTask task, TaskStatusCallback callback) {
        int offset = ((Number) task.getCheckpointData().get("lastOffset")).intValue();
        runFromOffset(task, callback, offset);
    }

    private void runFromOffset(ScheduledTask task, TaskStatusCallback callback, int start) {
        try {
            for (int i = start; i < total; i += batchSize) {
                reindexBatch(i, batchSize);
                callback.updateStep("reindex", Map.of("lastOffset", i + batchSize));
                callback.checkpoint(Map.of("lastOffset", i + batchSize));
            }
            callback.complete();
        } catch (Exception e) {
            callback.fail(e.getMessage());
        }
    }
}
```

After a crash, operators can call POST /cxs/tasks/{id}/resume or unomi:task-retry depending on the failure mode.

16.7.5. OSGi Declarative Services wiring

Inject SchedulerService with @Reference in your plugin component. New plugins use Declarative Services — not Blueprint XML (see Writing plugins).

```
@Component(service = MyPluginService.class)
public class MyPluginService {

    private SchedulerService schedulerService;
    private TaskExecutor registeredExecutor;

    @Reference
    public void setSchedulerService(SchedulerService schedulerService) {
        this.schedulerService = schedulerService;
    }

    @Activate
    public void activate() {
        registeredExecutor = new MyTaskExecutor();
        schedulerService.registerTaskExecutor(registeredExecutor);
    }

    @Deactivate
    public void deactivate() {
        if (schedulerService != null && registeredExecutor != null) {
            schedulerService.unregisterTaskExecutor(registeredExecutor);
        }
    }
}
```

Register the executor in activate() and unregister in deactivate() when possible. Pattern A above uses the same @Reference injection style for SchedulerService.

16.8. Built-in tasks you will see in production

Unomi core and plugins already create scheduler tasks. These are normal and expected:

Task type (examples)	Purpose
cache-refresh-*	Reload segments, rules, property types, scopes, etc. from persistence into memory
task-purge	System task that purges old completed scheduler tasks
merge-profiles-reassign-data	Async session/event reassignment after profile merge (baseplugin)

Use unomi:task-list or GET /cxs/tasks?type=... to inspect them.

16.9. Metrics

SchedulerService exposes counters useful for monitoring:

```
Map<String, Long> metrics = schedulerService.getAllMetrics();
// keys include:
// tasks.completed, tasks.failed, tasks.crashed, tasks.running,
// tasks.lock.conflicts, tasks.recovery.attempts, ...
```

In custom code:

```
long failures = schedulerService.getMetric("tasks.failed");
```

16.10. Choosing persistence and execution settings

Need	persistent?	runOnAllNodes?	Notes
Cluster-wide job (ETL, merge follow-up)	yes	no	Survives restart; one executor runs it
Per-node cache refresh	no	no or yes	nonPersistent() + often runOnAllNodes() for local caches
Fire-and-forget after HTTP request	yes or no	no	One-shot; tune maxRetries
Must never overlap	any	any	disallowParallelExecution()

16.11. Troubleshooting

Symptom	What to check
Task stays SCHEDULED forever	Is this node an executor (scheduler.executorNode=true)? Is a TaskExecutor registered for the taskType?
Task stays WAITING	Another run may hold the lock, or a dependency task is not COMPLETED
Task FAILED after retries	Read lastError via task-show or REST; fix root cause; call retry
Task CRASHED	Node failure during run; use resume if checkpoints exist
Duplicate cache refresh tasks	Use disallowParallelExecution() and stable task names (see AbstractMultiTypeCachingService)
Same persistent task runs on two nodes	Was allowParallelExecution() called explicitly (default is false/exclusive)? Use disallowParallelExecution(). Check tasks.lock.conflicts metrics.
Task CRASHED then stuck	Is an executor registered? Is retry budget exhausted (failureCount vs maxRetries)? Does the executor implement canResume/resume when checkpoints exist?

Symptom	What to check
Lock timeout storms / false CRASHED	Usually a starved heartbeat: check that scheduler.thread.poolSize exceeds the number of concurrent long-running tasks. Otherwise increase scheduler.lockTimeout (keep ≥ ~5s), or checkpoint + resume for long work. Verify clocks are roughly NTP-aligned across nodes.
Cancelled task becomes COMPLETED	Should not happen after Unomi 4.0 terminal CAS guards; if seen, capture seq_no history and node logs around Skipping terminal transition.
Tenant data wrong in background job	Missing executeAsTenant / security subject setup in executor

16.12. Related documentation

- Tenant REST examples — creating tenants whose data background tasks will process
- Karaf shell — operational commands including unomi:task-*
- Writing plugins — DS plugin registration and META-INF/cxs/ layout
- Multitenancy — tenant isolation rules

Chapter 17. Health check

Apache Unomi exposes an HTTP health endpoint at /health/check with pluggable providers.

Typical providers:

- karaf — always required (Karaf framework)
- elasticsearch or opensearch — search backend (match your distribution)
- persistence — persistence binding
- cluster — cluster node registry
- unomi — Unomi bundles / services

Docker / env example (OpenSearch distribution):

```
UNOMI_HEALTHCHECK_ENABLED=true
UNOMI_HEALTHCHECK_PROVIDERS=cluster,opensearch,unomi,persistence
```

The endpoint is protected by the health role (user health, password from UNOMI_HEALTHCHECK_PASSWORD — no default is shipped). Full provider configuration, sample JSON, and extension points are documented in the Configuration chapter: Health check extension.

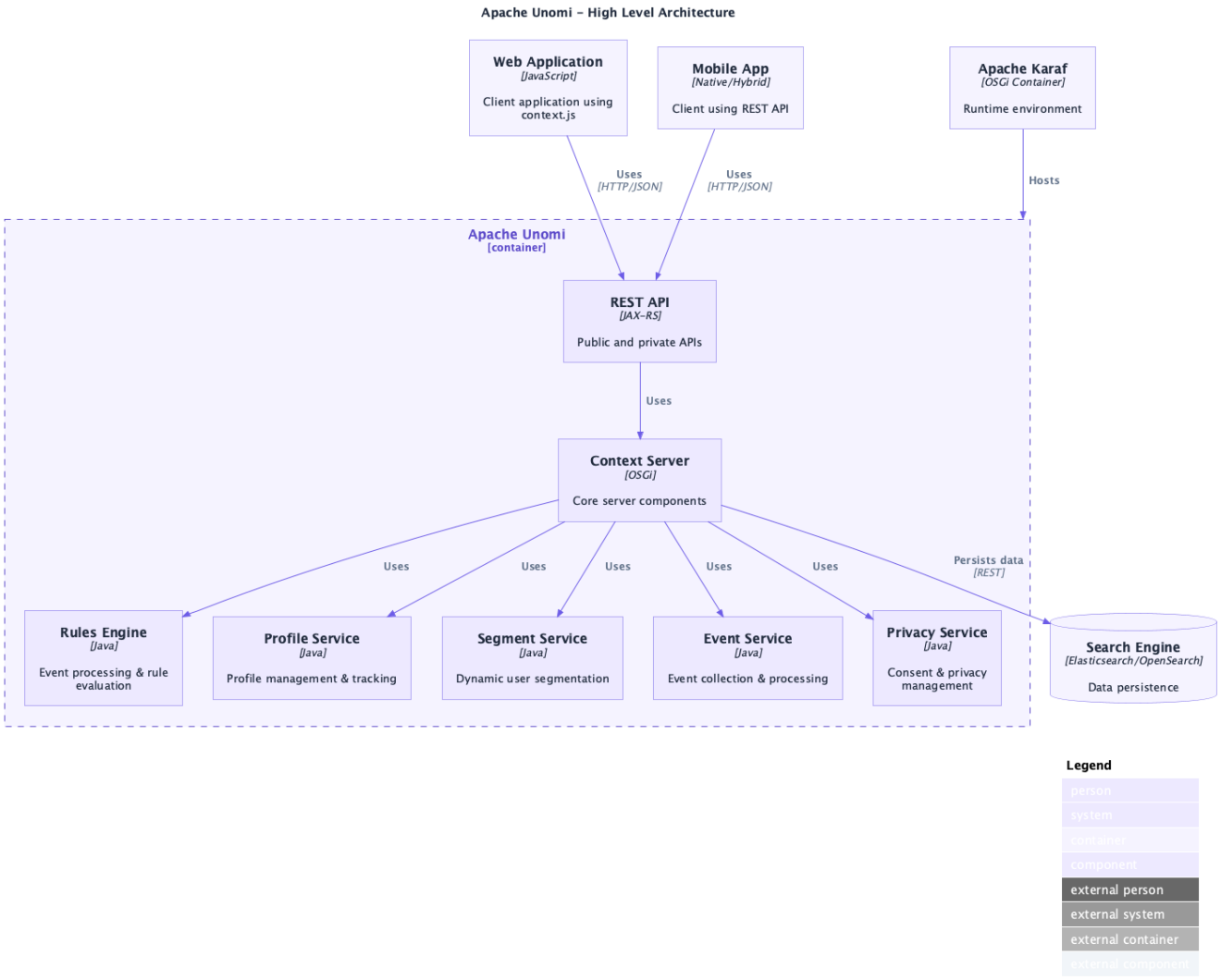
Chapter 18. Reference

18.1. Architecture Overview

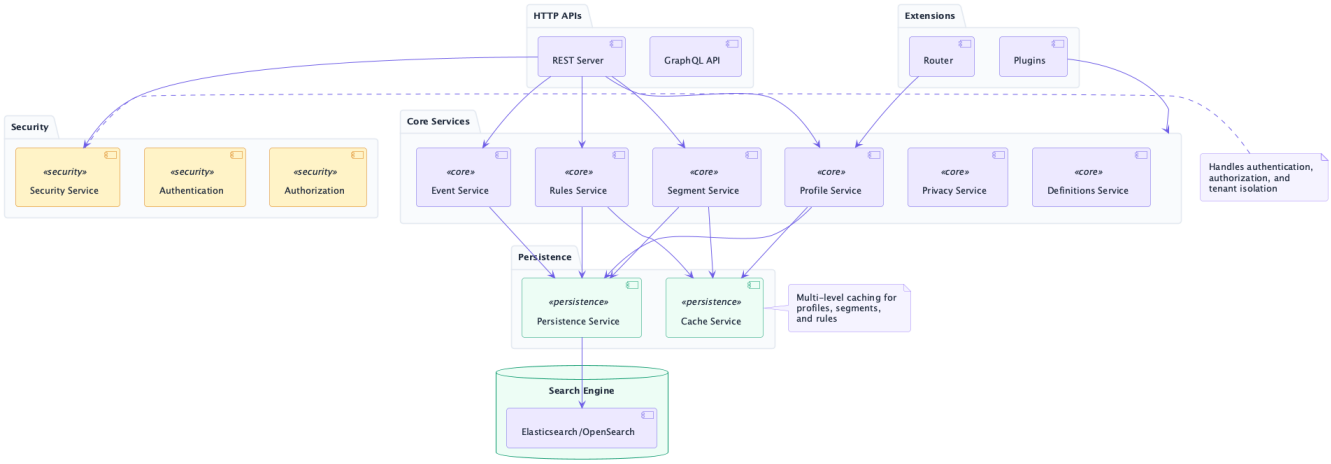
18.2. High-Level Architecture



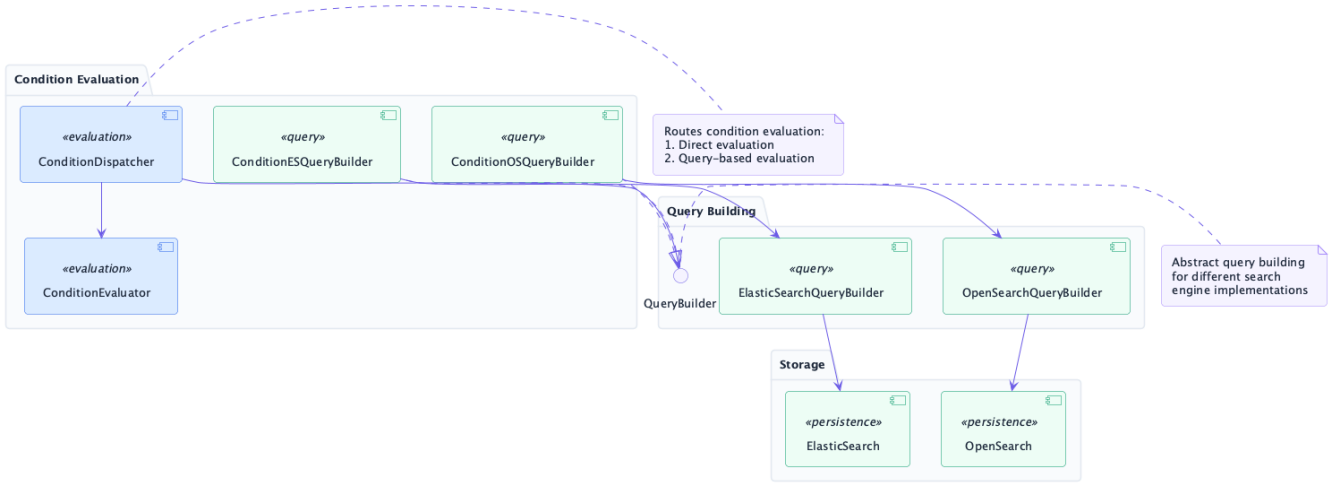
Unomi 4.0 adds multi-tenancy and task scheduling on top of the 3.0 platform (including persistence-based clustering). The diagram below focuses on core request processing.



18.3. Service Architecture



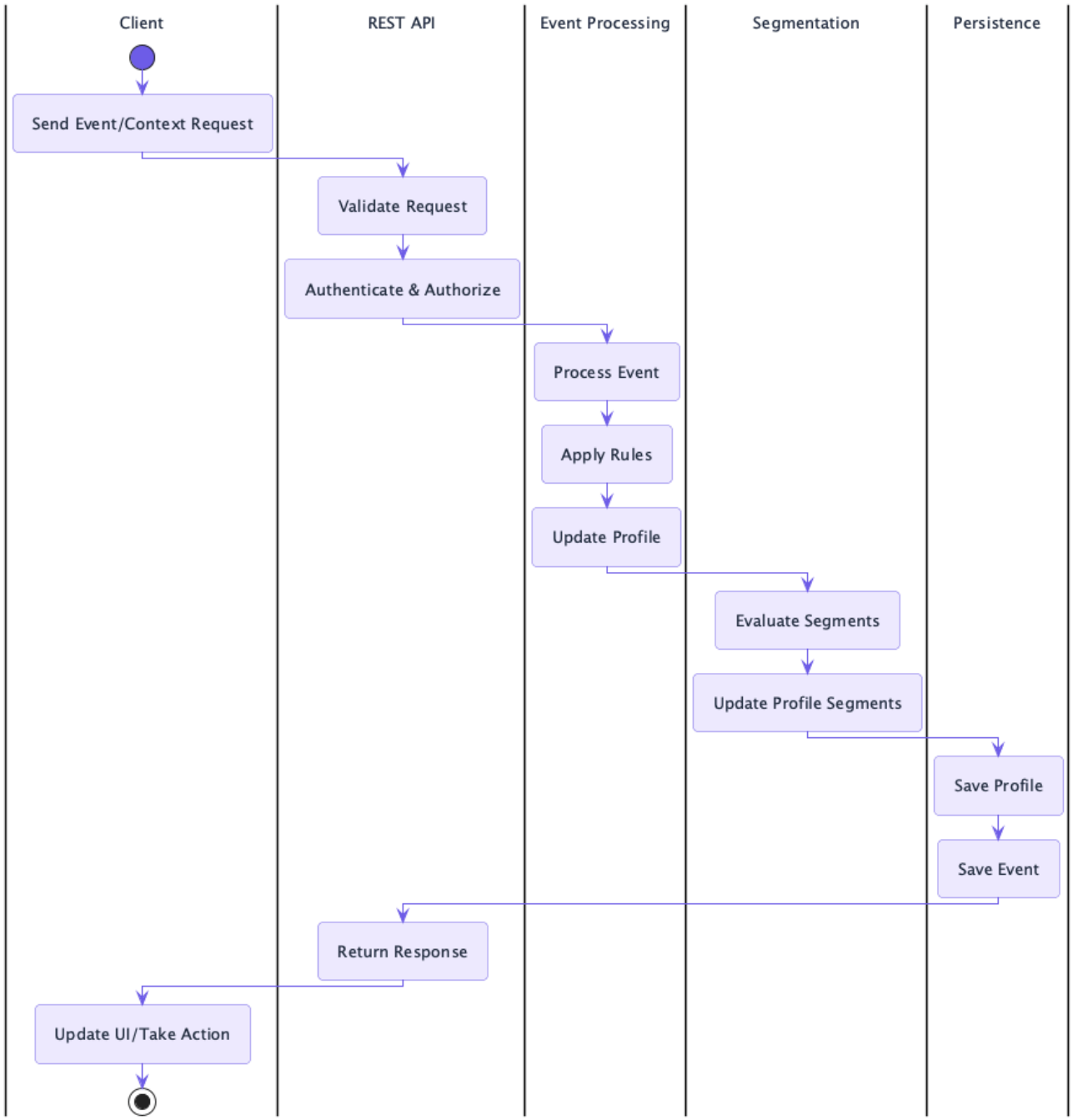
18.4. Condition Evaluation Architecture



18.4.1. Key Components

1. **ConditionDispatcher**
    - Routes conditions to appropriate evaluators
    - Determines evaluation strategy (direct vs query-based)
    - Handles caching and optimization
2. **ConditionEvaluator**
    - Performs direct condition evaluation
    - Handles boolean logic and property comparisons
    - Evaluates nested conditions
3. **QueryBuilders**
    - Convert conditions to search engine queries
    - Handle search engine specific syntax
    - Optimize query performance

18.5. Data Flow

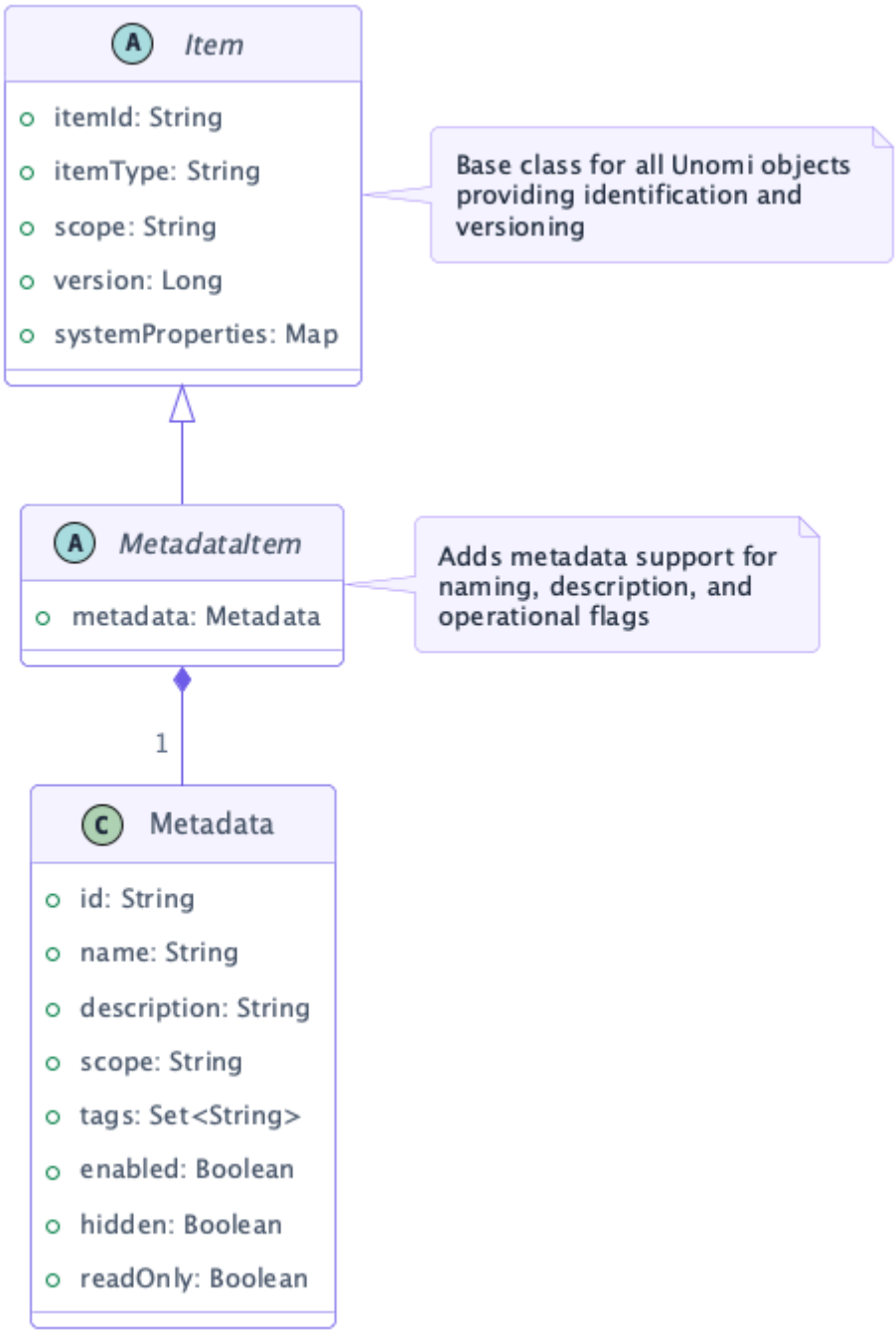


18.5.1. Data Structures

18.5.2. Core Data Structures Overview

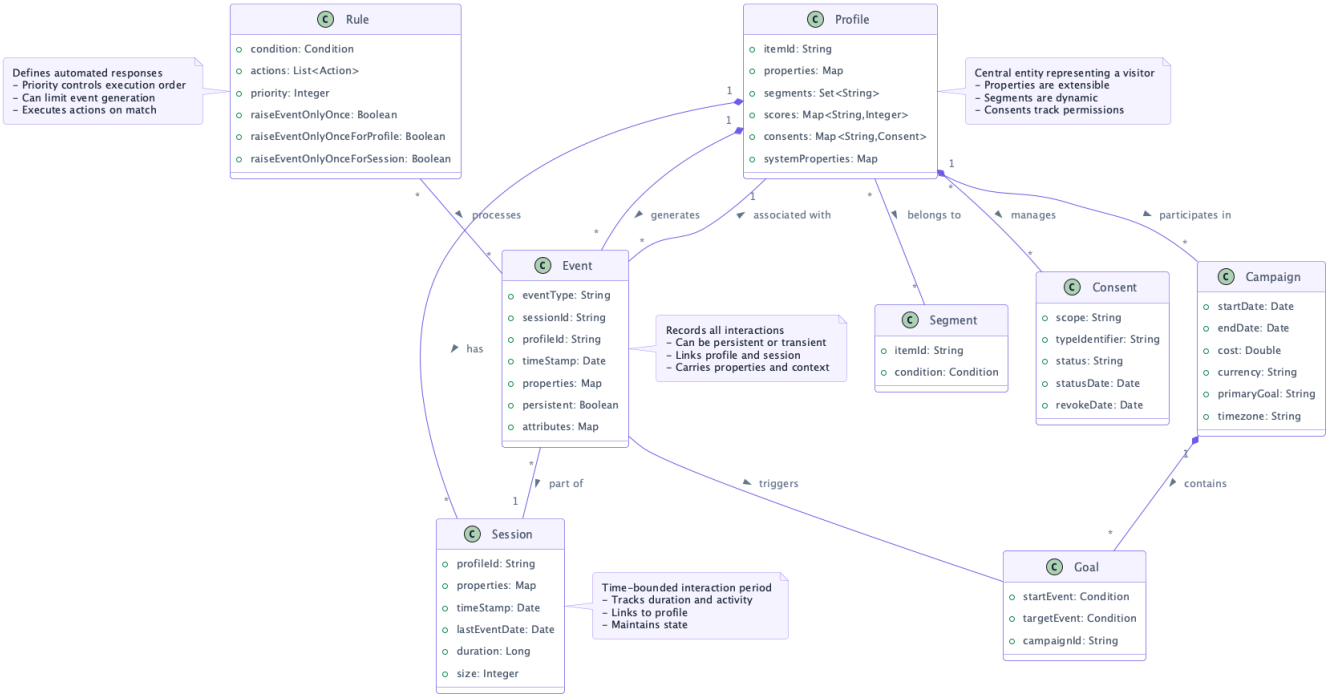
18.5.3. Base Types

All major entities in Apache Unomi inherit from two base classes that provide essential functionality:



18.5.4. Core Entities and Their Relationships

This diagram shows the key data structures in Apache Unomi and their relationships:



18.5.5. Key Relationships

- **Profile** is the central entity representing a visitor/customer
  - Properties are fully extensible via Map structure
  - Segments are dynamically evaluated and stored as Set
  - Scores track numeric metrics per profile
  - Consents manage user permissions and preferences
  - System properties store internal data
- **Event** represents any interaction or activity
  - Always associated with both profile and session
  - Can be persistent (stored) or transient
  - Carries properties and context attributes
  - Timestamp tracks occurrence time
  - Can trigger rules and goals
- **Session** represents a time-bounded interaction period
  - Belongs to a single profile
  - Tracks duration and last activity
  - Maintains size counter for events
  - Properties store session-specific state
- **Rule** defines automated behavior
  - Processes events using conditions

- Executes actions when conditions match
- Priority controls execution order
- Can limit event generation:
  - Once per profile
  - Once per session
  - Once globally

- **Segment** groups profiles dynamically
  - Uses conditions to determine membership
  - Profiles can belong to multiple segments
  - Automatically maintained by the system
- **Goal** tracks objective completion
  - Defined by start and target conditions
  - Can be part of a campaign
  - Triggered by specific events
- **Campaign** organizes marketing activities
  - Contains multiple goals
  - Tracks costs and timeframes
  - Associates profiles with marketing efforts
  - Supports timezone-aware scheduling
- **Consent** manages user permissions
  - Scoped to specific areas/features
  - Tracks status and dates
  - Supports revocation
  - Linked to profiles

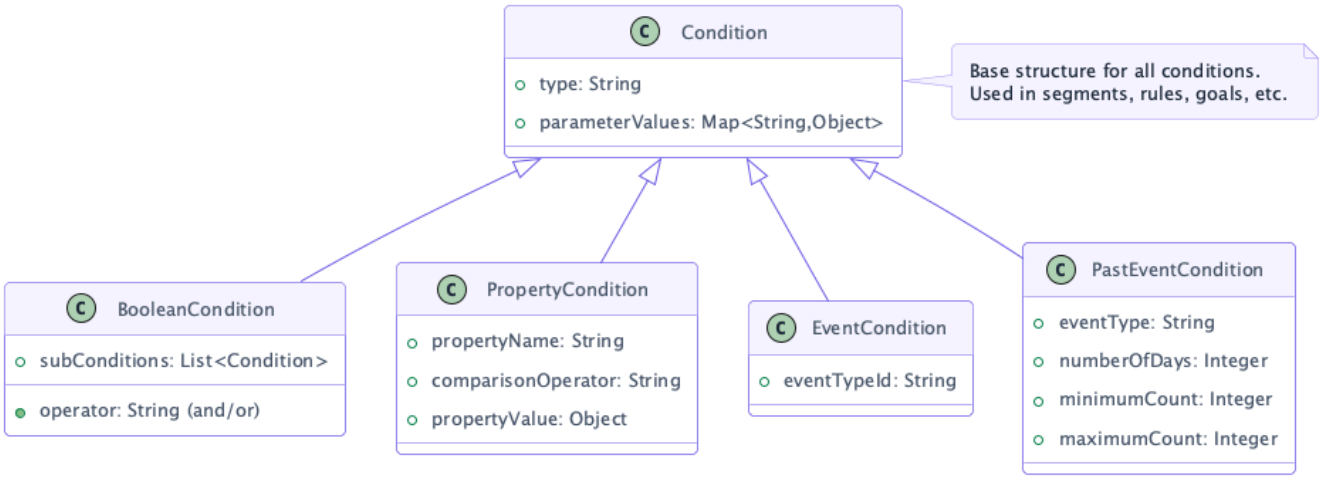
All these entities inherit from MetadataItem which provides: - Metadata (name, description, scope, tags) - Enabled/disabled status - Hidden/visible status - Read-only protection - Version tracking

18.5.6. Detailed Structure Definitions

The following sections provide detailed diagrams for each major component:

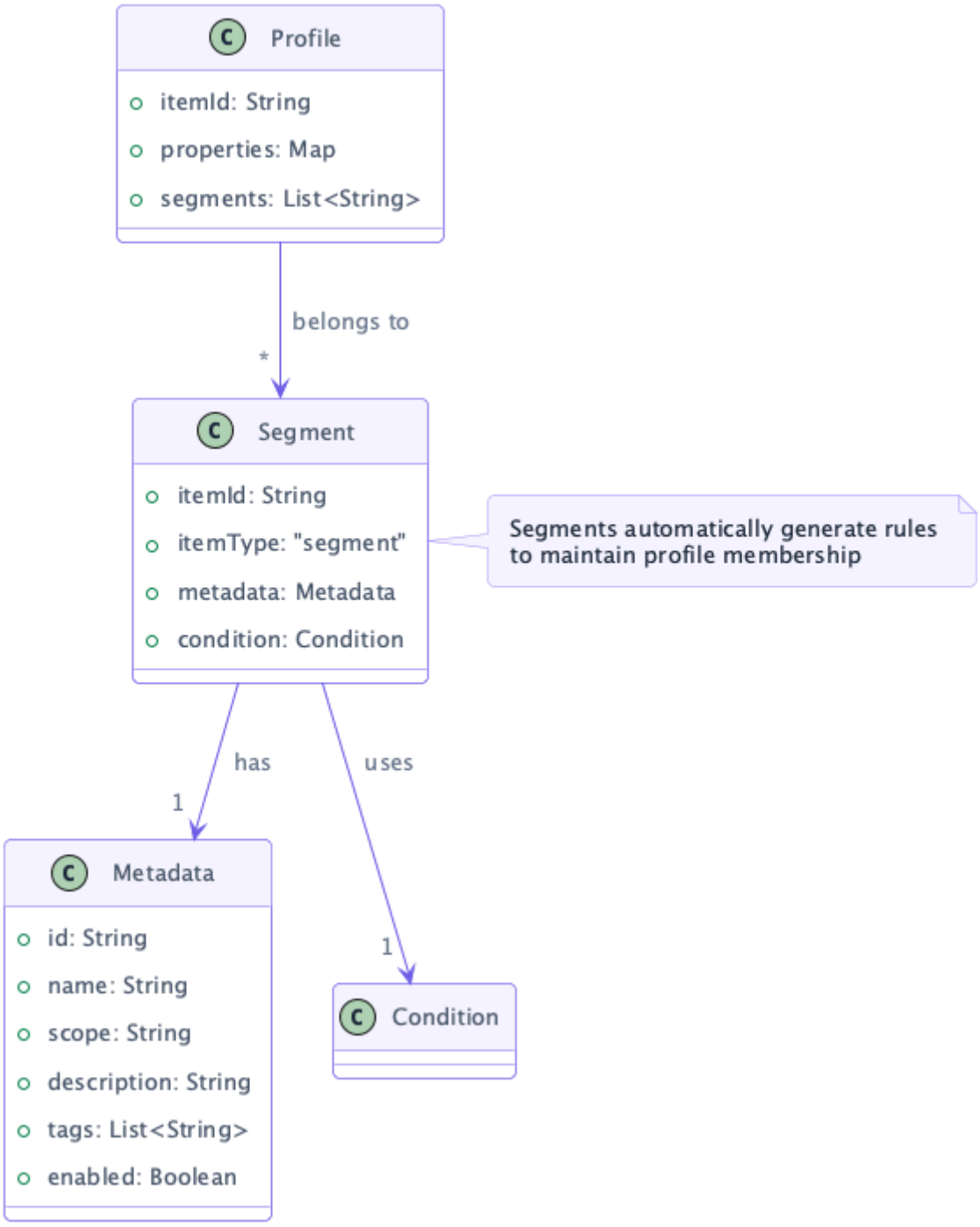
18.5.7. Condition Structure

Conditions are the fundamental building blocks used across many Unomi components. They evaluate to true/false and can be composed.



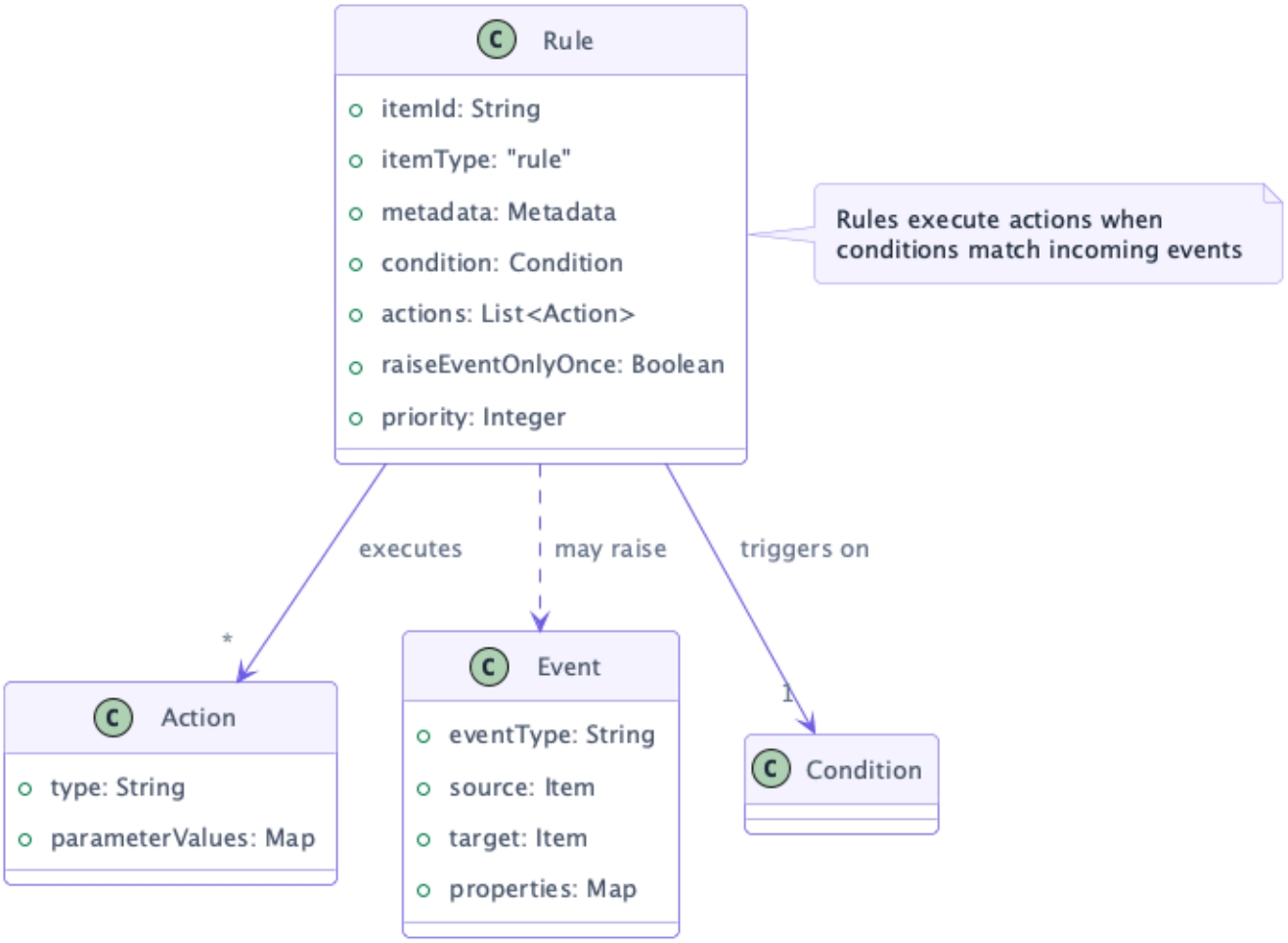
18.5.8. Segment Structure

Segments group profiles based on conditions. They are dynamic - profiles can enter/exit based on condition evaluation.



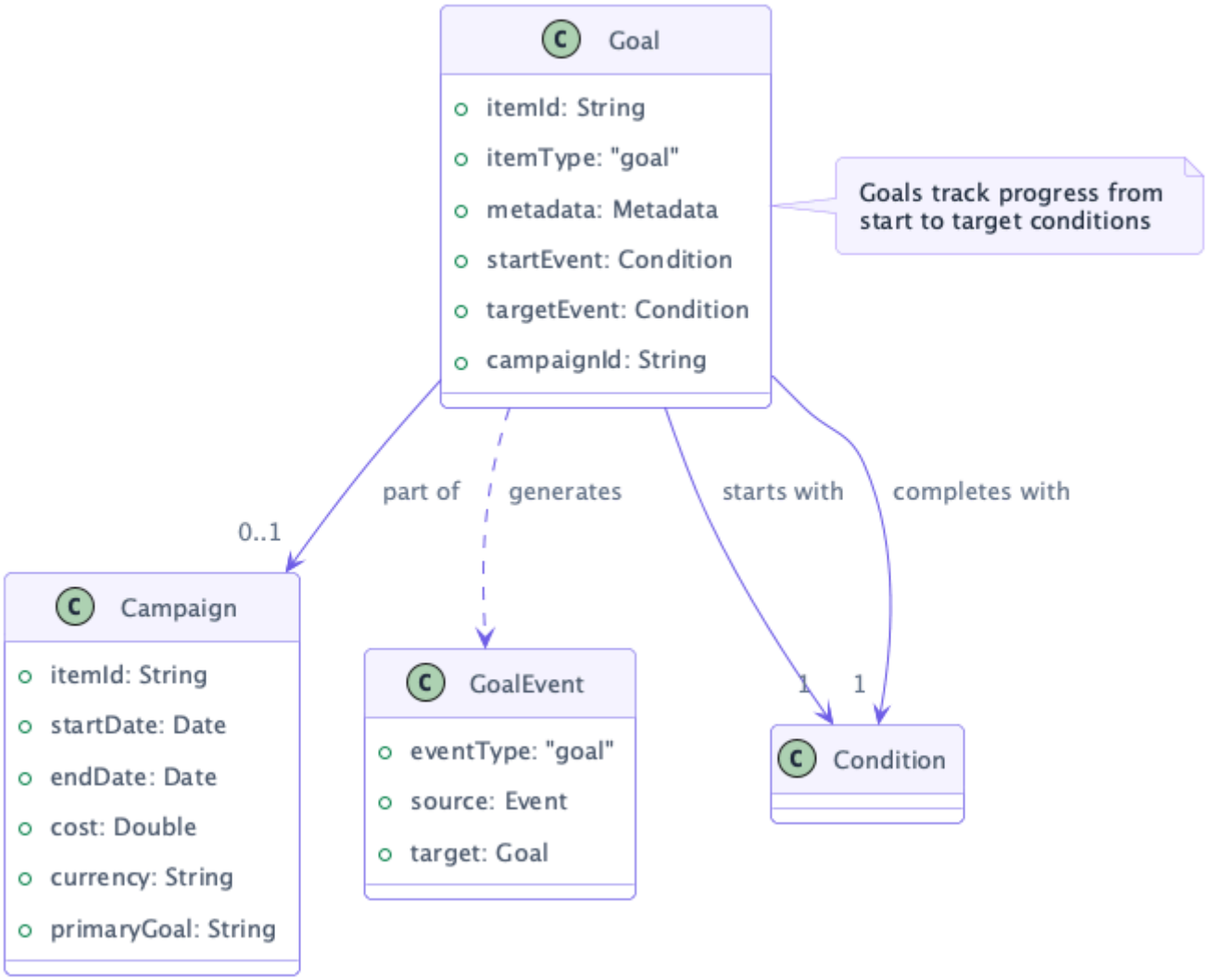
18.5.9. Rule Structure

Rules define automated actions triggered by conditions.



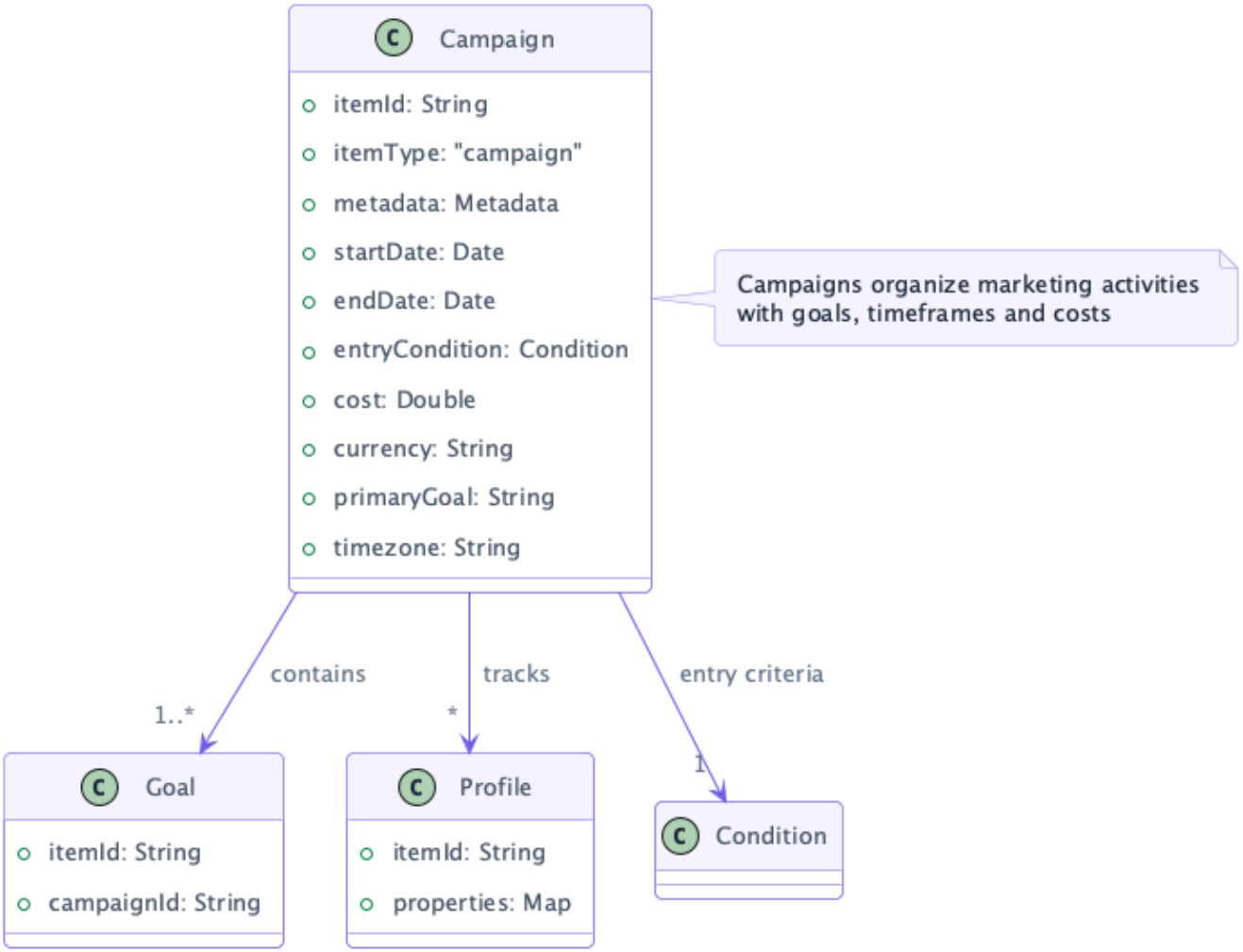
18.5.10. Goal Structure

Goals track visitor progress toward specific objectives.



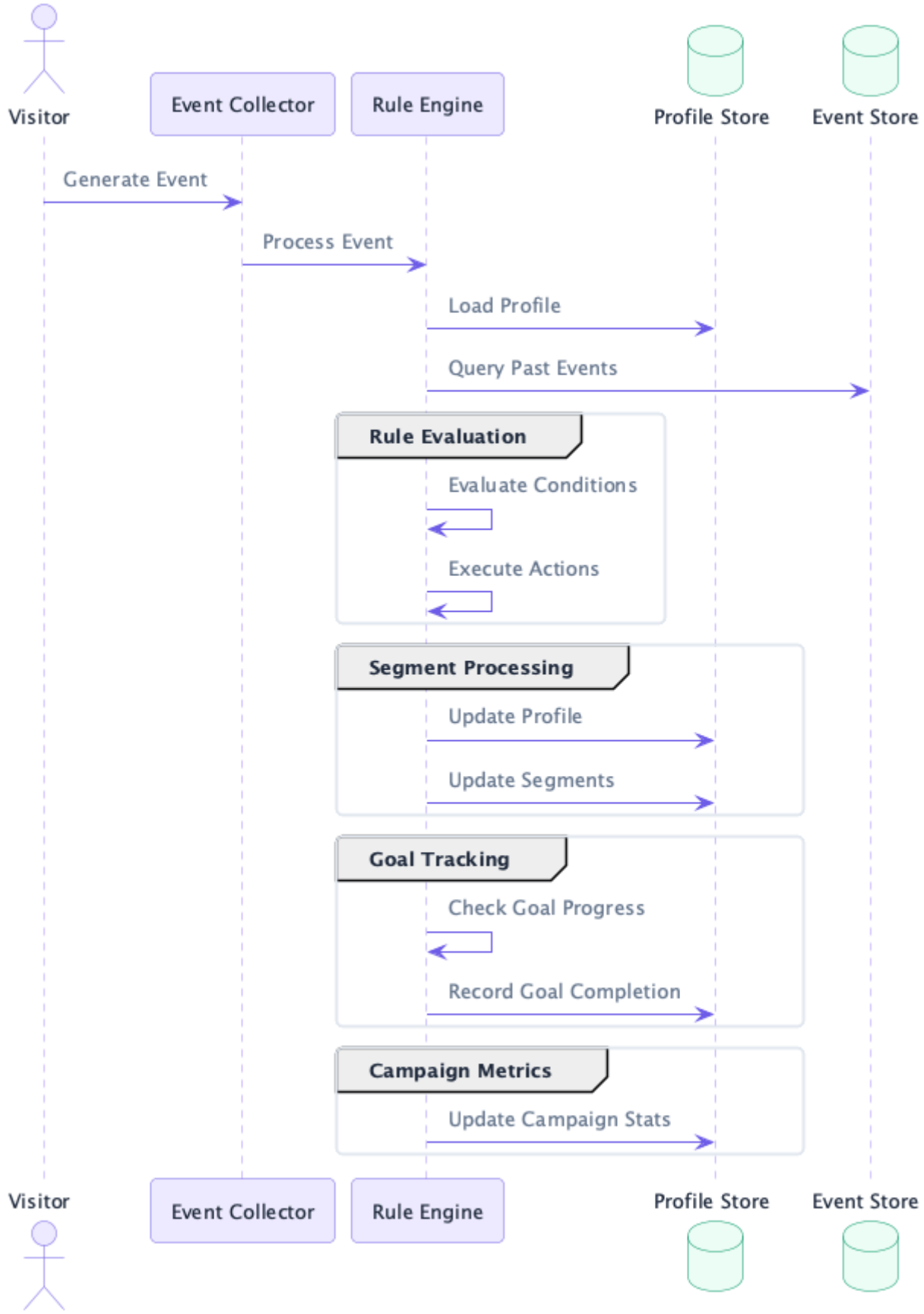
18.5.11. Campaign Structure

Campaigns organize marketing activities with goals, timeframes and costs.



18.5.12. Integration Flow

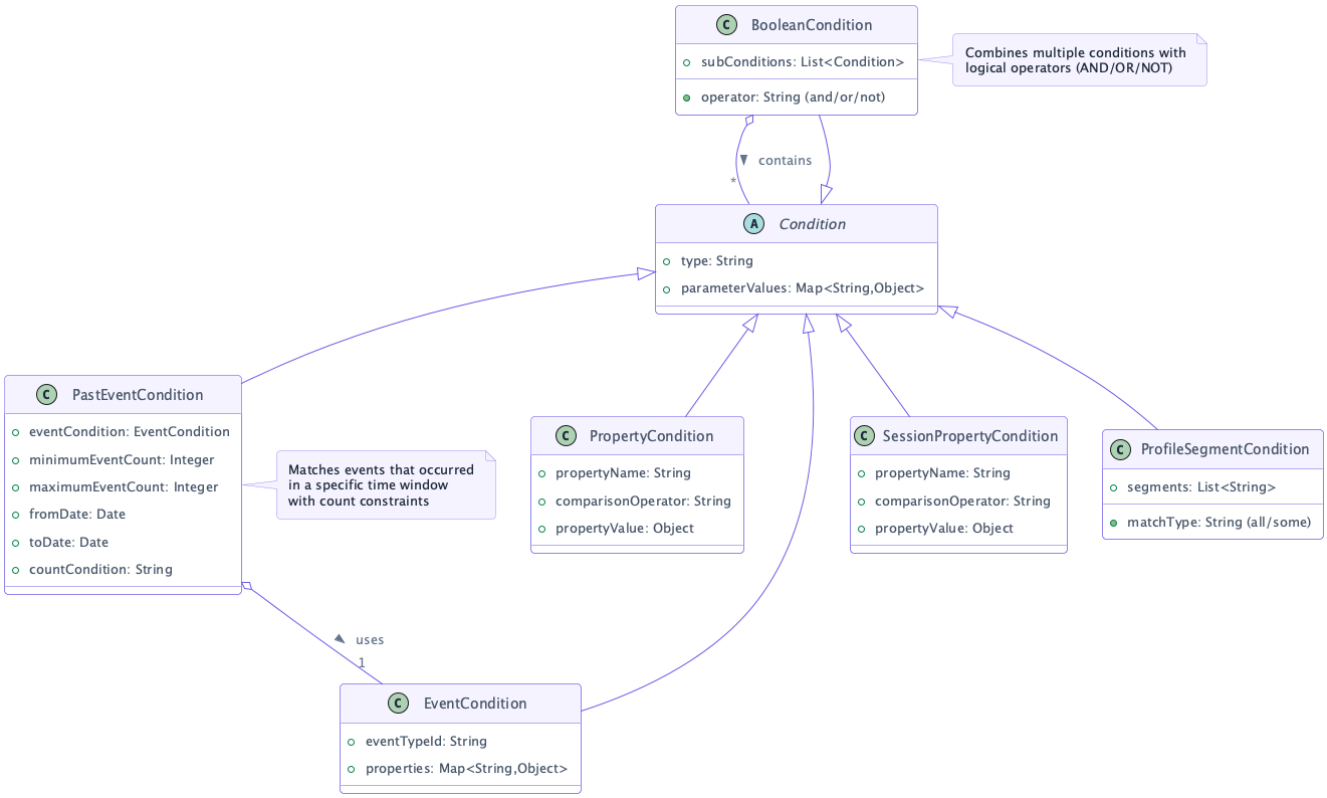
This diagram shows how these components work together in practice.



The diagrams above illustrate the key data structures in Apache Unomi and how they work together to enable personalization and marketing automation. Each component builds on the foundational concept of conditions to create increasingly sophisticated targeting and tracking capabilities.

18.5.13. Condition Composition Structure

The following diagram illustrates how conditions can be nested and composed to create complex matching logic:



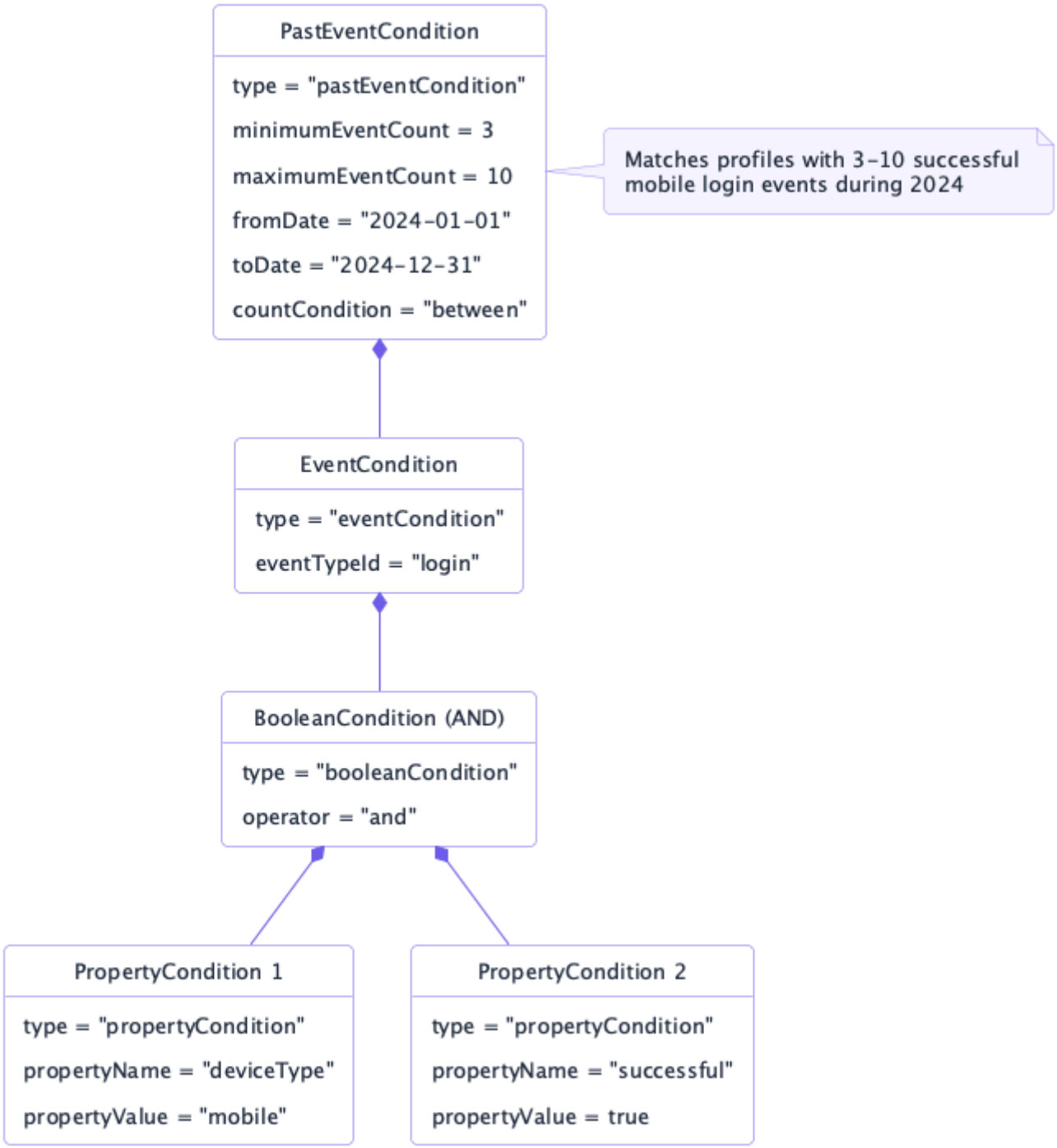
Example Condition Compositions

Here are some examples of how conditions can be composed:



Past Event Condition Structure

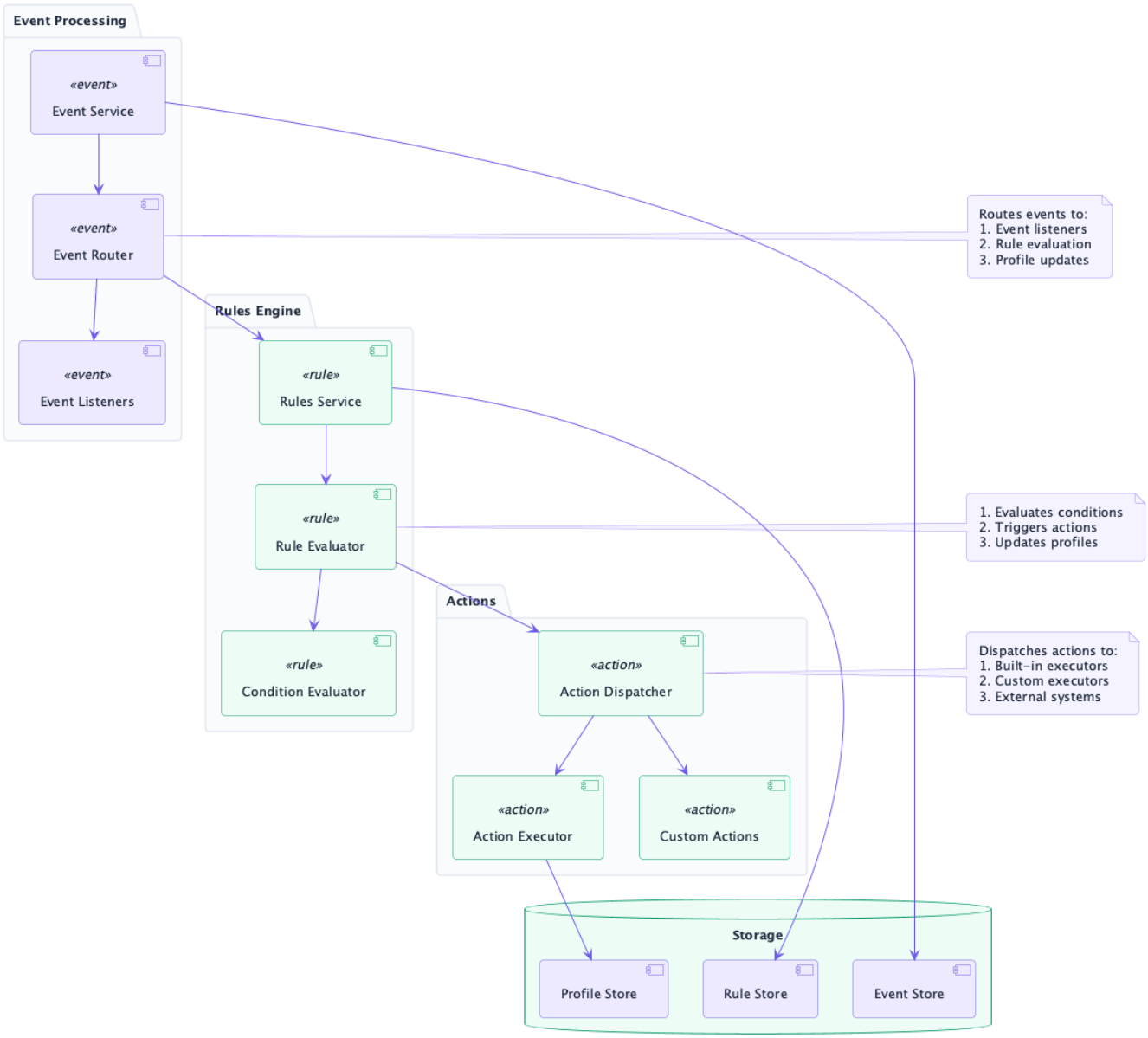
Past event conditions have a special structure that allows for complex temporal matching:



These diagrams show how conditions can be: 1. Nested using BooleanCondition to create complex logical expressions 2. Combined with PastEventCondition to match historical behavior 3. Enhanced with property constraints at any level 4. Used to create sophisticated temporal and behavioral matching rules

18.5.14. Event Processing Architecture

18.5.15. Event Processing Architecture



Overview

18.5.16. Security Architecture

18.5.17. Security Architecture

Unomi 4.0 authenticates every request so the server can resolve a **tenant** (or system-admin context) before profiles, events, and rules are touched.

Authentication overview

Path class	Credentials	Typical use
Public (/cxs/context.json, /cxs/eventcollector, /cxs/client/*, …)	X-Unomi-Api-Key: <publicApiKey>	Browser / mobile SDKs, trackers

Path class	Credentials	Typical use
Private REST	Basic tenantId:privateApiKey, or JAAS + X-Unomi-Tenant-Id (optional; see below)	Server-side integrations, admin UIs
Tenant administration (/cxs/tenants)	JAAS system administrator (for example karaf:$UNOMI_ROOT_PASSWORD)	Create tenants, rotate keys

When a JAAS administrator authenticates without X-Unomi-Tenant-Id, Unomi uses the configured admin default tenant (org.apache.unomi.rest.authentication.defaultTenantId / UNOMI_REST_AUTHENTICATION_DEFAULTTENANTID, default default). Set it to the tenant id chosen during the 3.0–4.0 migration (also default unless changed). An unknown tenant id, or a missing default tenant, returns HTTP 400 after authentication succeeds, on REST and GraphQL. Wrong credentials always return HTTP 401, whether or not the tenant exists. For tenant data, the system context is never the silent default: send X-Unomi-Tenant-Id: system, and only a full administrator (ROLE_UNOMI_ADMIN) may do so. Tenant administration (/cxs/tenants) is the exception: it manages the tenant registry, so a full administrator without the header works in the system context, even before any tenant exists.

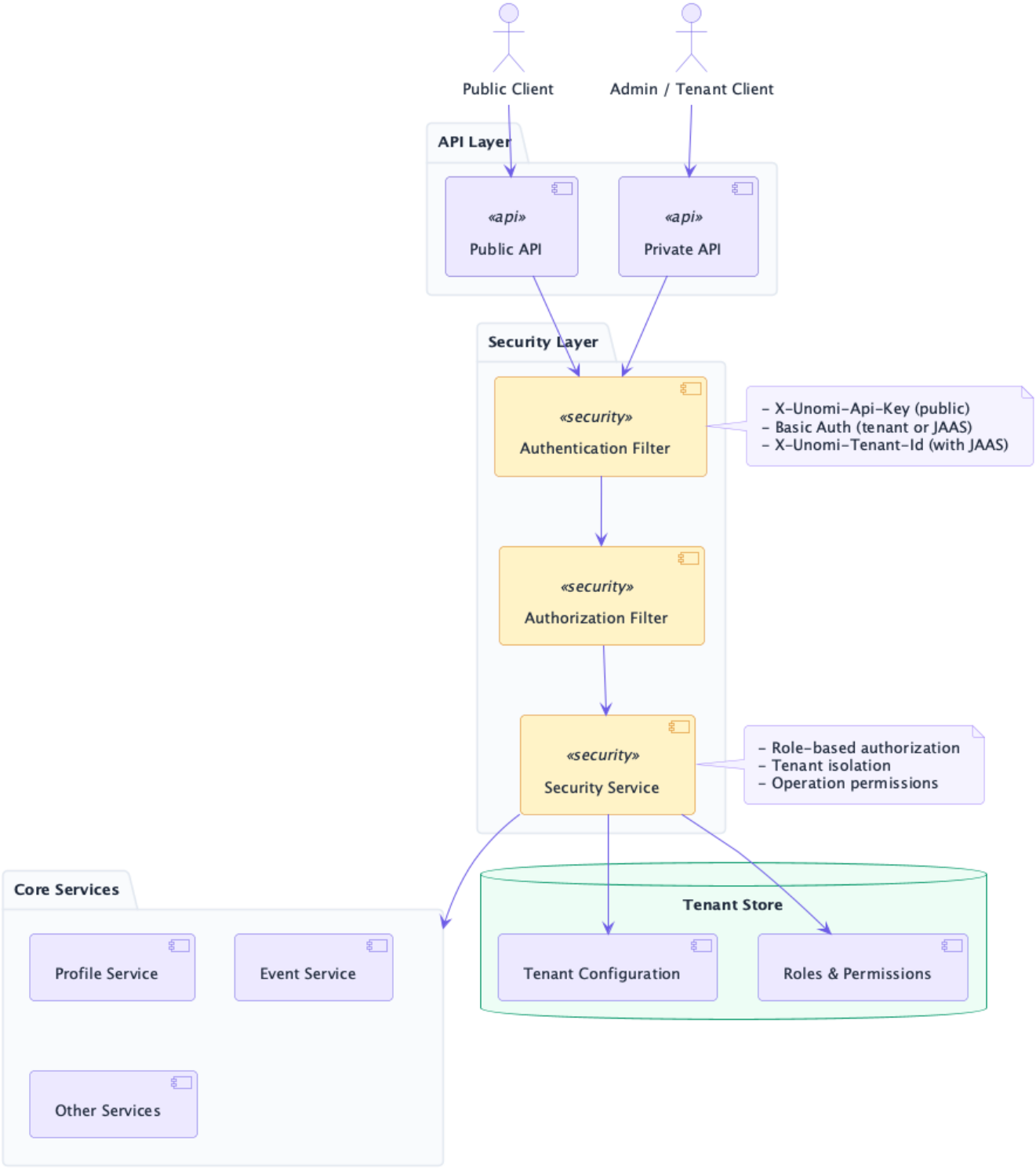
Temporary exception: single-tenant compatibility mode allows public endpoints without API keys while migrating Unomi 2 or Unomi 3 clients. In that mode every REST request, JAAS admin requests included, runs on tenant default and the default tenant setting is not used yet; GraphQL still follows the setting, so keep it at default while the mode is on.

Public context callers must present the profile cookie as the profile bearer (body profileId is ignored for public callers). See client-facing hardening and How profile tracking works.

Auth resolution sequence



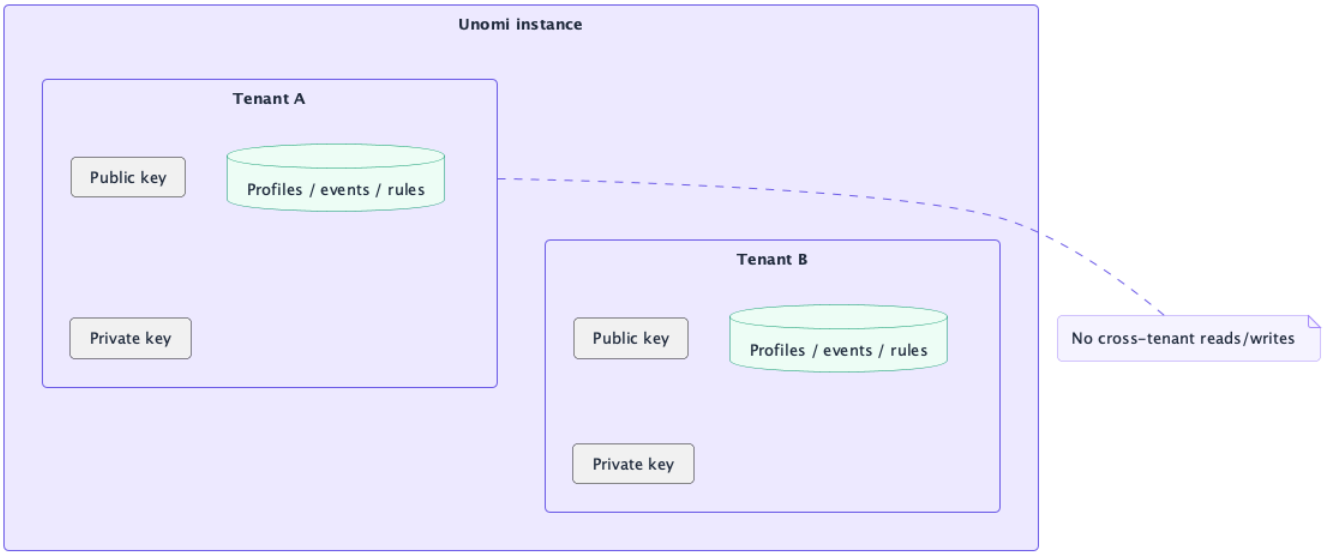
Component view



Configuration keys live in etc/org.apache.unomi.rest.authentication.cfg and custom.system.properties. Tenant API key settings are in etc/org.apache.unomi.tenant.cfg.

For tenant API keys, roles (ROLE_UNOMI_TENANT_USER, ROLE_UNOMI_TENANT_ADMIN, system administrator), operation permissions, and curl examples, see Multi-tenancy and Configuration (tenant management). For migrating Unomi 2 or Unomi 3 clients, see single-tenant compatibility mode and Migrate from 3.0 to 4.0.

Multi-tenancy isolation



18.5.18. Request tracing

Administrators with the appropriate roles can enable request tracing with explain=true on context and event-collector requests. See Using the explain parameter for request tracing.

18.6. Useful Apache Unomi URLs

In this section we will list some useful URLs that can be used to quickly access parts of Apache Unomi that can help you understand or diagnose what is going on in the system.

You can of course find more information about the REST API in the related section in the Apache Unomi website.

For these requests it can be nice to use a browser (such as Firefox) that understands JSON to make it easier to view the results as the returned JSON is not beautified (another possiblity is a tool such as Postman).

Important : all URLs are relative to the private Apache Unomi URL, by default: https://localhost:9443

Table 6. Useful URLs

Path	Method	Description
/cxs/profiles/properties	GET	Listing deployed properties
/cxs/definitions/conditions	GET	Listing deployed conditions
/cxs/definitions/actions	GET	Listing deployed actions
/cxs/profiles/PROFILE_ID	GET	Dumping a profile in JSON
/cxs/profiles/PROFILE_ID/sessions	GET	Listing sessions for a profile
/cxs/profiles/sessions/SESSION_ID	GET	Dumping a session in JSON

Path	Method	Description
/cxs/profiles/sessions/SESSION_ID/events	GET	Listing events for a session. This query can have additional such as eventTypes, q (query), offset, size, sort. See the related section in the REST API for details.
/cxs/events/search	POST	Listing events for a profile. You will need to provide a query in the body of the request that looks something like this (and documentation is available in the REST API) : { "offset" : 0, "limit" : 20, "condition" : { "type": "eventPropertyCondition", "parameterValues" : { "propertyName" : "profileId", "comparisonOperator" : "equals", "propertyValue" : "PROFILE_ID" } } } where PROFILE_ID is a profile identifier. This will indeed retrieve all the events for a given profile.
/cxs/rules/statistics	GET	Get all rule execution statistics
/cxs/rules/statistics	DELETE	Reset all rule execution statistics to 0
/cxs/tasks	GET	Listing scheduled background tasks (see Task scheduler) (filter with ?status= or ?type=)
/cxs/tasks/{taskId}	GET	View a single scheduled task
/cxs/tasks/{taskId}	DELETE	Cancel a scheduled task
/cxs/tasks/{taskId}/retry	POST	Retry a failed task (?resetFailureCount=true optional)
/cxs/tasks/{taskId}/resume	POST	Resume a crashed task from checkpoint
/cxs/tenants	GET	List all tenants (system administrator only). See Multi-tenancy.
/cxs/tenants/{tenantId}	GET	Get one tenant (system administrator or tenant administrator for that tenant).

Path	Method	Description
/cxs/tenants	POST	Create a tenant (system administrator). Returns the tenant with **masked** API key metadata; obtain plaintext via POST /cxs/tenants/{id}/apikeys.
/cxs/tenants/{tenantId}/apikeys	POST	Generate a new API key (?type=PUBLIC or PRIVATE, optional validityDays).
/cxs/tenants/{tenantId}/usage	GET	Tenant usage snapshot (?period=current-month or YYYY-MM).
/cxs/tenants/{tenantId}/purge/events	POST	Purge events older than retentionDays (destructive; tenant administrator).
/cxs/cluster	GET	List cluster nodes and statistics. See Cluster setup.
/health/check	GET	Health check JSON (role health, user health with the password from UNOMI_HEALTHCHECK_PASSWORD — no default is shipped). See Health Check extension.
/cxs/context.json?explain=true	POST	Context request with request tracing (admin roles). See Using the explain parameter for request tracing.

18.7. How profile tracking works

In this section you will learn how Apache Unomi keeps track of visitors and processes context requests.

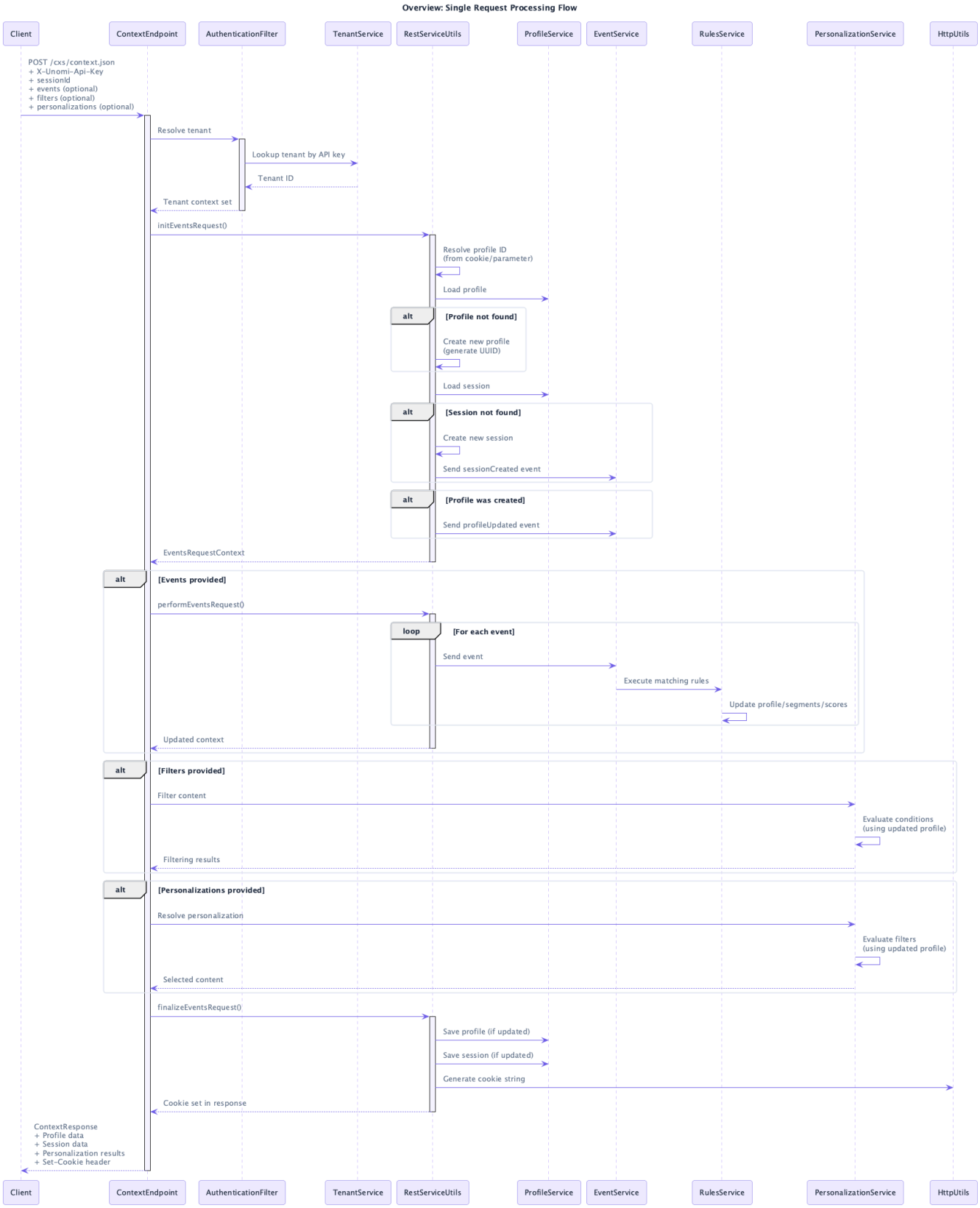
18.7.1. Overview

When a visitor interacts with a website that uses Apache Unomi, a single request to the /cxs/context.json or /cxs/context.js endpoint can perform multiple operations:

- Automatically create or load a visitor profile
- Automatically create or load a visitor session
- Process events (if provided)
- Execute rules triggered by those events
- Resolve personalization (if requested)

- Set a cookie with the profile ID for future requests

All of this happens in a single request-response cycle, making Apache Unomi efficient and easy to integrate.



18.7.2. Detailed Request Flow

When a request is made to /cxs/context.json or /cxs/context.js, Apache Unomi processes it through the following steps:

Tenant Resolution (Version 4.0+)

Starting with Apache Unomi 4.0, tenant resolution is mandatory for all requests to /cxs/context.json and /cxs/eventcollector endpoints. Apache Unomi must know which tenant's data to use before processing the request. The tenant is resolved through one of the following methods (in priority order):

1. **Public API Key Authentication** (Recommended for public endpoints):
   - Send the X-Unomi-Api-Key header with a public API key
   - Apache Unomi automatically resolves the tenant from the API key
   - This is the recommended method for public-facing applications
2. **Basic Authentication with Private API Key**:
   - Use Basic Auth with format: tenantId:privateApiKey
   - The tenant is resolved from the credentials
   - This is used for administrative or server-to-server requests
3. **Tenant ID Header** (When using JAAS authentication):
   - Send the X-Unomi-Tenant-Id header with the tenant ID
   - Used when authenticating via JAAS (e.g., the karaf user and your configured UNOMI_ROOT_PASSWORD)
   - The tenant ID must exist; an unknown id returns HTTP 400 after authentication succeeds
   - If the header is omitted, Unomi uses the configured admin default tenant (default default)
   - The system context needs an explicit X-Unomi-Tenant-Id: system and a full administrator

Wrong credentials return HTTP 401 whether or not the tenant exists. A missing admin default tenant also fails with HTTP 400, on REST and GraphQL. Tenant administration (/cxs/tenants) does not need the header: a full administrator then works in the system context, even before any tenant exists.

Example: Tenant Resolution

Here are examples of how to authenticate and resolve the tenant:

Example 1: Using Public API Key (Recommended)

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "X-Unomi-Api-Key: public-key-abc123..." \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   }
}'
```

The tenant is automatically resolved from the public API key. No additional headers needed.

Example 2: Using Basic Auth with Private API Key

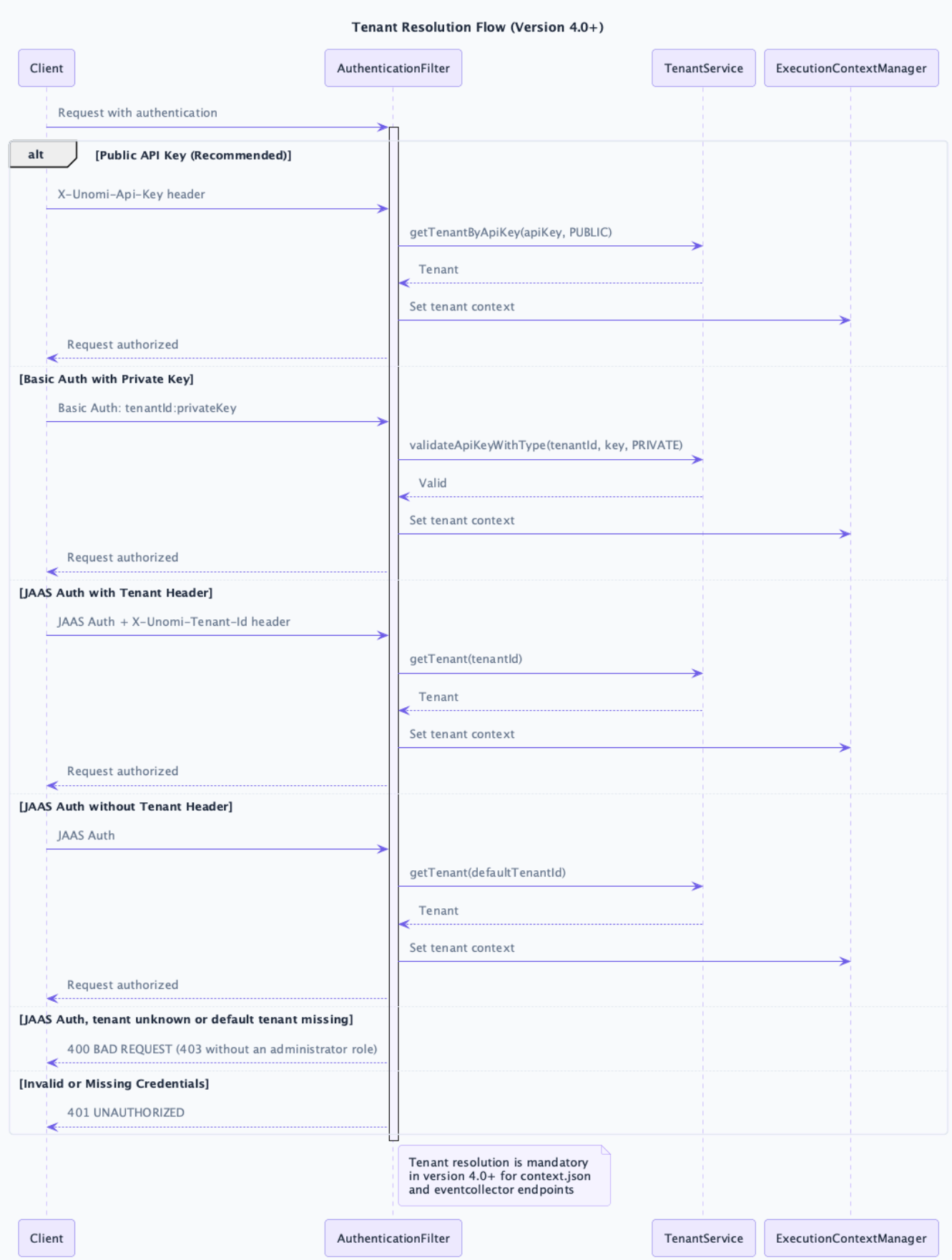
```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
--user "mytenant:private-key-xyz789..." \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   }
}'
```

The tenant is resolved from the Basic Auth credentials (mytenant is the tenant ID).

Example 3: Using JAAS Auth with Tenant Header

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
--user "karaf:$UNOMI_ROOT_PASSWORD" \
-H "X-Unomi-Tenant-Id: mytenant" \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   }
}'
```

With JAAS authentication, the tenant comes from the X-Unomi-Tenant-Id header, or from the configured default tenant when the header is omitted.



Step 1: Profile Identification and Creation



At least one of the following must be provided in the request: sessionId, a profile cookie (context-profile-id by default), a body/query profileId **from a trusted caller**, or personaId. If none of these are available after public-caller rules below, the request fails with a BadRequestException (400).

Apache Unomi attempts to identify the visitor's profile through the following process:

1. **Profile ID Resolution** (public vs trusted callers):
    - **Public callers** (public API key / unauthenticated context): the profile cookie is the **only** profile bearer. A body or query profileId is **ignored** (even when no cookie is present).
    - **Trusted callers** (system administrator or tenant private-key / TENANT_ADMINISTRATOR): an explicit body/query profileId is honored and may differ from the cookie.
    - Cookie name defaults to context-profile-id (configurable via org.apache.unomi.profile.cookie.name).
    - Cookie values are validated against a JSON schema — invalid values (for example containing script tags) cause a 400 Bad Request. Since 4.0 the cookie is read on every request, so this also applies when the request supplies an explicit profileId; before, a malformed cookie went unnoticed on those requests.
    - The resolved profile ID is used to attempt loading the profile from the database.
2. **Session Profile Override** (if session exists):
    - If a session is found (see Step 2) and its profile differs from the request profile, Unomi switches to the session profile **only when**:
        - the profile cookie already matches the session owner, **or**
        - the caller is trusted (and is not keeping an explicit trusted body profileId override).
    - Otherwise the session is **detached** for this request (Unomi does not adopt a foreign session profile for a public caller), and the supplied session id is **not echoed back** in the response — a client that saw its own id returned would keep replaying an id the server did not accept.
    - **Anonymous sessions** are a special case. Anonymity removes the owner from the session (getAnonymousProfile returns a profile with no ID, so the session's profileId is null), which leaves the rule above nothing to compare the cookie against. Binding such a session back to a named profile — what happens when a visitor turns anonymous browsing off — is therefore reserved for **trusted** callers. For a public caller the session stays anonymous, and the visitor gets a named session again once their client uses a new session id.
    - invalidateSession=true re-creates the session under the supplied id, so it observes the same rule: a public caller may only invalidate a session its own cookie owns, and a session with no recorded owner cannot be invalidated this way.
3. **Profile Creation**: If no profile ID is found or the profile doesn't exist:
    - If a profile ID was provided (from cookie, or from a trusted body/query profileId) but doesn't exist in the database, creates a new profile with that ID
    - If no profile ID was found, generates a new UUID as the profile ID and creates a new profile

- Sets the firstVisit property to the current timestamp
- Marks the profile for persistence
- Note: The profileUpdated event is sent after session creation (if a session is created) to ensure proper event ordering

This ensures that profiles are always available for processing, even for first-time visitors or after database resets.



For **public** callers, do not rely on body/query profileId as identity — always send the profile cookie (browsers do this automatically when credentials are included). To act as a specific profile from a backend or admin tool, authenticate as a **trusted** caller (tenant private key or system administrator) and then send profileId. See Migrate from 3.0 to 4.0 for client migration notes.

Why the cookie is safer than a body profileId for public callers

On public /context.json and /eventcollector endpoints, the profile id is a **bearer** identifier: whoever presents it can continue that visitor's tracking context. Unomi therefore prefers the **HTTP cookie** over a body or query profileId for public traffic:

- **Browser-enforced delivery** — Once Set-Cookie has established the profile cookie (with HttpOnly and SameSite by default), the browser attaches it on later requests to Unomi. Application JavaScript does not need to copy the id into JSON, so routine front-end code is less likely to mishandle or over-share it.
- **Harder for page script to exfiltrate** — With HttpOnly (the default), script running in the page cannot read the cookie via document.cookie. A body/query profileId is application-controlled data: anything that can influence the request payload can try to point at another profile id.
- **Clearer trust boundary** — Public callers prove continuity with the cookie the server previously issued. Selecting an arbitrary id in the body blurred that boundary (knowing or guessing a UUID was enough). Trusted callers (private key / system admin) may still pass profileId explicitly when a backend intentionally acts on a chosen profile.
- **Fewer accidental leaks in URLs and logs** — Query-string profileId values show up in browser history, proxies, and access logs more often than Cookie headers. Prefer the cookie (and read profileId from the JSON **response** when your app needs the value in memory).

Body/query profileId remains appropriate for **authenticated** integrations that are supposed to bind a specific profile under operator control — not for anonymous browser trackers.

Example: Profile Identification

Example 1: First-Time Visitor (No Cookie)

```
# Request (no cookie sent)
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   }
}'
```

Response Headers:

```
Set-Cookie: context-profile-id=a1b2c3d4-e5f6-7890-abcd-ef1234567890; Path=/; Max-Age=31536000;
SameSite=Lax
```

Response Body:

```
{
   "profileId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
   "sessionId": "1234"
}
```

Apache Unomi generated a new UUID (a1b2c3d4-e5f6-7890-abcd-ef1234567890) and created a new profile. The profile ID is returned in both the response body and the Set-Cookie header.

Example 2: Returning Visitor (With Cookie)

```
# Request (browser automatically sends cookie)
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Cookie: context-profile-id=a1b2c3d4-e5f6-7890-abcd-ef1234567890" \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   }
}'
```

Response:

```
{
   "profileId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
   "sessionId": "1234"
}
```

Apache Unomi loaded the existing profile using the profile ID from the cookie.

Example 3: Trusted caller using body/query profileId

Public callers must send the cookie (Example 2). A trusted caller (tenant private key) may select a profile explicitly:

```
# Trusted: Basic auth with tenantId:privateKey — body/query profileId is honored
curl -X POST "http://localhost:8181/cxs/context.json?sessionId=1234&profileId=a1b2c3d4-e5f6-7890-
abcd-ef1234567890" \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   }
}'
```

For public callers, a body/query profileId without a matching cookie is ignored (it does **not** take precedence over the cookie).


Profile and Session Identification Flow
RestServiceUtils
ProfileService
HttpServletRequest
EventService
alt
[trusted caller with body/query profileId]
load(bodyOrQueryProfileId)
[public or no explicit trusted profileId]
getCookie("context-profile-id")
profileId from cookie
load(profileId)
alt
[Profile found]
Use existing profile
[Profile not found]
createNewProfile(profileId or null)
Generate UUID if needed
Set firstVisit property
Send profileUpdated event
Mark for persistence
alt
[sessionId provided]
loadSession(sessionId)
alt
[Session found]
Check session profile
alt
[Session profile differs and (cookie owns session or trusted)]
load(sessionProfileId)
Use session's profile
[public mismatch]
Detach session for this request
[Session not found]
Create new Session(sessionId, profile)
Send sessionCreated event
Mark for persistence
RestServiceUtils
ProfileService
HttpServletRequest
EventService

Step 2: Session Identification and Creation

If a sessionId is provided in the request:

1. **Session Loading**: Apache Unomi attempts to load the existing session from the database
2. **Profile Association**: If a session is found and its profile differs from the request profile, Unomi switches only when the cookie owns that session or the caller is trusted (see Step 1). Public callers cannot adopt a foreign session profile.
3. **Session Creation**: If no session is found or the session is invalidated:
    - Creates a new session with the provided sessionId
    - Associates the session with the current profile (or anonymous profile if privacy settings require it)
    - Sends a sessionCreated event (this happens before the profileUpdated event if both profile and session are created)
    - Marks the session for persistence
    - After session creation, if a new profile was created, sends a profileUpdated event (non-persistent) to mark the profile creation

If no sessionId is provided, a transient (non-persisted) session may be used for the request.

Example: Session Creation

Example: Creating a New Session

```
# First request with a new sessionId
curl -X POST http://localhost:8181/cxs/context.json?sessionId=abc-123-xyz \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   }
}'
```

Apache Unomi will: 1. Check if session abc-123-xyz exists 2. If not found, create a new session with ID abc-123-xyz 3. Associate it with the current profile 4. Send a sessionCreated event 5. Save the session to the database

Subsequent Request with Same Session:

```
# Request with existing sessionId
curl -X POST http://localhost:8181/cxs/context.json?sessionId=abc-123-xyz \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Cookie: context-profile-id=a1b2c3d4-e5f6-7890-abcd-ef1234567890" \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "product-page",
      "itemType": "page",
      "scope": "mydigital"
   }
}'
```

Apache Unomi loads the existing session abc-123-xyz and uses the profile associated with it.

Step 3: Event Processing

If events are provided in the request body:

1. **Event Validation**: Each event is validated against its JSON schema
2. **Event Authorization**:
    - Events are checked for tenant authorization (the tenant must be authorized to send the event type)
    - Events are filtered based on privacy settings (if the profile has opted out of certain event types)
    - Events are checked against IP address restrictions if configured
3. **Event Execution**: For each valid event:
    - The event is sent to the event service
    - All matching rules are automatically executed (this happens synchronously during the request)
    - Rules can trigger actions that modify the profile, session, or trigger other events
    - Profile segments and scores are recalculated based on rule execution
    - If the profile is updated during rule execution, the updated profile is used for subsequent events in the same request
4. **Profile Updates**: If events cause profile changes, the profile is marked for persistence

This is where the power of Apache Unomi's rule engine comes into play - events automatically trigger rules, which can perform complex logic, update profiles, add segments, calculate scores, and more. All of this happens within the same request, ensuring that personalization and subsequent operations use the most up-to-date profile state.

Example: Event Processing with Rule Execution

Example: Sending a View Event

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Cookie: context-profile-id=a1b2c3d4-e5f6-7890-abcd-ef1234567890" \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "product-page",
      "itemType": "page",
      "scope": "mydigital"
   },
   "events": [
      {
         "eventType": "view",
         "scope": "mydigital",
         "source": {
            "itemType": "site",
            "scope": "mydigital",
            "itemId": "mysite"
         },
         "target": {
            "itemType": "product",
            "scope": "mydigital",
            "itemId": "product-123",
            "properties": {
               "name": "Awesome Product",
               "category": "electronics",
               "price": 99.99
            }
         }
      }
   ]
}'
```

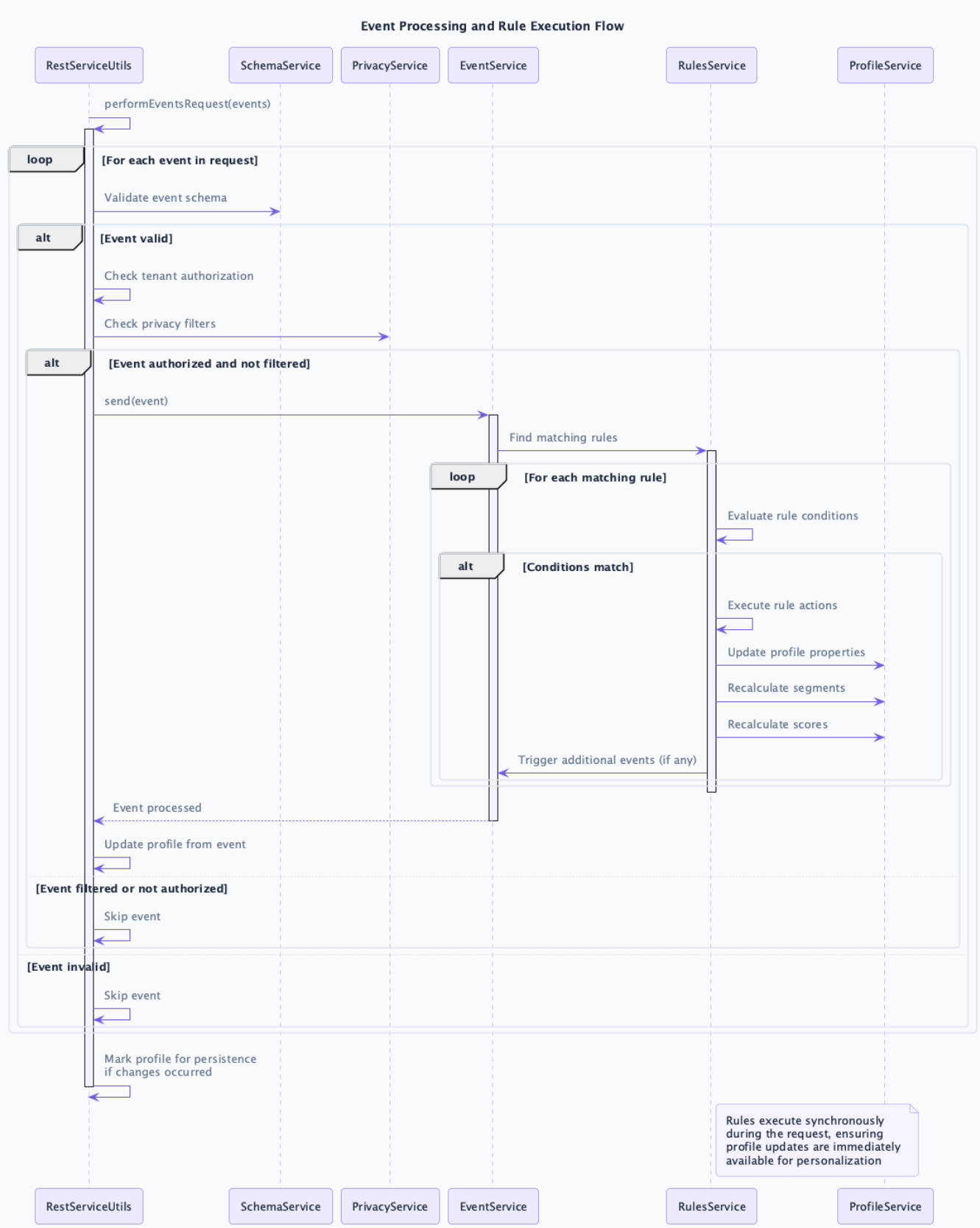
What Happens:

1. The view event is validated against its schema
2. The event is checked for tenant authorization
3. The event is sent to the event service
4. All rules matching the view event are executed:
    - Rules might increment a pageViewCount property
    - Rules might add the profile to a "product-viewers" segment
    - Rules might calculate a score based on product category
    - Rules might trigger additional events
5. The profile is updated with any changes from rule execution
6. The updated profile is used for personalization (if requested)

Response:

```
{
  "profileId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "sessionId": "1234",
  "processedEvents": 1
}
```


The processedEvents field indicates that 1 event was successfully processed. The profile may have been updated by rules triggered by this event.



Step 4: Content Filtering and Personalization Resolution

If filters or personalization requests are provided in the request body:

1. **Content Filtering** (if filters are provided):
    - For each filter node, evaluates conditions against the current profile and session

- Returns filtering results indicating which content variants match
- Results are added to the response's filteringResults object

2. **Personalization Resolution** (if personalizations are provided):
   - For each personalization request:
   - Evaluates content filters against the current profile and session (which may have been updated by events in Step 3)
   - Filters use conditions to determine which content variants match
   - Applies the personalization strategy (e.g., "matching-first", "random", etc.)
   - **Content Selection**: Selects the appropriate content based on:
   - Filter condition evaluation results
   - Personalization strategy
   - Fallback content (if no filters match)
   - Results are added to both personalizationResults (detailed results) and personalizations (selected content IDs) in the response

Note that personalization uses the profile state after event processing, ensuring that events can influence which content is selected.

Example: Personalization Resolution

Example: Request with Events and Personalization

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=1234 \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Cookie: context-profile-id=a1b2c3d4-e5f6-7890-abcd-ef1234567890" \
-H "Content-Type: application/json" \
-d '{
  "source": {
    "itemId": "homepage",
    "itemType": "page",
    "scope": "mydigital"
  },
  "events": [
    {
      "eventType": "view",
      "scope": "mydigital",
      "target": {
        "itemType": "page",
        "scope": "mydigital",
        "itemId": "homepage"
      }
    }
  ],
  "personalizations": [
    {
      "id": "homepage-hero",
```

```
        "strategy": "matching-first",
        "strategyOptions": {
          "fallback": "default-hero"
        },
        "contents": [
          {
            "id": "premium-user-hero",
            "content": "Welcome, Premium Member!",
            "filters": [
              {
                "condition": {
                  "type": "profileSegmentCondition",
                  "parameterValues": {
                    "segments": ["premium-users"]
                  }
                }
              }
            ]
          },
          {
            "id": "new-visitor-hero",
            "content": "Welcome! Sign up today!",
            "filters": [
              {
                "condition": {
                  "type": "profilePropertyCondition",
                  "parameterValues": {
                    "propertyName": "properties.firstVisit",
                    "comparisonOperator": "exists"
                  }
                }
              }
            ]
          }
        ]
      }
    ],
    "requireSegments": true
}'
```

Processing Flow:

1. **Event Processing**: The view event is processed first, which may trigger rules that:
   - Add the profile to segments
   - Update profile properties
   - Calculate scores
2. **Personalization Resolution**: After events are processed, personalization is evaluated:
   - For a first-time visitor: The new-visitor-hero content matches (because firstVisit exists)
   - For a premium user: The premium-user-hero content matches (if the profile is in the "premium-users" segment)

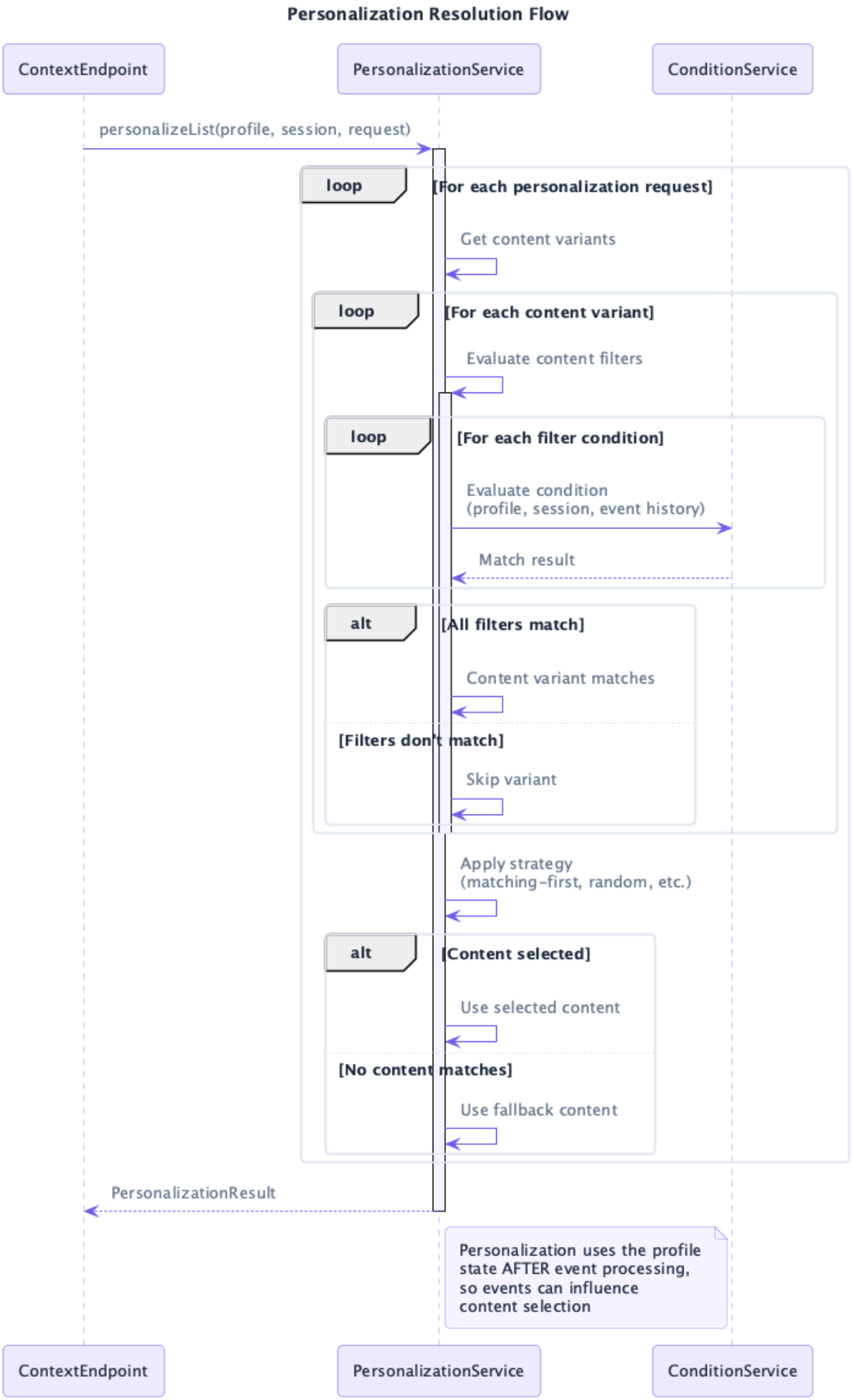
- The matching-first strategy selects the first matching content

3. **Response**: The response includes the selected content

Response:

```
{
  "profileId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "sessionId": "1234",
  "processedEvents": 1,
  "profileSegments": ["new-visitors"],
  "personalizations": {
    "homepage-hero": "new-visitor-hero"
  },
  "personalizationResults": {
    "homepage-hero": {
      "selectedContentId": "new-visitor-hero",
      "strategy": "matching-first"
    }
  }
}
```

The personalizations object shows the selected content ID for each personalization request. The personalizationResults provides detailed information about the selection process.


Personalization Resolution Flow
ContextEndpoint
PersonalizationService
ConditionService
personalizeList(profile, session, request)
loop
[For each personalization request]
Get content variants
loop
[For each content variant]
Evaluate content filters
loop
[For each filter condition]
Evaluate condition
(profile, session, event history)
Match result
alt
[All filters match]
Content variant matches
[Filters don't match]
Skip variant
Apply strategy
(matching-first, random, etc.)
alt
[Content selected]
Use selected content
[No content matches]
Use fallback content
PersonalizationResult
Personalization uses the profile state AFTER event processing, so events can influence content selection
ContextEndpoint
PersonalizationService
ConditionService

Step 5: Response Preparation

The response is built with the following information (based on what was requested in the ContextRequest):

- **Profile Information**:
- Profile ID (always included)
- Profile properties (if requiredProfileProperties is specified, with "*" for all properties)
- Profile segments (if requireSegments is true)
- Profile scores (if requireScores is true)
- Anonymous browsing status
- Consent information
- **Session Information**:
- Session ID (if session exists)
- Session properties (if requiredSessionProperties is specified)
- **Event Processing Results**:
- Count of processed events (processedEvents)
- **Personalization Results**:
- personalizations: Map of personalization ID to selected content IDs
- personalizationResults: Detailed results for each personalization request
- **Filtering Results**:
- filteringResults: Results of content filtering operations (if filters were provided)
- **Tracked Conditions**:
- Conditions that are being tracked for the current source (if not using a Persona)

Step 6: Cookie Setting

After processing, Apache Unomi sets a cookie in the HTTP response header (Set-Cookie) with:



Cookies are only set for regular profiles, not for Personas. If a personaId is used in the request, no cookie will be set in the response.

- **Cookie Name**: context-profile-id (configurable, default: context-profile-id)
- **Cookie Value**: The profile ID (UUID)
- **Cookie Attributes**:
- Path=/ - Available for all paths on the domain
- Max-Age=31536000 - Valid for 1 year by default (configurable via org.apache.unomi.profile.cookie.maxAgeInSeconds)

- SameSite=Lax - CSRF protection. Configurable via org.apache.unomi.profile.cookie.sameSite; None is used only with Secure, and only once origins can be listed (a non-empty server list, or tenant lists on, the default)
- Secure - Set if the request reached Unomi over HTTPS, directly or through a trusted proxy that sends X-Forwarded-Proto: https
- HttpOnly - Configurable via org.apache.unomi.profile.cookie.httpOnly (default: true; only an explicit false turns it off)
- Domain - Configurable via org.apache.unomi.profile.cookie.domain. Unomi's own host, or a parent domain of it; never the website's domain when the website is on another site

The cookie is used and set only for pages on Unomi's own origin, for origins listed in org.apache.unomi.public.cors.allowedOrigins or in the requesting tenant's publicCorsAllowedOrigins property, and for requests that do not come from a page on another site. A request from any other page is served with a new anonymous profile, and the visitor's cookie is left untouched. See Websites on other origins.

This cookie allows the browser to automatically send the profile ID on subsequent requests, enabling Apache Unomi to load the existing profile.

Example: Cookie Setting

Example: Cookie in Response Headers

When you make a request, Apache Unomi sets a cookie in the response. Here's what the cookie looks like:

Response Headers:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Set-Cookie: context-profile-id=a1b2c3d4-e5f6-7890-abcd-ef1234567890; Path=/; Max-Age=31536000;
SameSite=Lax
```

Cookie Breakdown: * context-profile-id - Cookie name (configurable) * a1b2c3d4-e5f6-7890-abcd-ef1234567890 - Profile ID (UUID) * Path=/ - Available for all paths on the domain * Max-Age=31536000 - Valid for 1 year (31,536,000 seconds) * SameSite=Lax - CSRF protection

Subsequent Requests:

On the next request, the browser automatically includes the cookie:

```
# Browser automatically sends cookie
GET /cxs/context.json?sessionId=1234 HTTP/1.1
Host: localhost:8181
X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY
Cookie: context-profile-id=a1b2c3d4-e5f6-7890-abcd-ef1234567890
```

Apache Unomi reads the cookie and loads the existing profile, maintaining continuity across requests.

Step 7: Persistence

Finally, Apache Unomi persists any changes:

- **Profile Persistence**: If the profile was created or updated, it is saved to the database
- **Session Persistence**: If the session was created or updated, it is saved to the database

18.7.3. First-Time Visitor Flow

For a first-time visitor making their first request:

1. No context-profile-id cookie exists
2. No profileId parameter is provided
3. Apache Unomi generates a new UUID and creates a new profile
4. If a sessionId is provided, a new session is created
5. The profile ID is returned in a cookie
6. The browser stores the cookie
7. On subsequent requests, the browser automatically sends the cookie
8. Apache Unomi loads the existing profile using the cookie value

Complete Example: First-Time Visitor Journey

Request 1: First Visit (No Cookie)

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=visitor-123 \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Content-Type: application/json" \
-d '{
   "source": {
      "itemId": "homepage",
      "itemType": "page",
      "scope": "mydigital"
   },
   "events": [
      {
         "eventType": "view",
         "scope": "mydigital",
         "target": {
            "itemType": "page",
            "scope": "mydigital",
            "itemId": "homepage"
         }
      }
   ],
   "requiredProfileProperties": ["properties.firstVisit"]
}'
```

Response Headers:

```
Set-Cookie: context-profile-id=7f8e9d0c-1b2a-3456-7890-abcdef123456; Path=/; Max-Age=31536000;
SameSite=Lax
```

Response Body:

```
{
   "profileId": "7f8e9d0c-1b2a-3456-7890-abcdef123456",
   "sessionId": "visitor-123",
   "processedEvents": 1,
   "profileProperties": {
      "firstVisit": "2024-01-15T10:30:00Z"
   }
}
```

What Happened: * New profile created with UUID 7f8e9d0c-1b2a-3456-7890-abcdef123456 * firstVisit property set to current timestamp * New session created with ID visitor-123 * sessionCreated event sent (triggers any matching rules) * profileUpdated event sent (triggers any matching rules) * view event processed (may have triggered rules) * Cookie set in response header * Profile and session saved to database

Request 2: Second Visit (Cookie Present)

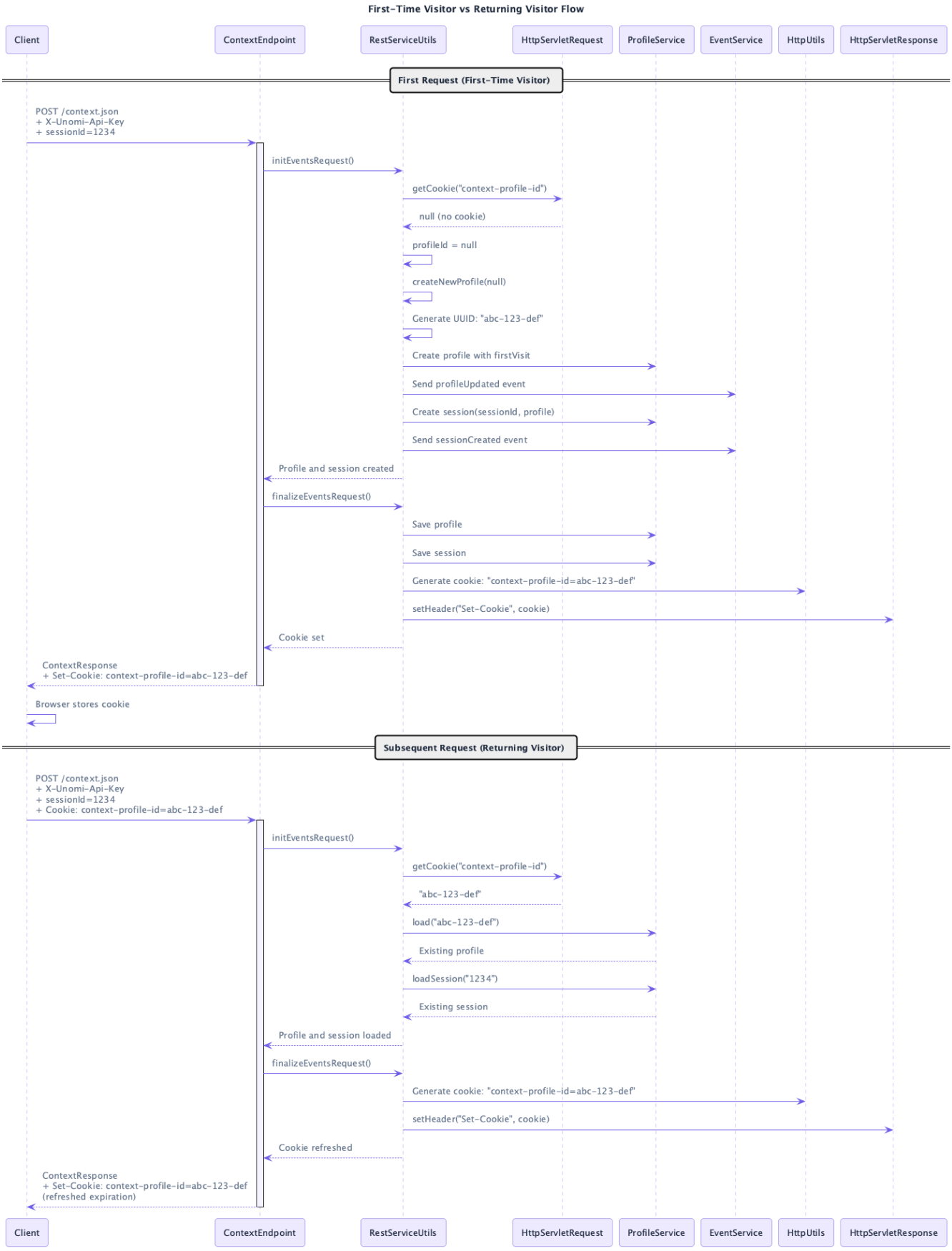
```
# Browser automatically includes the cookie from Request 1
curl -X POST http://localhost:8181/cxs/context.json?sessionId=visitor-123 \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Cookie: context-profile-id=7f8e9d0c-1b2a-3456-7890-abcdef123456" \
-H "Content-Type: application/json" \
-d '{
  "source": {
    "itemId": "product-page",
    "itemType": "page",
    "scope": "mydigital"
  },
  "events": [
    {
      "eventType": "view",
      "scope": "mydigital",
      "target": {
        "itemType": "product",
        "scope": "mydigital",
        "itemId": "product-456"
      }
    }
  ],
  "requiredProfileProperties": ["properties.firstVisit", "properties.pageViewCount"]
}'
```

Response:

```
{
  "profileId": "7f8e9d0c-1b2a-3456-7890-abcdef123456",
  "sessionId": "visitor-123",
  "processedEvents": 1,
  "profileProperties": {
    "firstVisit": "2024-01-15T10:30:00Z",
    "pageViewCount": 2
  }
}
```

What Happened: * Existing profile loaded using cookie value 7f8e9d0c-1b2a-3456-7890-abcdef123456 * Existing session loaded using visitor-123 * view event processed, which may have triggered a rule that incremented pageViewCount * Cookie refreshed in response (expiration extended) * Updated profile saved to database

The profile is now being tracked across multiple requests, and any rules triggered by events can build up a profile of the visitor's behavior.



18.7.4. Complete Example: Full Request Flow

Here's a complete example showing all aspects of a context request:

Request:

```
curl -X POST http://localhost:8181/cxs/context.json?sessionId=example-session \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-H "Cookie: context-profile-id=existing-profile-uuid" \
-H "Content-Type: application/json" \
-d '{
  "source": {
    "itemId": "checkout-page",
    "itemType": "page",
    "scope": "mydigital"
  },
  "events": [
    {
      "eventType": "view",
      "scope": "mydigital",
      "target": {
        "itemType": "page",
        "scope": "mydigital",
        "itemId": "checkout-page"
      }
    }
  ],
  "personalizations": [
    {
      "id": "checkout-banner",
      "strategy": "matching-first",
      "strategyOptions": {
        "fallback": "default-banner"
      },
      "contents": [
        {
          "id": "discount-banner",
          "content": "Use code SAVE10 for 10% off!",
          "filters": [
            {
              "condition": {
                "type": "profileSegmentCondition",
                "parameterValues": {
                  "segments": ["frequent-buyers"]
                }
              }
            }
          ]
        }
      ]
    }
  ],
  "requiredProfileProperties": ["*"],
  "requireSegments": true,
  "requireScores": true
}'
```

Processing Steps:

1. **Tenant Resolution**: Resolved from X-Unomi-Api-Key header

2. **Profile Loading**: Loaded existing profile existing-profile-uuid from cookie
3. **Session Loading**: Loaded existing session example-session
4. **Event Processing**:
    - view event validated and processed
    - Rules executed (may update profile, segments, scores)
    - Profile updated if rules triggered changes
5. **Personalization**:
    - Evaluated using updated profile state
    - Selected content based on segment membership
6. **Response Building**: Constructed with all requested data
7. **Cookie Setting**: Cookie refreshed in response header
8. **Persistence**: Profile and session saved if updated

Response:

```
{
  "profileId": "existing-profile-uuid",
  "sessionId": "example-session",
  "processedEvents": 1,
  "profileProperties": {
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com",
    "pageViewCount": 15
  },
  "profileSegments": ["frequent-buyers", "premium-customers"],
  "profileScores": {
    "loyalty": 85,
    "engagement": 92
  },
  "personalizations": {
    "checkout-banner": "discount-banner"
  },
  "personalizationResults": {
    "checkout-banner": {
      "selectedContentId": "discount-banner",
      "strategy": "matching-first"
    }
  }
}
```

Response Headers:

```
Set-Cookie: context-profile-id=existing-profile-uuid; Path=/; Max-Age=31536000; SameSite=Lax
```

This example demonstrates how a single request can: * Load an existing profile and session * Process events that trigger rules * Update the profile based on rule execution * Resolve personalization using the updated profile state * Return comprehensive profile data * Maintain tracking via cookies

18.7.5. Summary

A single request to /cxs/context.json or /cxs/context.js performs the following operations in sequence:

1. **Tenant Resolution** (mandatory in 4.0+): Identifies which tenant's data to use
2. **Profile/Session Management**: Creates or loads profiles and sessions automatically
3. **Event Processing**: Processes events and executes matching rules (synchronously)
4. **Content Filtering**: Evaluates filter conditions (if filters are provided)
5. **Personalization**: Resolves personalization requests (if personalizations are provided)
6. **Response Building**: Constructs the response with requested data (properties, segments, scores, etc.)
7. **Persistence**: Saves any changes to the database (profile and session)
8. **Cookie Setting**: Sets the profile ID cookie for future requests

This integrated approach ensures that: * Profiles and sessions are always available, even for first-time visitors * Events immediately trigger rules, updating profiles in real-time * Personalization uses the most up-to-date profile state * The browser automatically tracks the visitor across requests via cookies

18.7.6. Configuration

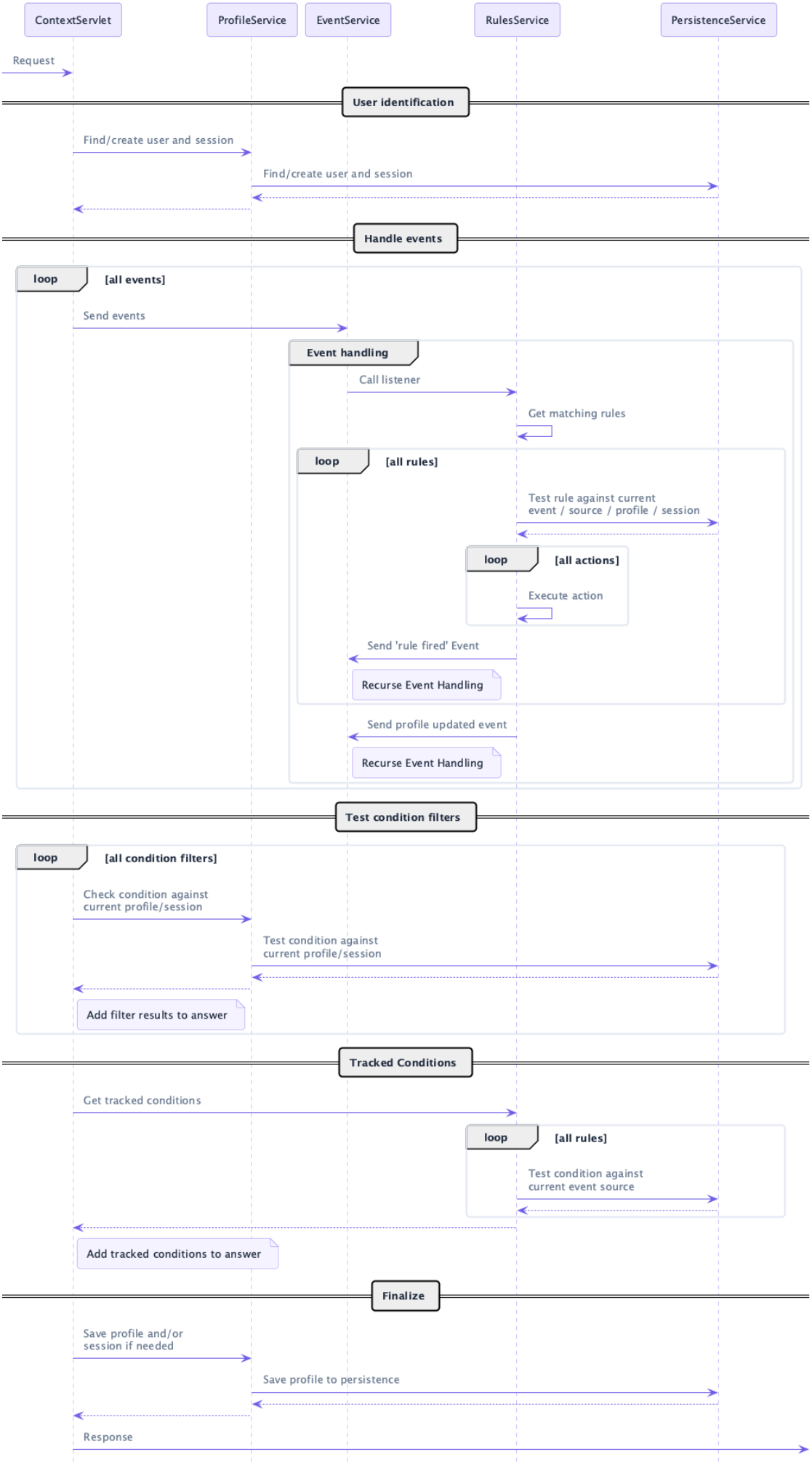
The following configuration properties control profile tracking behavior:

- org.apache.unomi.profile.cookie.name - Cookie name (default: context-profile-id)
- org.apache.unomi.profile.cookie.maxAgeInSeconds - Cookie expiration time (default: 31536000 = 1 year)
- org.apache.unomi.profile.cookie.httpOnly - Whether cookie is HTTP-only (default: true; only false turns it off, any other value keeps it on and is logged as a warning). When true, browser JavaScript cannot read the cookie; the browser still sends it on requests. Prefer reading profileId from the context JSON response.
- org.apache.unomi.profile.cookie.domain - Cookie domain (optional)
- org.apache.unomi.profile.cookie.sameSite - Lax (default), Strict or None; None needs Secure, and a non-empty server list or tenant lists on (the default)
- org.apache.unomi.public.cors.allowedOrigins - Website origins, other than Unomi's own, that may use the cookie (default: none). See Websites on other origins
- org.apache.unomi.public.cors.tenantOrigins - Whether each tenant may list its own websites in its publicCorsAllowedOrigins property (default: true). See Websites listed by each tenant
- org.apache.unomi.public.cors.matchReferer - Recognise a listed website that embeds context.js with a <script> tag by its Referer (default: false). See Embedding context.js with a script tag

These can be configured in etc/custom.system.properties / org.apache.unomi.web.cfg.

18.8. Context Request Flow

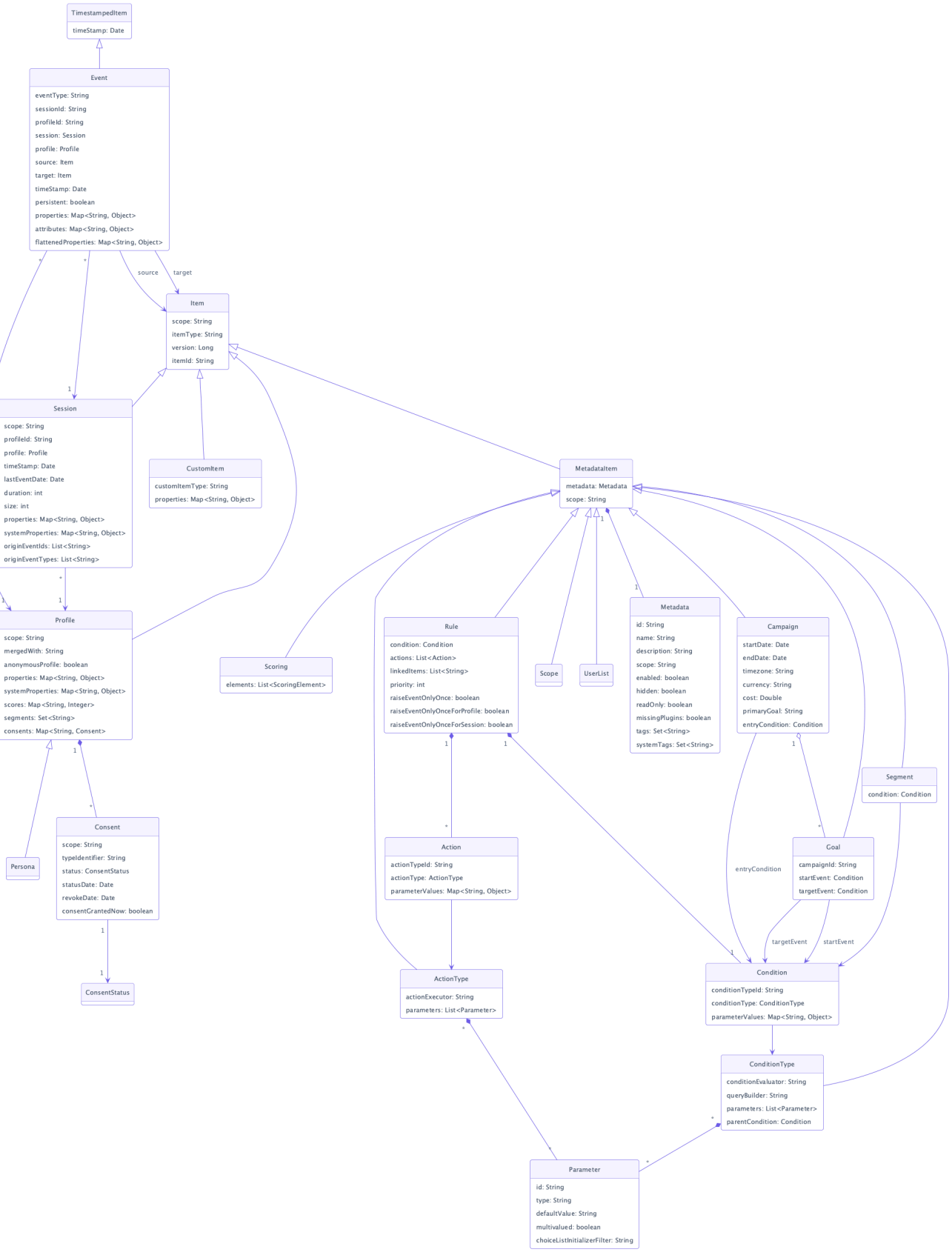
Here is an overview of how Unomi processes incoming requests to the ContextServlet.


ContextServlet
ProfileService
EventService
RulesService
PersistenceService
Request
User identification
Find/create user and session
Find/create user and session
Handle events
loop
[all events]
Send events
Event handling
Call listener
Get matching rules
loop
[all rules]
Test rule against current
event / source / profile / session
loop
[all actions]
Execute action
Send 'rule fired' Event
Recurse Event Handling
Send profile updated event
Recurse Event Handling
Test condition filters
loop
[all condition filters]
Check condition against
current profile/session
Test condition against
current profile/session
Add filter results to answer
Tracked Conditions
Get tracked conditions
loop
[all rules]
Test condition against
current event source
Add tracked conditions to answer
Finalize
Save profile and/or
session if needed
Save profile to persistence
Response

18.9. Data Model Overview

Apache Unomi gathers information about users actions, information that is processed and stored by Unomi services. The collected information can then be used to personalize content, derive insights on user behavior, categorize the user profiles into segments along user-definable dimensions or acted upon by algorithms.

The following data model only contains the classes and properties directly related to the most important objects of Apache Unomi. There are other classes that are less central to the functionality but all the major ones are represented in the diagram below:



We will detail many of these classes in the document below.

18.10. Scope

Scopes are objects which simply contains unique strings that are used to "classify" objects. For example, when using scopes with a web content management system, a scope could be associated with a site

identifier or even a host name.

In events, scopes are used to validate event. Events with scope which are unknown by the system will be considered as invalid

> Unomi defines a built-in scope (called systemscope) that clients can use to share data across scopes.

18.10.1. Example

In the following example, the scope uses the unique identifier of a web site called “digitall”.

```
{
  ... other fields of an object type ...
  “scope”: “digitall”
  ... other fields of an object type ...
}
```

18.11. Item

Unomi structures the information it collects using the concept of Item which provides the base information (an identifier and a type) the context server needs to process and store the data. Items are persisted according to their type (structure) and identifier (identity). This base structure can be extended, if needed, using properties in the form of key-value pairs.

These properties are further defined by the Item’s type definition which explicits the Item’s structure and semantics. By defining new types, users specify which properties (including the type of values they accept) are available to items of that specific type.

18.12. Tenant (Unomi 4.0+)

A **tenant** is the top-level isolation boundary in Unomi 4.0. Each tenant owns its own profiles, events, segments, rules, property types, and JSON schemas. Tenant metadata is stored as an Item with itemType tenant.

Most persisted items carry a tenantId (or inherit tenant scope through the execution context). Client requests must establish tenant context through:

- X-Unomi-Api-Key (public key resolves the tenant), or
- Basic authentication tenantId:privateApiKey, or
- JAAS administrator credentials with X-Unomi-Tenant-Id.

See Multi-tenancy for management and authentication.

Unomi defines default value types: date, email, integer and string, all pretty self-explanatory. While you

can think of these value types as "primitive" types, it is possible to extend Unomi by providing additional value types.

Additionally, most items are also associated to a scope, which is a concept that Unomi uses to group together related items. A given scope is represented in Unomi by a simple string identifier and usually represents an application or set of applications from which Unomi gathers data, depending on the desired analysis granularity. In the context of web sites, a scope could, for example, represent a site or family of related sites being analyzed. Scopes allow clients accessing the context server to filter data to only see relevant data.

Items are a generic object, that is common to many objects in the data model. It contains the following fields, that are inherited by other objects that inherit from it.

18.12.1. Structure definition

Inherits all the fields from: n/a

Field	Type	Description
itemId	String	This field contains a unique identifier (usually a UUID) that uniquely identifies the item in the whole system. It should be unique to a Unomi installation
itemType	String	A string containing the subtype of this item. Examples are : event, profile, session, ... any class that inherits from the Item class will have a unique and different itemType value.
scope	String (optional)	If present, this will contain a scope identifier. A scope is just a way to regroup objects notably for administrative purposes. For example, when integrating with a CMS a scope could be mapped to a website. The “system” scope value is reserved for values that are used internally by Apache Unomi

18.13. Metadata

The Metadata object is an object that contains additional information about an object. It is usually associated with an Item object (see MetadataItem below).

18.13.1. Structure definition

Inherits all the fields from: n/a

Field	Type	Description
id	String	This field contains a unique identifier (UUID) for the object the metadata object is attached to. It is usually a copy of the itemId field on an Item object.
name	String	A name for the associated object. Usually, this name will be displayed on the user interface
description	String (optional)	A description of the associated object. Will also usually be used in user interfaces
scope	String	The scope for the associated object.
tags	String array	A list of tags for the associated object, this list may be edited through a UI.
systemTags	String array	A (reserved) list of tags for the associated object. This is usually populated through JSON descriptors and is not meant to be modified by end users. These tags may include values such as “profileProperties” that help classify associated objects.
enabled	Boolean	Indicates whether the associated is enabled or not. For example, a rule may be disabled using this field.
missingPlugins	Boolean	This is used for associated objects that require plugins to be deployed to work. If the plugin is not deployed, this object will not perform its function. For example if a rule is registered but the condition or actions it needs are not installed, the rule will not be used.

Field	Type	Description
hidden	Boolean	Specifies whether the associated object should be visible in UIs or not
readOnly	Boolean	Specifies whether editing of the associated object should be allowed or not.

18.13.2. Example

This example of a Metadata object structure was taken from a List associated object. See the MetadataItem to understand how the two fit together.

```
{
      "id": "firstListId",
      "name": "First list",
      "description": "Description of the first list.",
      "scope": "digitall",
      "tags": [],
      "systemTags": [],
      "enabled": true,
      "missingPlugins": false,
      "hidden": false,
      "readOnly": false
}
```

18.14. MetadataItem

18.14.1. Structure definition

Inherits all the fields from: Item

Field	Type	Description
metadata	Metadata	This object contains just one field, of type Metadata as define just before this object type.

18.14.2. Example

The following example is actually the definition of a List object, which is simply a MetadataItem sub-type with no additional fields. We can see here the “itemId” and “itemType” fields that come from the Item parent class and the “metadata” field that contains the object structure coming from the Metadata object type.

```
{
  "itemId": "userListId",
  "itemType": "userList",
  "metadata": {
    "id": "userListId",
    "name": "First list",
    "description": "Description of the first list.",
    "scope": "digitall",
    "tags": [],
    "systemTags": [],
    "enabled": true,
    "missingPlugins": false,
    "hidden": false,
    "readOnly": false
  }
}
```


18.15. Event

Events represent something that is happening at a specific time (they are timestamped). They can be used to track visitor behavior, or even for back-channel system-to-system (as for example for a login) communication. Examples of events may include a click on a link on a web page, a login, a form submission, a page view or any other time-stamped action that needs to be tracked.

Events are persisted and immutable, and may be queried or aggregated to produce powerful reports.

Events can also be triggered as part of Unomi's internal processes for example when a rule is triggered.

18.15.1. Fields

Inherits all the fields from: Item

Field	Type	Description
eventType	String	Contains an identifier for the event type, which may be any value as Apache Unomi does not come with strict event type definitions and accepts custom events types. The system comes with built-in event types such as "view", "form", "login", "updateProperties" but additional event types may of course be used by developers integrating with Apache Unomi.
sessionId	String	The unique identifier of a Session object

Field	Type	Description
profileId	String	The unique identifier of a Profile object
timestamp	Date	The precise date at which the Event was received by Unomi. This date is in the ISO 8601 format.
scope	String	(Optional, event type specific) An identifier for a scope
persistent	Boolean	Defines if the event should be persisted or not (default: true)
source	Item	An Item that is the source of the event. For example a web site, an application name, a web page
target	Item	An Item that is the target of the event. For example a button, a link, a file or a page
properties	Map<String,Object>	Properties for the event. These will change depending on the event type.
flattenedProperties	Map<String,Object>	Properties that will be persisted as flattened. These will change depending on the event type.

18.15.2. Event types

Event types are completely open, and any new event type will be accepted by Apache Unomi.

Apache Unomi also comes with an extensive list of built-in event types you can find in the reference section of this manual.

18.16. Profile

By processing events, Unomi progressively builds a picture of who the user is and how they behave. This knowledge is embedded in Profile object. A profile is an Item with any number of properties and optional segments and scores. Unomi provides default properties to cover common data (name, last name, age, email, etc.) as well as default segments to categorize users. Unomi users are, however, free and even encouraged to create additional properties and segments to better suit their needs.

Contrary to other Unomi items, profiles are not part of a scope since we want to be able to track the associated user across applications. For this reason, data collected for a given profile in a specific scope is still available to any scoped item that accesses the profile information.

It is interesting to note that there is not necessarily a one to one mapping between users and profiles as

users can be captured across applications and different observation contexts. As identifying information might not be available in all contexts in which data is collected, resolving profiles to a single physical user can become complex because physical users are not observed directly. Rather, their portrait is progressively patched together and made clearer as Unomi captures more and more traces of their actions. Unomi will merge related profiles as soon as collected data permits positive association between distinct profiles, usually as a result of the user performing some identifying action in a context where the user hadn't already been positively identified.

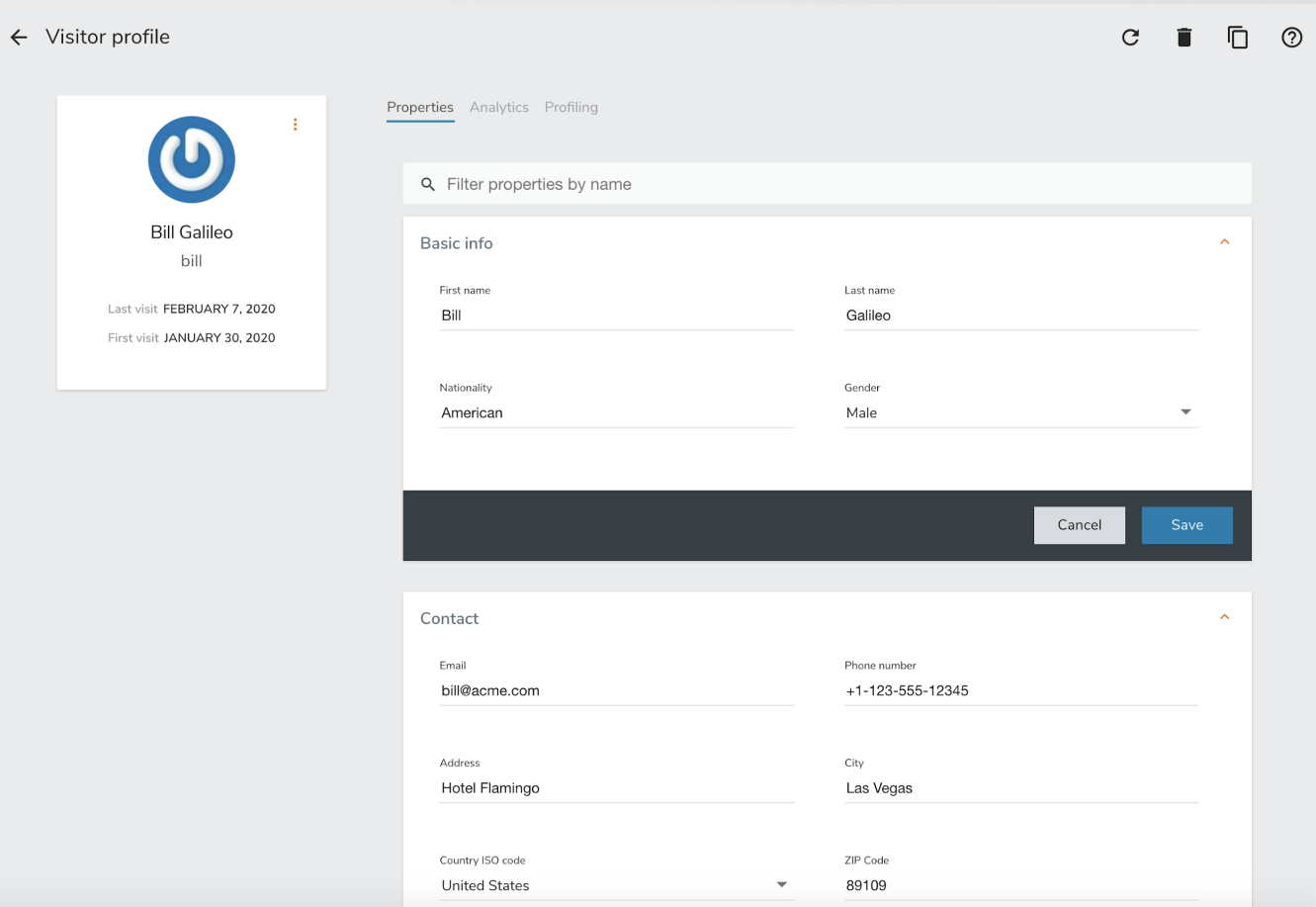
18.16.1. Structure definition

Inherits all the fields from: Item

Field name	Type	Description
properties	Map<String,Object>	All the (user-facing) properties for the profile
systemProperties	Map<String,Object>	Internal properties used to track things such as goals reached, merges with other profiles, lists the profile belongs to.
segments	String set	A set of Segment identifiers that profile is (currently) associated with
scores	Map<String,Integer>	A map of scores with the score identifier as the key and the score total value as the value.
@Deprecated mergedWith	String	If merged with another profile, the profile identifier to the master profile is stored here
consents	Map<String,Consent>	The consents for the profile, as a map with the consent identifier as a key and the Consent object type as a value.

18.16.2. Example

In the example below, a profile for a visitor called "Bill Galileo" is detailed. A lot of user properties (such as first name, last name, gender, job title and more) were copied over from the CMS upon initial login. The profile is also part of 4 segments (leads, contacts, gender_male, age_60_70) and has a lot of different scores as well. It is also part of a list (systemProperties.lists), and has granted two consents for receiving newsletters. It has also been engaged in some goals (systemProperties.goals.*StartReached) and completed some goals (systemProperties.goals.*TargetReached)




```
{
   "itemId": "f7d1f1b9-4415-4ff1-8fee-407b109364f7",
   "itemType": "profile",
   "properties": {
      "lastName": "Galileo",
      "preferredLanguage": "en",
      "nbOfVisits": 2,
      "totalNbOfVisits": 5,
      "gender": "male",
      "jobTitle": "Vice President",
      "lastVisit": "2020-01-31T08:41:22Z",
      "j:title": "mister",
      "j:about": "<p> Lorem Ipsum dolor sit amet,consectetur adipisicing elit, sed doeiusmod tempor incididunt ut laboreet dolore magna aliqua. Ut enim adminim veniam, quis nostrudexercitation ullamco laboris nisi utaliquip ex ea commodo consequat.Duis aute irure dolor inreprehenderit in coluptate velit essecillum dolore eu fugiat nulla pariatur.Excepteur sint occaecat cupidatatnon proident, sunt in culpa quiofficia deserunt mollit anim id estlaborum.</p> ",
      "firstName": "Bill",
      "pageViewCount": {
         "digitall": 19
      },
      "emailNotificationsDisabled": "true",
      "company": "Acme Space",
      "j:nodename": "bill",
      "j:publicProperties":
"j:about,j:firstName,j:function,j:gender,j:lastName,j:organization,j:picture,j:title",
      "firstVisit": "2020-01-30T21:18:12Z",
      "phoneNumber": "+1-123-555-12345",
      "countryName": "US",
```

```
    "city": "Las Vegas",
    "address": "Hotel Flamingo",
    "zipCode": "89109",
    "email": "bill@acme.com",
    "maritalStatus": "Married",
    "birthDate": "1959-08-12T23:00:00.000Z",
    "kids": 2,
    "age": 60,
    "income": 1000000,
    "facebookId": "billgalileo",
    "twitterId": "billgalileo",
    "linkedInId": "billgalileo",
    "leadAssignedTo": "Important Manager",
    "nationality": "American"
  },
  "systemProperties": {
    "mergeIdentifier": "bill",
    "lists": [
      "userListId"
    ],
    "goals": {
      "viewLanguagePageGoalTargetReached": "2020-02-10T19:30:31Z",
      "downloadGoalExampleTargetReached": "2020-02-10T15:22:41Z",
      "viewLandingPageGoalStartReached": "2020-02-10T19:30:27Z",
      "downloadGoalExampleStartReached": "2020-02-10T19:30:27Z",
      "optimizationTestGoalStartReached": "2020-02-10T19:30:27Z"
    }
  },
  "segments": [
    "leads",
    "age_60_70",
    "gender_male",
    "contacts"
  ],
  "scores": {
    "scoring_9": 10,
    "scoring_8": 0,
    "scoring_1": 10,
    "scoring_0": 10,
    "_s02s6220m": 0,
    "scoring_3": 10,
    "_27ir92oa2": 0,
    "scoring_2": 10,
    "scoring_5": 10,
    "scoring_4": 10,
    "scoring_7": 10,
    "scoring_6": 10,
    "_86igp9j1f": 1,
    "_ps8d573on": 0
  },
  "mergedWith": null,
  "consents": {
    "digitall/newsletter1": {
      "scope": "digitall",
      "typeIdentifier": "newsletter1",
      "status": "GRANTED",
      "statusDate": "2019-05-15T14:47:28Z",
```

```
        "revokeDate": "2021-05-14T14:47:28Z"
      },
      "digitall/newsletter2": {
        "scope": "digitall",
        "typeIdentifier": "newsletter2",
        "status": "GRANTED",
        "statusDate": "2019-05-15T14:47:28Z",
        "revokeDate": "2021-05-14T14:47:28Z"
      }
    }
}
```

18.17. Profile aliases

Profile aliases make it possible to reference profiles using multiple identifiers. The profile alias object basically contains a link between the alias ID and the profile ID. The itemId of a profile alias is the actual alias ID, which the profileID field contains the reference to the aliased profile.

18.17.1. Structure definition

Inherits all the fields from: Item

Field name	Type	Description
profileID	String	The identifier of the profile this aliases points to
creationTime	DateTime	The date and time of creation of the alias
modifiedTime	DateTime	The date and time of last modification of the alias

18.17.2. Example

In the following example we show an alias ID facebook_johndoe for the profile with ID f72242d2-3145-43b1-8be7-d1d47cf4ad0e

```
{
  "profileID": "f72242d2-3145-43b1-8be7-d1d47cf4ad0e",
  "itemId" : "facebook_johndoe",
  "creationTime" : "2022-09-16T19:23:51Z",
  "modifiedTime" : "2022-09-16T19:23:51Z"
}
```

18.18. Persona

A persona is a specialized version of a Profile object. It basically represents a "typical" profile and can be used notably to simulate personalized for a type of profiles. Usually personas are created from Profile

data and then edited to represent a specific marketing persona.

18.18.1. Structure definition

Inherits all the fields from: Profile

There are no fields specific to a Persona.

18.18.2. Example

In the following example a Persona represents a visitor from Europe, that can be used to match by location.

```
{
  "itemId": "europeanVisitor",
  "itemType": "persona",
  "properties": {
    "description": "Represents a visitor browsing from Europe",
    "firstName": "European",
    "lastName": "Visitor",
    "continent": "Europe"
  },
  "systemProperties": {},
  "segments": [],
  "scores": null,
  "consents": {}
}
```

18.19. Consent

A consent represents a single instance of a consent granted/refused or revoked by a profile. A profile will contain multiple instances of consent identified by unique identifiers.

18.19.1. Structure definition

Inherits all the fields from: n/a

Field name	Type	Description
scope	String	The scope this consent is associated with. In the case of a website this might be the unique identifier for the site.

Field name	Type	Description
typeIdentifier	String	This is a unique consent type identifier, basically a unique name for the consent. Example of such types might include: “newsletter”, “personalization”, “tracking”.
status	GRANTED / DENIED / REVOKED	The type of status for this consent
statusDate	Date	The date (in ISO 8601 format) at which the current status was set
revokeDate	Date	The date (in ISO 8106 format) at which time the current status is automatically revoked.

18.19.2. Example

In this example, the consent called “newsletter” was given on the “digitall” website.

```
{
        "scope": "digitall",
        "typeIdentifier": "newsletter",
        "status": "GRANTED",
        "statusDate": "2019-05-15T14:47:28Z",
        "revokeDate": "2021-05-14T14:47:28Z"
}
```

18.20. Session

A session represents a period of time during which a visitor/profile has been active. It makes it possible to gather data and then use it for reporting and further analysis by regrouping all the events that occurred during the session.

18.20.1. Structure definition

Inherits all the fields from: Item

Field name	Type	Description
properties	Map<String,Object>	All the properties for the session. These contain information such as the browser, operating system and device used, as well as information about the location of the visitor.
systemProperties	Map<String,Object>	Not used (empty)

Field name	Type	Description
profileId	String	The identifier of the profile that generated the session
profile	Profile	A copy of the profile associated with the session
size	Integer	The number of view event types received during this session
duration	Integer	The duration of the session in milliseconds
lastEventDate	Date	The date of the last event that occurred in the session, in ISO 8601 format.

18.20.2. Example

In this example the session contains a copy of the profile of the visitor. It is a visitor that has previously authentified in a CMS and who'se information was copied at the time of login from the CMS user account to the profile. You can also notice that the session contains the information coming from the browser's user agent which contains the browser type, version as well as the operating system used. The visitor's location is also resolve based on the IP address that was used to send events.


```
{
   "itemId": "4dcb5b74-6923-45ae-861a-6399ef88a209",
   "itemType": "session",
   "scope": "digitall",
   "profileId": "f7d1f1b9-4415-4ff1-8fee-407b109364f7",
   "profile": {
      "itemId": "f7d1f1b9-4415-4ff1-8fee-407b109364f7",
      "itemType": "profile",
      "properties": {
         "preferredLanguage": "en",
         "nbOfVisits": 2,
         "totalNbOfVisits": 5,
         "gender": "male",
         "jobTitle": "Vice President",
         "lastVisit": "2020-01-31T08:41:22Z",
         "j:title": "mister",
         "j:about": "<p> Lorem Ipsum dolor sit amet,consectetur adipisicing elit, sed doeiusmod tempor
incididunt ut laboreet dolore magna aliqua. Ut enim adminim veniam, quis nostrudexercitation
ullamco laboris nisi utaliquip ex ea commodo consequat.Duis aute irure dolor inreprehenderit in
coluptate velit essecillum dolore eu fugiat nulla pariatur.Excepteur sint occaecat cupidatatnon
proident, sunt in culpa quiofficia deserunt mollit anim id estlaborum.</p> ",
         "pageViewCount": {
            "digitall": 19
         },
         "emailNotificationsDisabled": "true",
         "company": "Acme Space",
         "j:publicProperties":
"j:about,j:firstName,j:function,j:gender,j:lastName,j:organization,j:picture,j:title",
```

```
        "firstVisit": "2020-01-30T21:18:12Z",
        "countryName": "US",
        "city": "Las Vegas",
        "zipCode": "89109",
        "maritalStatus": "Married",
        "birthDate": "1959-08-12T23:00:00.000Z",
        "kids": 25,
        "age": 60,
        "income": 1000000,
        "leadAssignedTo": "Important Manager"
      },
      "systemProperties": {
        "mergeIdentifier": "bill",
        "lists": [
          "_xb2bcm4wl"
        ]
      },
      "segments": [
        "leads",
        "age_60_70",
        "gender_male",
        "contacts"
      ],
      "scores": {
        "scoring_9": 10,
        "scoring_8": 0,
        "scoring_1": 10,
        "scoring_0": 10,
        "_s02s6220m": 0,
        "scoring_3": 10,
        "_27ir92oa2": 0,
        "scoring_2": 10,
        "scoring_5": 10,
        "scoring_4": 10,
        "scoring_7": 10,
        "scoring_6": 10,
        "_86igp9j1f": 1,
        "_ps8d573on": 0
      },
      "mergedWith": null,
      "consents": {}
    },
    "properties": {
      "sessionCity": "Geneva",
      "operatingSystemFamily": "Desktop",
      "userAgentNameAndVersion": "Firefox@@72.0",
      "countryAndCity": "Switzerland@@Geneva@@2660645@@6458783",
      "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:72.0) Gecko/20100101
Firefox/72.0",
      "userAgentName": "Firefox",
      "sessionCountryCode": "CH",
      "deviceName": null,
      "sessionCountryName": "Switzerland",
      "referringURL": "null",
      "deviceCategory": "Apple Macintosh",
      "pageReferringURL": "http://localhost:8080/sites/digitall/home/corporate-responsibility.html",
      "userAgentVersion": "72.0",
```

```
        "sessionAdminSubDiv2": 6458783,
        "sessionAdminSubDiv1": 2660645,
        "location": {
          "lon": 6.1282508,
          "lat": 46.1884341
        },
        "sessionIsp": "Cablecom",
        "operatingSystemName": "Mac OS X",
        "deviceBrand": "Apple"
    },
    "systemProperties": {},
    "timeStamp": "2020-01-31T08:41:22Z",
    "lastEventDate": "2020-01-31T08:53:32Z",
    "size": 19,
    "duration": 730317
}
```


18.21. Segment

Segments are used to group profiles together, and are based on conditions that are executed on profiles to determine if they are part of a segment or not.

This also means that a profile may enter or leave a segment based on changes in their properties, making segments a highly dynamic concept.

18.21.1. Structure definition

Inherits all the fields from: MetadataItem

Field name	Type	Description
condition	Condition	The root condition for the segment. Conditions may be composed by using built-in condition types such as booleanCondition that can accept sub-conditions.

18.21.2. Example

```
{
  "itemId": "age_20_30",
  "itemType": "segment",
  "condition": {
    "parameterValues": {
      "subConditions": [
        {
          "parameterValues": {
            "propertyName": "properties.age",
            "comparisonOperator": "greaterThanOrEqualTo",
            "propertyValueInteger": 20
          },
          "type": "profilePropertyCondition"
        },
        {
          "parameterValues": {
            "propertyName": "properties.age",
            "comparisonOperator": "lessThan",
            "propertyValueInteger": 30
          },
          "type": "profilePropertyCondition"
        }
      ],
      "operator": "and"
    },
    "type": "booleanCondition"
  },
  "metadata": {
    "id": "age_20_30",
    "name": "age_20_30",
    "description": null,
    "scope": "digitall",
    "tags": [],
    "enabled": true,
    "missingPlugins": false,
    "hidden": false,
    "readOnly": false
  }
}
```


Here is an example of a simple segment definition registered using the REST API:

```
curl -X POST http://localhost:8181/cxs/segments \
--user "TENANT_ID:PRIVATE_KEY" \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
 "metadata": {
  "id": "leads",
  "name": "Leads",
  "scope": "systemscope",
  "description": "You can customize the list below by editing the leads segment.",
  "readOnly":true
 },
 "condition": {
  "type": "booleanCondition",
  "parameterValues": {
   "operator" : "and",
   "subConditions": [
    {
     "type": "profilePropertyCondition",
     "parameterValues": {
      "propertyName": "properties.leadAssignedTo",
      "comparisonOperator": "exists"
     }
    }
   ]
  }
 }
}
EOF
```

For more details on the conditions and how they are structured using conditions, see the next section.

18.22. Condition

Conditions are a very useful notion inside of Apache Unomi, as they are used as the basis for multiple other objects. Conditions may be used as parts of:

- Segments
- Rules
- Queries
- Campaigns
- Goals
- Profile filters (using to search for profiles)

The result of a condition is always a boolean value of true or false.

Apache Unomi provides quite a lot of built-in condition types, including boolean types that make it possible to compose conditions using operators such as and, or or not. Composition is an essential element of building more complex conditions.

For tutorials and examples, see Conditions guide. For the full catalog, see Built-in condition types.

18.22.1. Structure definition

Inherits all the fields from: n/a

Field name	Type	Description
conditionTypeId	String	A condition type identifier is a string that contains a unique identifier for a condition type. Example condition types may include booleanCondition, eventTypeCondition, eventPropertyCondition, and so on. Plugins may implement new condition types that may implement any logic that may be needed.
parameterValues	Map<String,Object>	The parameter values are simply key-value paris that may be used to configure the condition. In the case of a booleanCondition for example one of the parameter values will be an operator that will contain values such as and or or and a second parameter value called subConditions that contains a list of conditions to evaluate with that operator.

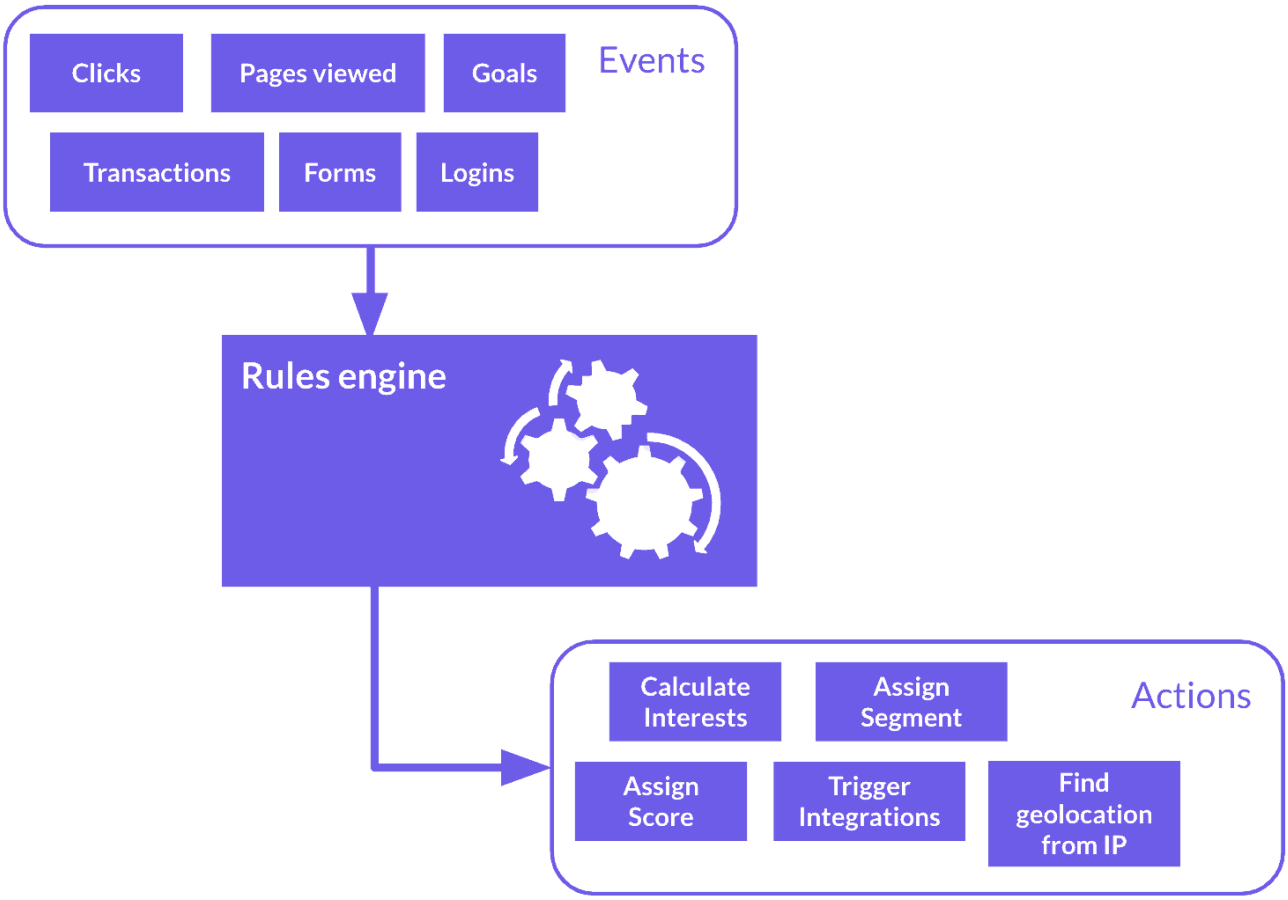
18.22.2. Example

Here is an example of a complex condition:

```
{
 "condition": {
  "type": "booleanCondition",
  "parameterValues": {
   "operator":"or",
   "subConditions":[
    {
     "type": "eventTypeCondition",
     "parameterValues": {
      "eventTypeId": "sessionCreated"
     }
    },
    {
     "type": "eventTypeCondition",
     "parameterValues": {
      "eventTypeId": "sessionReassigned"
     }
    }
   ]
  }
 }
}
```

As we can see in the above example we use the boolean or condition to check if the event type is of type sessionCreated or sessionReassigned.

18.23. Rule



Apache Unomi has a built-in rule engine that is one of the most important components of its architecture. Every time an event is received by the server, it is evaluated against all the rules and the ones matching the incoming event will be executed. You can think of a rule as a structure that looks like this:

```
when
   conditions
then
   actions
```

Basically when a rule is evaluated, all the conditions in the when part are evaluated and if the result matches (meaning it evaluates to true) then the actions will be executed in sequence.

The real power of Apache Unomi comes from the fact that conditions and actions are fully pluggable and that plugins may implement new conditions and/or actions to perform any task. You can imagine conditions checking incoming event data against third-party systems or even against authentication systems, and actions actually pulling or pushing data to third-party systems.

For example the Salesforce CRM connector is simply a set of actions that pull and push data into the CRM. It is then just a matter of setting up the proper rules with the proper conditions to determine when and how the data will be pulled or pushed into the third-party system.

18.23.1. Structure definition

Inherits all the fields from: MetadataItem

Field name	Type	Description
condition	Condition	The root condition for the rule. Conditions may be composed by using built-in condition types such as booleanCondition that can accept sub-conditions.
action	Action array	A list of Action object that will be executed if the condition is true.
linkedItems	String array	A list of references to objects that may have generated this rule. Goals and segments dynamically generate rules to react to incoming events. It is not recommend to manipulate rules that have linkedItems as it may break functionality.
raiseEventOnlyOnce	Boolean	If true, the rule will only be executed once for a given event.
raiseEventOnlyOnceForProfile	Boolean	If true, the rule will only be executed once for a given profile and a matching event. Warning: this functionality has a performance impact since it looks up past events.
raiseEventOnlyOnceForSession	Boolean	If true, the rule will only be executed once for a given session and a matching event. Warning: this functionality has a performance impact since it looks up past events.
priority	Integer	The priority for the rule. The lower the priority value the higher the effective priority (they are sorted by ascending order of priority)

18.23.2. Example

In this example we can see the default updateProperties built-in rule that matches the updateProperties event and executes the built-in updatePropertiesAction

```
{
  "itemId": "updateProperties",
  "itemType": "rule",
  "condition": {
    "parameterValues": {},
    "type": "updatePropertiesEventCondition"
  },
  "actions": [
    {
      "parameterValues": {},
      "type": "updatePropertiesAction"
    }
  ],
  "linkedItems": null,
  "raiseEventOnlyOnceForProfile": false,
  "raiseEventOnlyOnceForSession": false,
  "priority": 0,
  "metadata": {
    "id": "updateProperties",
    "name": "Update profile/persona properties",
    "description": "Update profile/persona properties",
    "scope": "systemscope",
    "tags": [],
    "systemTags": [],
    "enabled": true,
    "missingPlugins": false,
    "hidden": false,
    "readOnly": true
  }
}
```


18.24. Action

Actions are executed by rules in a sequence, and an action is only executed once the previous action has finished executing. If an action generates an exception, it will be logged and the execution sequence will continue unless in the case of a Runtime exception (such as a NullPointerException).

Action use Action types that are implemented as Java classes, and as such may perform any kind of tasks that may include calling web hooks, setting profile properties, extracting data from the incoming request (such as resolving location from an IP address), or even pulling and/or pushing data to third-party systems such as a CRM server.

Apache Unomi also comes with built-in action types. You may find the list of built-in action types in the Built-in action types section.

18.24.1. Structure definition

Inherits all the fields from: n/a

Field name	Type	Description
actionTypeId	String	An action type identifier is a string that contains a unique identifier for a action type.
parameterValues	Map<String,Object>	The parameter values are simply key-value paris that may be used to configure the action.

18.24.2. Example

In this example of an action, taking from the form-mapping-example.json rule, the setPropertyAction action is used to set the properties.firstName profile property to a value read from the event properties called properties.firstName. The setPropertyStrategy is a parameter specific to this action that allows to define if existing values should be overridden or not.

```
{
 "type": "setPropertyAction",
 "parameterValues": {
  "setPropertyName": "properties(firstName)",
  "setPropertyValue": "eventProperty::properties(firstName)",
  "setPropertyStrategy": "alwaysSet"
 }
}
```

18.25. List

Lists are a “manual” way to organize profiles, whereas Segments are a dynamic way to regroup them. List objects actually only define the list in terms of name, description and other metadata but the list of members is actually not represented in the object. The profiles contain references to the lists in their “systemProperties.lists” property. This property is an array of list identifiers so in order to retrieve all the list names for a given profile, a lookup of List objects is required using the identifiers.

18.25.1. Structure definition

Inherits all the fields from: MetadataItem

Field name	Description
	No additional fields are present in this object type

18.25.2. Example

Here’s an example of a list called “First list”, along with its description, its scope, tags, etc.. . As a List object is basically a MetadataItem sub-class it simply has all the fields defined in that parent class. Note that the List does not contain Profiles, it is Profiles that reference the Lists, not the reverse.

```
{
  "itemId": "userListId",
  "itemType": "userList",
  "metadata": {
    "id": "userListId",
    "name": "First list",
    "description": "Description of the first list.",
    "scope": "digitall",
    "tags": [],
    "systemTags": [],
    "enabled": true,
    "missingPlugins": false,
    "hidden": false,
    "readOnly": false
  }
}
```


18.26. Goal

A goal can be defined with two conditions: a start event condition and an target event condition. Basically the goal will be “active” when its start event condition is satisfied, and “reached” when the target event condition is true. Goals may also (optionally) be associated with Campaigns. Once a goal is “reached”, a “goal” event triggered and the profile that is currently interacting with the system will see its system properties updated to indicate which goal has been reached.

18.26.1. Structure definition

Inherits all the fields from: MetadataItem

Field name	Type	Description
startEvent	Condition	The condition that will be used to determine if this goal was activated by the current profile
targetEvent	Condition	The condition that will be used to determine if the current profile has reached the goal.
campaignId	String	If this goal was setup as part of a Campaign, the unique identifier for the campaign is stored in this field.

18.26.2. Example

In the following example, a goal called “downloadGoalExample” is started when a new session is created (we use the “sessionCreatedEventCondition” for that) and is reached when a profile downloads a file called “ACME_WP.pdf” (that’s what the “downloadEventCondition” means).

```
{
  "itemId": "downloadGoalExample",
  "itemType": "goal",
  "startEvent": {
    "parameterValues": {},
    "type": "sessionCreatedEventCondition"
  },
  "targetEvent": {
    "parameterValues": {
      "filePath": "/sites/digitall/files/PDF/Publications/ACME_WP.pdf"
    },
    "type": "downloadEventCondition"
  },
  "campaignId": "firstCampaignExample",
  "metadata": {
    "id": "downloadGoalExample",
    "name": "downloadGoalExample",
    "description": null,
    "scope": "digitall",
    "enabled": true,
    "missingPlugins": false,
    "hidden": false,
    "readOnly": false,
    "systemTags": [
      "goal",
      "downloadGoal"
    ]
  }
}
```


18.27. Campaign

A Campaign object represents a digital marketing campaign, along with conditions to enter the campaign and a specific duration, target and costs.

18.27.1. Structure definition

Inherits all the fields from: MetadataItem

Field name	Type	Description
startDate	Date	The start date of the Campaign (in ISO 8601 format)
endDate	Date	The end date of the Campaign (in ISO 8601 format)
entryCondition	Condition	The condition that must be satisfied for a profile to become a participant in the campaign
cost	Double	An indicative cost for the campaign

Field name	Type	Description
currency	String	The currency code (3-letter) for the cost of the campaign
primaryGoal	String	A unique identifier of the primary Goal for the campaign.
timezone	String	The timezone of the campaign identified by the TZ database name (see https://en.wikipedia.org/wiki/List_of_tz_database_time_zones)

18.27.2. Example

In the following example a campaign that starts January 1st 31, 2020 at 8:38am and finished on February 29th, 2020 at the same time has the following entry condition: the session duration must be less or equal to 3000 milliseconds (3 seconds) and the profile has viewed the “about” page on the “digitall” website. The cost of the campaign is USD 1’000’000 and the timezone is Europe/Zurich. The primary goal for the campaign is the goal we should have as an example in the Goal section.

```
{
  "itemId": "firstCampaignExample",
  "itemType": "campaign",
  "startDate": "2020-01-31T08:38:00Z",
  "endDate": "2020-02-29T08:38:00Z",
  "entryCondition": {
    "parameterValues": {
      "subConditions": [
        {
          "parameterValues": {
            "propertyName": "duration",
            "comparisonOperator": "lessThanOrEqualTo",
            "propertyValueInteger": 3000
          },
          "type": "sessionPropertyCondition"
        },
        {
          "parameterValues": {
            "pagePath": "/sites/digitall/home/about"
          },
          "type": "pageViewEventCondition"
        }
      ],
      "operator": "and"
    },
    "type": "booleanCondition"
  },
  "cost": 1000000,
  "currency": "USD",
  "primaryGoal": "downloadGoalExample",
  "timezone": "Europe/Zurich",
  "metadata": {
    "id": "firstCampaignExample",
    "name": "firstCampaign",
    "description": "Example of a campaign",
    "scope": "digitall",
    "tags": [],
    "systemTags": [
      "landing",
      "campaign"
    ],
    "enabled": true,
    "missingPlugins": false,
    "hidden": false,
    "readOnly": false
  }
}
```


18.28. Scoring plan

Scoring plans make it possible to define scores that will be tracked for profiles and use conditions to increment a score when the conditions are met. This makes it possible to then use threshold conditions on profiles when they reach a certain score.

18.28.1. Structure definition

Inherits all the fields from: MetadataItem

Field name	Type	Description
elements	ScoringElement array	A ScoringElement is composed of: a Condition and a score value to increment. Each element defines a separate condition (tree) that will increment the defined score for this scoring plan, making it possible to have completely different conditions to augment a score.

18.28.2. Example

In this example a scoring plan contains a single element that will increment a score with an increment one 1 once the profile has viewed at least 3 pages (using the “hasSeenNPagesCondition” condition).


```
{
  "itemId": "viewMoreThan3PagesId",
  "itemType": "scoring",
  "elements": [
    {
      "condition": {
        "parameterValues": {
          "value": 3,
          "scope": "digitall",
          "comparisonOperator": "greaterThanOrEqualTo"
        },
        "type": "hasSeenNPagesCondition"
      },
      "value": 1
    }
  ],
  "metadata": {
    "id": "viewMoreThan3PagesId",
    "name": "Viewed more than 3 pages",
    "description": null,
    "scope": "digitall",
    "tags": [],
    "systemTags": [
      "st:behavioral"
    ],
    "enabled": true,
    "missingPlugins": false,
    "hidden": false,
    "readOnly": false
  }
}
```

18.29. Property Types

Property types define the structure and metadata for properties that can be used in profiles and sessions within Apache Unomi. They specify the data type, display hints (ranges, constraints), default values, and other metadata that help inform UIs and enable rich querying capabilities.



Property types are primarily designed for **UI purposes** - they inform user interfaces about what properties should be editable, how they should be displayed, and what constraints apply. Property types are **dynamically created and updated** at runtime and do not enforce strict validation on the server side. For actual data validation, Apache Unomi uses JSON Schemas, which are used exclusively for validating events sent through public endpoints.

18.29.1. Quick Reference: Property Types vs JSON Schemas

Aspect	Property Types	JSON Schemas
Primary Purpose	UI metadata and display hints	Server-side validation
Used For Objects	Profiles (editing), Sessions (editing)	Events (validation only)
Storage	Stored as PropertyType items, accessible via REST API	Stored in Elasticsearch/OpenSearch
Dynamic Creation	Yes, can be created/updated at runtime via REST API	Yes, can be added via API but primarily for events
Validation	No server-side validation (informational only)	Yes, enforces strict validation
Event Support	Not applicable - property types are for profiles and sessions only	Required for all events sent through public endpoints
Accessibility	Accessible via /cxs/profiles/properties/targets/{target} REST API	Accessible via JSON schema API

18.29.2. Overview

Property types serve as **informational schemas** for properties, primarily used by UIs to:

- The data type (string, integer, date, boolean, etc.)
- Whether the property can have multiple values
- Default values
- Validation constraints (ranges, date ranges, IP ranges) - primarily for UI hints
- Which object types the property applies to (profiles or sessions)
- How properties should be merged when profiles are combined

- Whether the property is protected (read-only)
- Display metadata (names, descriptions, tags, ranks) for UI organization

18.29.3. Property Type Targets

Property types are organized by their target, which indicates which type of object the property applies to:

- profiles or profile - Properties for Profile objects (primary use case: editing profile data in UIs)
- sessions or session - Properties for Session objects (primary use case: editing session data in UIs)

The target is typically determined automatically from the file path when property types are loaded from JSON files:

- Files in /META-INF/cxs/properties/profiles/... □ target = "profiles"
- Files in /META-INF/cxs/properties/sessions/... □ target = "sessions"



Property types are only used for profiles and sessions. Events are immutable records and should not be edited through property type-based UIs. Event validation is handled by JSON schemas, not property types.

18.29.4. Property Type Structure

A property type is defined using the following structure:

Structure Definition

Inherits all the fields from: MetadataItem

Field name	Type	Description
target	String	The type of object this property type applies to (e.g., "profiles", "sessions"). Determined automatically from file path if not explicitly set. The target is extracted from the directory name immediately following "properties" in the file path (e.g., META-INF/cxs/properties/profiles/... □ "profiles").

Field name	Type	Description
type / valueTypeId	String	The identifier of the value type constraining values for properties using this PropertyType (e.g., "string", "integer", "date", "boolean", "set").
defaultValue	String	The default value that properties using this PropertyType will use if no value is specified explicitly.
multivalued	Boolean	Whether properties using this property type are multi-valued (a vector of values as opposed to a scalar value).
protected / protekted	Boolean	Whether properties with this type are marked as protected. Protected properties can be displayed but their value cannot be changed. Note: In JSON files, use "protected" (the Java field name is protekted).
rank	Double	The rank of this PropertyType for ordering purposes. Used to control the display order of properties in UIs.
mergeStrategy	String	The identifier of the PropertyMergeStrategyType to be used when profiles with properties using this PropertyType are being merged.
dateRanges	List<DateRange>	Optional list of date ranges that can be used for UI categorization and hints. Note: These are informational for UIs, not enforced server-side validation.
numericRanges	List<NumericRange>	Optional list of numeric ranges that can be used for UI categorization and hints (e.g., age groups). Note: These are informational for UIs, not enforced server-side validation.

Field name	Type	Description
ipRanges	List<IpRange>	Optional list of IP ranges that can be used for UI categorization and hints. Note: These are informational for UIs, not enforced server-side validation.
automaticMappingsFrom	Set<String>	Set of property names from which properties of this type would be automatically initialized from (legacy feature).
childPropertyTypes	Set<PropertyType>	For nested/complex properties, this contains the child property types that define the structure.

Metadata Fields

Property types inherit from MetadataItem, which provides:

Field name	Type	Description
id	String	Unique identifier for the property type (usually matches the property name).
name	String	Display name for the property type.
description	String	Description of what the property represents.
scope	String	The scope for the property type.
tags	Set<String>	User-editable tags for categorization.
systemTags	Set<String>	System-reserved tags for categorization (e.g., "profileProperties", "sessionProperties", "personalIdentifierProperties").
enabled	Boolean	Whether the property type is enabled.
hidden	Boolean	Whether the property type should be hidden in UIs.
readOnly	Boolean	Whether the property type definition can be edited.

18.29.5. Value Types

Property types use value types to define the primitive data type. Common value types include:

- string - Text values
- integer - Whole numbers
- long - Large integers
- float - Floating point numbers
- date - Date and time values
- boolean - True/false values
- set - Collections of values (for multi-valued or nested properties)
- email - Email addresses
- geoPoint - Geographic coordinates

18.29.6. Examples

Profile Property Type Example

This example shows a profile property type for storing a person's age with numeric ranges:

```
{
 "metadata": {
  "id": "age",
  "name": "Age",
  "systemTags": [
   "properties",
   "profileProperties",
   "personalProfileProperties"
  ]
 },
 "type": "integer",
 "defaultValue": "",
 "numericRanges": [
  {"key": "*_10", "to": 10},
  {"key": "10_20", "from": 10, "to": 20},
  {"key": "20_30", "from": 20, "to": 30},
  {"key": "30_40", "from": 30, "to": 40},
  {"key": "40_50", "from": 40, "to": 50},
  {"key": "50_*", "from": 50}
 ],
 "automaticMappingsFrom": [],
 "rank": "501.0"
}
```

Session Property Type Example

This example shows a session property type for storing geographic information:

```
{
 "metadata": {
  "id": "sessionCity",
  "name": "City",
  "systemTags": [
   "properties",
   "sessionProperties",
   "geographicSessionProperties"
  ]
 },
 "type": "string",
 "defaultValue": "",
 "automaticMappingsFrom": [],
 "rank": "3.0"
}
```

Simple Profile Property Type Example

This example shows a basic profile property type for storing a first name:

```
{
 "metadata": {
  "id": "firstName",
  "name": "First name",
  "systemTags": [
   "properties",
   "profileProperties",
   "basicProfileProperties",
   "personalIdentifierProperties"
  ]
 },
 "type": "string",
 "defaultValue": "",
 "automaticMappingsFrom": [
  "j:firstName"
 ],
 "rank": "101.0"
}
```

18.29.7. Dynamic Creation and Management

Property types are designed to be **dynamically created and updated at runtime**. This makes them ideal for scenarios where property definitions need to change without code deployments or server restarts.

Key Characteristics

- **Runtime Creation**: Property types can be created, updated, and deleted via REST API or GraphQL without restarting the server
- **No Code Deployment Required**: Changes to property types take effect immediately

- **UI-Driven**: Primarily used to inform UIs about property structure and constraints
- **Flexible**: Can be modified to adapt to changing business requirements
- **Multi-tenant**: Property types can be scoped to specific tenants

Use Cases for Dynamic Property Types

- **Custom Fields**: Allow users to define custom profile or session properties through a UI
- **A/B Testing**: Quickly add or modify property definitions for testing
- **Configuration-Driven UIs**: Build forms and property editors that adapt to property type definitions
- **Integration Flexibility**: Add properties for new integrations without code changes
- **Business Rule Changes**: Update property constraints and metadata as business rules evolve

18.29.8. Creating Property Types

Property types can be created in several ways:

Via JSON Files in Plugins

The most common way is to include property type definitions as JSON files in your plugin bundle:

1. Create JSON files in your plugin's resources directory:
    - Profile properties: META-INF/cxs/properties/profiles/...
    - Session properties: META-INF/cxs/properties/sessions/...
2. The target is automatically determined from the directory structure:
    - Files in properties/profiles/ → target = "profiles"
    - Files in properties/sessions/ → target = "sessions"
3. Subdirectories can be used for organization (e.g., profiles/personal/, profiles/contact/)

Via REST API

Property types can be created or updated using the REST API:

```
POST /cxs/profiles/properties
Content-Type: application/json

{
  "metadata": {
    "id": "customProperty",
    "name": "Custom Property",
    "systemTags": ["properties", "profileProperties"]
  },
  "type": "string",
  "defaultValue": "",
  "target": "profiles",
  "rank": "100.0"
}
```

Via GraphQL API

Property types can also be managed through the GraphQL API using property type mutations and queries.

18.29.9. Querying Property Types

Get All Property Types by Target

Retrieve all property types for a specific target:

```
GET /cxs/profiles/properties/targets/profiles
```

Response:

```
[
  {
    "metadata": {
      "id": "firstName",
      "name": "First name",
      ...
    },
    "type": "string",
    ...
  },
  ...
]
```

Get All Property Types

Retrieve all property types grouped by target:

```
GET /cxs/profiles/properties
```

Response:

```
{
  "profiles": [...],
  "sessions": [...]
  // Note: Event property types are NOT included here as they are not stored as PropertyType items
}
```

Get Property Type by ID

Retrieve a specific property type:

```
GET /cxs/profiles/properties/firstName
```

Get Property Types by Tag

Retrieve property types with specific tags:

```
GET /cxs/profiles/properties/tags/profileProperties,personalProfileProperties
```

Get Property Types by System Tag

Retrieve property types with specific system tags:

```
GET /cxs/profiles/properties/systemTags/profileProperties
```

18.29.10. Property Type Features

Multi-valued Properties

Set multivalued to true to allow a property to contain multiple values:

```
{
  "metadata": {
    "id": "interests",
    "name": "Interests"
  },
  "type": "string",
  "multivalued": true,
  "rank": "200.0"
}
```

Protected Properties

Set protected to true to make a property read-only (note: use "protected" in JSON, not "protekted"):

```
{
  "metadata": {
    "id": "systemId",
    "name": "System ID"
  },
  "type": "string",
  "protected": true,
  "rank": "1.0"
}
```

Numeric Ranges

Define ranges for numeric properties to enable categorization:

```
{
  "metadata": {
    "id": "income",
    "name": "Income"
  },
  "type": "integer",
  "numericRanges": [
    {"key": "low", "to": 30000},
    {"key": "medium", "from": 30000, "to": 100000},
    {"key": "high", "from": 100000}
  ]
}
```

Date Ranges

Define ranges for date properties:

```
{
  "metadata": {
    "id": "registrationDate",
    "name": "Registration Date"
  },
  "type": "date",
  "dateRanges": [
    {"key": "recent", "from": "2024-01-01"},
    {"key": "old", "to": "2023-12-31"}
  ]
}
```

Merge Strategies

Specify how properties should be merged when profiles are combined:

```
{
  "metadata": {
    "id": "lastLoginDate",
    "name": "Last Login Date"
  },
  "type": "date",
  "mergeStrategy": "mostRecentMergeStrategy",
  "rank": "300.0"
}
```

Common merge strategies include: * defaultMergeStrategy - Default merge behavior (typically uses the first non-null value) * mostRecentMergeStrategy - Use the most recent value (for dates and timestamps) * oldestMergeStrategy - Use the oldest value (for dates and timestamps) * addMergeStrategy - Add values together (for numeric types) * nonEmptyMergeStrategy - Prefer non-empty values * sumByKeyMergeStrategy - For a list of {key, value} objects, add value for entries that share a key. A value may be a number or a numeric string; entries that cannot be summed are kept unchanged

The built-in profile property interests uses sumByKeyMergeStrategy.



The merge strategy identifier in property type JSON files should match the id field from the merge strategy definition JSON files located in the META-INF/cxs/mergers/ directory (this is a file system path within plugin bundles, not an API endpoint).

Nested Properties

For complex objects, use childPropertyTypes to define nested structures:

```
{
 "metadata": {
  "id": "address",
  "name": "Address"
 },
 "type": "set",
 "childPropertyTypes": [
  {
   "metadata": {
    "id": "street",
    "name": "Street"
   },
   "type": "string"
  },
  {
   "metadata": {
    "id": "city",
    "name": "City"
   },
   "type": "string"
  },
  {
   "metadata": {
    "id": "zipCode",
    "name": "Zip Code"
   },
   "type": "string"
  }
 ]
}
```

18.29.11. Tags and System Tags

Property types support two types of tags for categorization: tags and systemTags. Understanding the difference between them is important for organizing and filtering property types in UIs.

Tags vs System Tags

Tags (tags field): * User-editable tags that can be modified through UIs * Intended for custom categorization and organization * Can be added, removed, or changed by end users * Perfect for creating custom groupings for UI display * Example use cases: grouping properties by department, project, or custom business logic

System Tags (systemTags field): * Reserved tags typically populated through JSON descriptors * Not meant to be modified by end users * Used for system-level categorization and filtering * Help classify properties for internal system logic * Example: profileProperties, sessionProperties, personalIdentifierProperties

Using Tags to Group Properties in UIs

Tags provide a flexible way to organize property types for display and editing in user interfaces. Here's how you can leverage tags:

UI Grouping Strategy

1. **Create Custom Tags**: Assign custom tags to property types based on how you want to group them in your UI
2. **Query by Tags**: Use the REST API to retrieve property types filtered by specific tags
3. **Display in Groups**: Organize properties in tabs, sections, or accordions based on their tags

Example: Grouping Properties by Department

Suppose you want to group profile properties by department in your UI:

```
// Marketing properties
{
  "metadata": {
    "id": "marketingSource",
    "name": "Marketing Source",
    "tags": ["marketing", "acquisition"]
  },
  "type": "string"
}

// Sales properties
{
  "metadata": {
    "id": "salesRep",
    "name": "Sales Representative",
    "tags": ["sales", "account-management"]
  },
  "type": "string"
}

// Support properties
{
  "metadata": {
    "id": "supportTier",
    "name": "Support Tier",
    "tags": ["support", "customer-service"]
  },
  "type": "string"
}
```

Then in your UI, you can fetch and display properties grouped by tag:

```
# Get all marketing properties
GET /cxs/profiles/properties/tags/marketing

# Get all sales properties
GET /cxs/profiles/properties/tags/sales

# Get all support properties
GET /cxs/profiles/properties/tags/support
```

Example: Multi-Tag Filtering

Properties can have multiple tags, allowing for flexible filtering:

```
{
 "metadata": {
  "id": "vipCustomer",
  "name": "VIP Customer",
  "tags": ["customer-tier", "priority", "billing"]
 },
 "type": "boolean"
}
```

You can query by multiple tags (comma-separated):

```
# Get properties with any of these tags
GET /cxs/profiles/properties/tags/customer-tier,priority,billing
```

Creating and Assigning Tags

Creating Tags

Tags don't need to be pre-registered. Simply assign a tag string to a property type, and it becomes available for filtering. Tags are created implicitly when first used.

Assigning Tags via JSON Files

When defining property types in JSON files, include tags in the metadata:

```
{
 "metadata": {
  "id": "customProperty",
  "name": "Custom Property",
  "tags": ["my-custom-tag", "another-tag"],
  "systemTags": ["properties", "profileProperties"]
 },
 "type": "string",
 "rank": "100.0"
}
```

Assigning Tags via REST API

You can add or modify tags when creating or updating a property type:

```
POST /cxs/profiles/properties
Content-Type: application/json

{
  "metadata": {
    "id": "newProperty",
    "name": "New Property",
    "tags": ["custom-group", "editable"],
    "systemTags": ["properties", "profileProperties"]
  },
  "type": "string",
  "target": "profiles"
}
```

Updating Tags on Existing Property Types

To add tags to an existing property type, retrieve it, modify the tags, and save it:

```
# 1. Get the existing property type
GET /cxs/profiles/properties/existingProperty

# 2. Update it with new tags
POST /cxs/profiles/properties
Content-Type: application/json

{
  "metadata": {
    "id": "existingProperty",
    "name": "Existing Property",
    "tags": ["existing-tag", "new-tag", "another-tag"],
    "systemTags": ["properties", "profileProperties"]
  },
  "type": "string"
}
```

Assigning Tags via GraphQL

Tags can also be managed through the GraphQL API when creating or updating property types.

Tag-Based UI Organization Examples

Example 1: Tabbed Interface

Organize properties into tabs based on tags:

```
// Fetch properties grouped by tags
const marketingProps: PropertyType[] = await fetch('/cxs/profiles/properties/tags/marketing')
  .then(res => res.json());
const salesProps: PropertyType[] = await fetch('/cxs/profiles/properties/tags/sales')
  .then(res => res.json());
const supportProps: PropertyType[] = await fetch('/cxs/profiles/properties/tags/support')
  .then(res => res.json());

// Display in tabs (pseudo-rendering - in real UI framework, this would be JSX/TSX)
console.log('Tabs:');
console.log('  Tab: Marketing');
console.log('    PropertyEditor with', marketingProps.length, 'properties');
console.log('  Tab: Sales');
console.log('    PropertyEditor with', salesProps.length, 'properties');
console.log('  Tab: Support');
console.log('    PropertyEditor with', supportProps.length, 'properties');
```

Example 2: Accordion Sections

Group properties in collapsible sections:

```
const allProps: PropertyType[] = await fetch('/cxs/profiles/properties/targets/profiles')
  .then(res => res.json());

// Group properties by tag
function groupByTag(propertyTypes: PropertyType[]): Record<string, PropertyType[]> {
  const grouped: Record<string, PropertyType[]> = {};
  for (const pt of propertyTypes) {
    const tags = pt.metadata.tags || [];
    for (const tag of tags) {
      if (!grouped[tag]) {
        grouped[tag] = [];
      }
      grouped[tag].push(pt);
    }
  }
  return grouped;
}

const grouped = groupByTag(allProps);

// Display in accordion (pseudo-rendering - in real UI framework, this would be JSX/TSX)
console.log('Accordion:');
for (const [tag, properties] of Object.entries(grouped)) {
  console.log(`  AccordionItem: ${tag}`);
  console.log(`    PropertyEditor with ${properties.length} properties`);
}
```

Example 3: Filterable List

Allow users to filter properties by tag:

```
let selectedTag: string | null = null;

const allProps: PropertyType[] = await fetch('/cxs/profiles/properties/targets/profiles')
  .then(res => res.json());

// Extract unique tags
function extractUniqueTags(propertyTypes: PropertyType[]): string[] {
  const tags: string[] = [];
  for (const pt of propertyTypes) {
    const propertyTags = pt.metadata.tags || [];
    for (const tag of propertyTags) {
      if (!tags.includes(tag)) {
        tags.push(tag);
      }
    }
  }
  return tags;
}

const uniqueTags = extractUniqueTags(allProps);

// Filter properties based on selected tag
const filteredProps: PropertyType[] = selectedTag
  ? await fetch(`/cxs/profiles/properties/tags/${selectedTag}`).then(res => res.json())
  : allProps;

// Display (pseudo-rendering - in real UI framework, this would be JSX/TSX)
console.log('Select dropdown:');
console.log('  Option: All Properties');
for (const tag of uniqueTags) {
  console.log(`  Option: ${tag}`);
}

console.log(`PropertyList with ${filteredProps.length} properties`);
```

Best Practices for Tags

1. **Use Descriptive Tag Names**: Choose clear, meaningful tag names (e.g., marketing, sales, support rather than m, s, sup)
2. **Create Hierarchical Tags**: Use dot notation for hierarchical organization (e.g., department.marketing, department.sales)
3. **Use Consistent Tagging**: Establish a tagging convention and apply it consistently across all property types
4. **Combine Tags and System Tags**: Use system tags for system-level categorization and tags for UI-level organization
5. **Document Tag Conventions**: Maintain documentation about your tagging conventions so team members use tags consistently
6. **Avoid Over-Tagging**: Don't assign too many tags to a single property type - 2-4 tags is usually sufficient

7. **Use Tags for Permissions**: Consider using tags to control which properties are visible or editable by different user roles

18.29.12. System Tags

System tags are used to categorize property types. Common system tags include:

- properties - General property tag
- profileProperties - Properties for profiles
- sessionProperties - Properties for sessions
- basicProfileProperties - Basic profile information
- personalProfileProperties - Personal information
- contactProfileProperties - Contact information
- workProfileProperties - Work-related information
- socialProfileProperties - Social media identifiers
- systemProfileProperties - System-managed properties (automatically maintained by Unomi)
- leadProfileProperties - Lead management properties
- personalIdentifierProperties - Properties that identify individuals (may be subject to privacy regulations)
- geographicSessionProperties - Geographic session information
- technicalSessionProperties - Technical session information

18.29.13. Validation and Property Types



Property types provide **informational metadata** for UIs but do **not enforce server-side validation**. The ranges, types, and constraints defined in property types are hints for UI components, not strict validation rules.

What Property Types Do

- **Inform UIs**: Tell user interfaces what properties exist, their types, and how to display them
- **Provide Hints**: Give UI components information about expected data formats and constraints
- **Enable Dynamic Forms**: Allow UIs to generate forms and property editors automatically
- **Organize Display**: Help organize properties in UIs using tags, ranks, and categories

What Property Types Don't Do

- **Server-Side Validation**: Property types do not validate data on the server
- **Enforce Constraints**: Ranges and constraints are informational, not enforced

- **Prevent Invalid Data**: Invalid data can still be stored if not validated elsewhere
- **Type Safety**: Property types don't guarantee type safety at the storage level

When Validation Happens

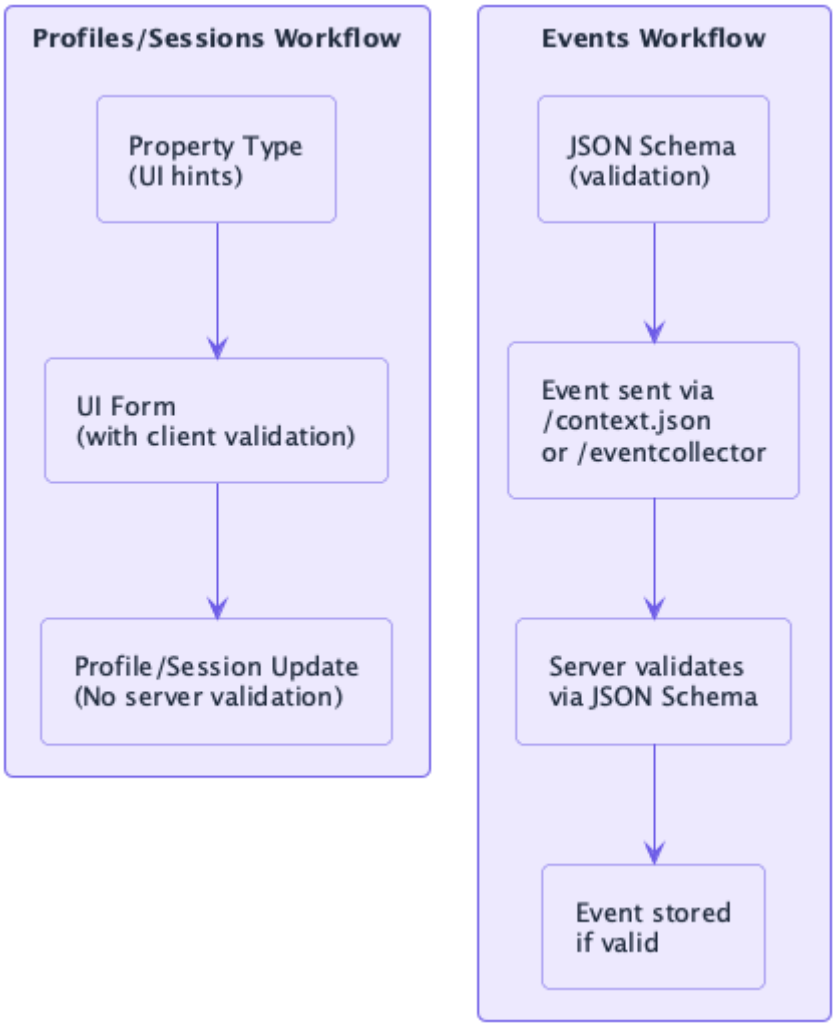
- **JSON Schemas**: Used exclusively for server-side validation of events sent through /context.json and /eventcollector endpoints
- **Property Types**: Provide no server-side validation - they are informational only
- **UI Validation**: Implement client-side validation in your UI based on property type hints (recommended for profiles and sessions)
- **Application Logic**: Add validation in your application code if needed for profiles and sessions
- **Database Constraints**: Use database-level constraints if your persistence layer supports them

Best Practice: Property Types for Profiles/Sessions, JSON Schemas for Events

Property types and JSON schemas serve different purposes for different objects:

For Profiles and Sessions: * Use property types to build UIs for editing profile/session properties * Implement client-side validation in your UI based on property type hints * Property types provide no server-side validation - data is stored as-is

For Events: * Use JSON schemas to validate events sent through /context.json or /eventcollector * Event property types are not accessible via REST API (they're generated internally) * To understand event structure, query actual events or use JSON schemas



18.29.14. Best Practices

1. **Use descriptive names**: Property type IDs should be clear and descriptive (e.g., firstName rather than fn).
2. **Set appropriate ranks**: Use the rank field to control display order in UIs. Lower numbers appear first.
3. **Use system tags**: Apply appropriate system tags to help categorize and filter property types.
4. **Organize with subdirectories**: Use subdirectories in your plugin structure to organize related property types.
5. **Define ranges for categorization**: Use numeric ranges, date ranges, or IP ranges to enable automatic categorization of values.
6. **Mark sensitive properties**: Use personalIdentifierProperties system tag for properties that identify individuals.
7. **Set merge strategies**: Define appropriate merge strategies for properties that may need to be merged when profiles are combined.
8. **Use appropriate value types**: Choose the correct value type (string, integer, date, etc.) for your data.

18.29.15. Built-in Property Types

Apache Unomi comes with a comprehensive set of built-in property types that cover common use cases. These are organized by category and target type.

Built-in Profile Property Types

Basic Profile Properties

These properties store basic identifying information about profiles:

- firstName (string) - First name of the profile
- lastName (string) - Last name of the profile
- gender (string) - Gender
- nationality (string) - Nationality

All basic profile properties are tagged with basicProfileProperties. firstName and lastName are also tagged with personalIdentifierProperties.

Personal Profile Properties

These properties store personal information:

- age (integer) - Age with numeric ranges (0-10, 10-20, 20-30, 30-40, 40-50, 50+)
- birthDate (date) - Birth date with date ranges
- kids (integer) - Number of children
- maritalStatus (string) - Marital status

All personal profile properties are tagged with personalProfileProperties.

Contact Profile Properties

These properties store contact information:

- email (email) - Email address
- phoneNumber (string) - Phone number
- address (string) - Street address
- city (string) - City
- zipCode (string) - Zip/postal code
- countryName (string) - Country name

All contact profile properties are tagged with contactProfileProperties. Email, address, and phoneNumber are also tagged with personalIdentifierProperties.

Work Profile Properties

These properties store work-related information:

- company (string) - Company name
- jobTitle (string) - Job title
- income (integer) - Income

All work profile properties are tagged with workProfileProperties.

Social Profile Properties

These properties store social media identifiers:

- facebookId (string) - Facebook ID
- twitterId (string) - Twitter ID
- linkedInId (string) - LinkedIn ID
- googleId (string) - Google ID

All social profile properties are tagged with socialProfileProperties.

System Profile Properties

These properties are automatically managed by Unomi and are typically protected (read-only):

- firstVisit (date) - Date of first visit (protected, uses oldestMergeStrategy)
- lastVisit (date) - Date of last visit (protected, uses mostRecentMergeStrategy)
- previousVisit (date) - Date of previous visit (protected, uses mostRecentMergeStrategy)
- nbOfVisits (integer) - Number of visits with ranges (protected, uses addMergeStrategy)
- totalNbOfVisits (integer) - Total number of visits (protected, uses addMergeStrategy)

All system profile properties are tagged with systemProfileProperties.

Lead Profile Properties

- leadAssignedTo (string) - Person assigned to the lead

All lead profile properties are tagged with leadProfileProperties.

Built-in Session Property Types

Geographic Session Properties

These properties store geographic information about the session:

- sessionCity (string) - City where the session occurred
- sessionCountryName (string) - Country name
- sessionCountryCode (string) - Country code
- sessionAdminSubDiv1 (string) - First-level administrative subdivision (e.g., state, province)
- sessionAdminSubDiv2 (string) - Second-level administrative subdivision (e.g., county)
- latitude (string) - Latitude coordinate
- longitude (string) - Longitude coordinate

All geographic session properties are tagged with geographicSessionProperties.

Technical Session Properties

These properties store technical information about the session:

- deviceCategory (string) - Device category (e.g., desktop, mobile, tablet)
- operatingSystemName (string) - Operating system name
- operatingSystemFamily (string) - Operating system family
- userAgentName (string) - User agent name (browser name)
- userAgentVersion (string) - User agent version (browser version)
- pageReferringURL (string) - URL of the referring page

All technical session properties are tagged with technicalSessionProperties.

Property Type Organization

Built-in property types are organized in the following directory structure:

```
META-INF/cxs/properties/
  profiles/
    basic/          # Basic identifying information
    personal/       # Personal information
    contact/        # Contact information
    work/           # Work-related information
    social/         # Social media identifiers
    system/         # System-managed properties
    lead/           # Lead management
  sessions/
    geographic/     # Geographic information
    technical/      # Technical information
```

The directory structure helps organize property types, and the target is automatically determined from the parent directory (profiles or sessions).

Viewing Built-in Property Types

You can view all built-in property types using the REST API:

```
# Get all profile property types
GET /cxs/profiles/properties/targets/profiles

# Get all session property types
GET /cxs/profiles/properties/targets/sessions

# Get property types by system tag
GET /cxs/profiles/properties/systemTags/profileProperties
GET /cxs/profiles/properties/systemTags/sessionProperties
```

18.29.16. Property Types vs JSON Schemas

Property types and JSON schemas serve complementary but distinct roles in Apache Unomi:

Property Types: UI-Focused Metadata

Property types are: * **Primarily for UIs**: Inform user interfaces about property structure, display names, types, and constraints * **Dynamically managed**: Can be created, updated, and deleted at runtime via REST API or GraphQL * **Informational**: Provide metadata about properties but don't enforce strict server-side validation * **Flexible**: Allow UIs to adapt to changing property definitions without code changes * **Target-specific**: Organized by target (profiles and sessions)

Use property types when you need to: * Build dynamic forms and property editors in UIs * Display property information to users * Organize properties by tags or categories * Control property visibility and editability in interfaces * Provide hints about data types and constraints to UI components

JSON Schemas: Server-Side Validation

JSON schemas are: * **For validation**: Enforce strict data validation on the server side * **Event-only**: Used exclusively to validate events sent through public endpoints (/context.json and /eventcollector) * **Standard-based**: Use the JSON Schema standard for validation * **Persistent**: Stored in Elasticsearch/OpenSearch and loaded at startup * **Required for events**: Each event type must have an associated JSON schema with target "events"

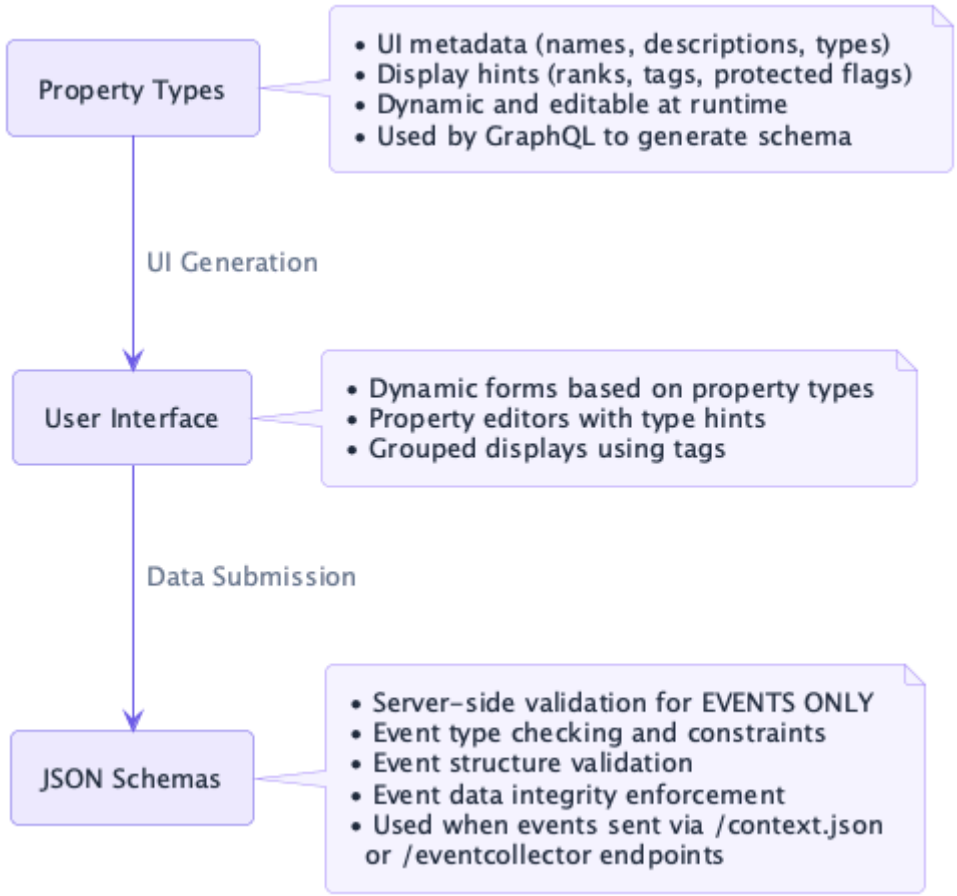


While JSON schemas technically support targets like "profiles", "sessions", "rules", and "segments", in practice they are **only used for event validation**. The validation code only processes schemas with target "events" when validating incoming requests through the public endpoints.

Use JSON schemas when you need to: * Validate incoming events from external sources * Ensure event data integrity and type safety * Enforce business rules and constraints on events * Validate event structures before processing

How They Work Together

Property types and JSON schemas complement each other:



Example Workflow:

1. **Property Type Definition**: A property type defines email as a string with description "Email address" for profiles
2. **UI Generation**: The UI uses the property type to render an email input field with appropriate validation hints for profile editing
3. **User Input**: User enters an email address in the UI
4. **Data Storage**: Data is stored in the profile (property types don't validate, but UI can implement client-side validation)
5. **Event Validation (separate)**: When events are sent through /context.json or /eventcollector, JSON schemas validate the event structure (not profile properties)

When to Use Each

Use Property Types for: * Building dynamic property editors * Organizing properties in UIs (tabs, sections, filters) * Providing type hints to form builders * Controlling property visibility and editability * GraphQL schema generation

Use JSON Schemas for: * Validating incoming events from external sources (the only actual use case) * Enforcing event data type constraints * Validating complex nested event structures * Ensuring event

data integrity on the server

Use Both When: * You want UI hints for profiles/sessions (property types) AND need to validate events (JSON schemas) * Building UIs that display profile/session data (property types) while also handling event validation (JSON schemas)

18.29.17. Property Types for Events

Property types are **not used for events**. Events are immutable records of user interactions and should not be edited through property type-based UIs.

If you need to understand event structure for display purposes: * Query actual events to see their structure * Use JSON schemas (with target "events") to understand the expected structure of events * JSON schemas define the validation rules for events sent through public endpoints

Event validation is performed by JSON schemas (with target "events") when events are sent through the public endpoints (/context.json and /eventcollector).

18.29.18. Building UIs with Property Types

Property types are designed to enable dynamic UI generation. Here's how to use them effectively:

Step 1: Fetch Property Types

Retrieve property types for your target (profiles or sessions):

```
# Fetch all profile property types
GET /cxs/profiles/properties/targets/profiles

# Fetch session property types
GET /cxs/profiles/properties/targets/sessions

# Fetch by tag for grouping
GET /cxs/profiles/properties/tags/marketing
```

TypeScript example:

```
// Fetch property types from REST API
const propertyTypes: PropertyType[] = await fetch('/cxs/profiles/properties/targets/profiles')
  .then(res => res.json());

// Or fetch by tag
const marketingProps: PropertyType[] = await fetch('/cxs/profiles/properties/tags/marketing')
  .then(res => res.json());
```

Step 2: Generate Form Fields

Use property type metadata to generate appropriate form fields:

```
interface FormField {
  id: string;
  name: string;
  description?: string;
  inputType: string;
  multivalued: boolean;
  defaultValue: string;
  readOnly: boolean;
  tags: string[];
}

function generateFormField(propertyType: PropertyType): FormField {
  // Extract property type information
  const id = propertyType.metadata.id;
  const name = propertyType.metadata.name;
  const description = propertyType.metadata.description;
  const valueType = propertyType.type;
  const isMultivalued = propertyType.multivalued || false;
  const defaultValue = propertyType.defaultValue || '';
  const isProtected = propertyType.protected || false;

  // Determine input type based on value type
  let inputType: string;
  if (valueType === 'integer' || valueType === 'long') {
    inputType = 'number';
  } else if (valueType === 'date') {
    inputType = 'date';
  } else if (valueType === 'boolean') {
    inputType = 'checkbox';
  } else if (valueType === 'email') {
    inputType = 'email';
  } else {
    inputType = 'text';
  }

  // Return field configuration
  return {
    id,
    name,
    description,
    inputType,
    multivalued: isMultivalued,
    defaultValue,
    readOnly: isProtected,
    tags: propertyType.metadata.tags || []
  };
}

// Generate form fields for all property types
const formFields: FormField[] = propertyTypes.map(generateFormField);
```

Step 3: Render Dynamic Forms

Use the generated fields to render your form:

```
interface PropertyFormProps {
  propertyTypes: PropertyType[];
  onSubmit: (values: Record<string, any>) => void;
}

function renderPropertyForm({ propertyTypes, onSubmit }: PropertyFormProps): void {
  // Initialize form values
  const values: Record<string, any> = {};

  // Generate form fields
  const fields: FormField[] = propertyTypes.map(generateFormField);

  // Group fields by tags
  const groupedFields: Record<string, FormField[]> = {};
  for (const field of fields) {
    const tag = field.tags[0] || 'other';
    if (!groupedFields[tag]) {
      groupedFields[tag] = [];
    }
    groupedFields[tag].push(field);
  }

  // Render form (pseudo-rendering logic)
  for (const [tag, tagFields] of Object.entries(groupedFields)) {
    // Render fieldset with legend
    console.log(`Fieldset: ${tag}`);

    for (const field of tagFields) {
      // Render label
      console.log(`Label: ${field.name}`);

      if (field.description) {
        // Render help-text
        console.log(`Help: ${field.description}`);
      }

      if (field.multivalued) {
        // Render multi-value-input
        const currentValue = values[field.id] || [];
        // onChange handler would update values[field.id] = newValue
        console.log(`Multi-value input: ${field.inputType}`, currentValue);
      } else {
        // Render single input
        const currentValue = values[field.id] ?? field.defaultValue;
        // onChange handler would update values[field.id] = newValue
        console.log(`Input: ${field.inputType}`, currentValue, `readOnly: ${field.readOnly}`);
      }
    }
  }

  // Render submit button
  console.log('Submit button: Save');
}
```

Step 4: Handle Numeric Ranges

For properties with numeric ranges, you can provide range-based inputs:

```
interface NumericRange {
  key: string;
  from?: number;
  to?: number;
}

function renderNumericRangeInput(
  propertyType: PropertyType,
  value: string,
  onChange: (value: string) => void
): void {
  const numericRanges = propertyType.numericRanges;

  if (numericRanges && numericRanges.length > 0) {
    // Render as select dropdown with range options
    console.log('Select dropdown:');
    console.log('  Option: Select range');

    for (const range of numericRanges) {
      const label = formatRange(range);
      console.log(`  Option: ${label} (value: ${range.key})`);
    }
  } else {
    // Fallback to regular number input
    console.log(`Number input: ${value}`);
  }
}

function formatRange(range: NumericRange): string {
  if (range.from !== undefined && range.to !== undefined) {
    return `${range.from} - ${range.to}`;
  } else if (range.from !== undefined) {
    return `${range.from}+`;
  } else if (range.to !== undefined) {
    return `up to ${range.to}`;
  } else {
    return range.key;
  }
}
```

Step 5: Sort by Rank

Use the rank field to control display order:

```
// Sort property types by rank (lower numbers first)
function sortByRank(propertyTypes: PropertyType[]): PropertyType[] {
  return [...propertyTypes].sort((a, b) => {
    const rankA = a.rank ?? 999;  // Default rank for items without rank
    const rankB = b.rank ?? 999;
    return rankA - rankB;
  });
}

const sortedPropertyTypes: PropertyType[] = sortByRank(propertyTypes);
```

Complete Example: Dynamic Property Editor

Here's a complete example of a dynamic property editor:

```
interface PropertyType {
  metadata: {
    id: string;
    name: string;
    description?: string;
    tags?: string[];
    systemTags?: string[];
  };
  type: string;
  defaultValue?: string;
  multivalued?: boolean;
  protected?: boolean;
  rank?: number;
  numericRanges?: NumericRange[];
}

interface DynamicPropertyEditorState {
  propertyTypes: PropertyType[];
  values: Record<string, any>;
  selectedTag: string | null;
}

async function createDynamicPropertyEditor(
  target: string,
  onSave: (values: Record<string, any>) => void
): Promise<DynamicPropertyEditorState> {
  // Initialize state
  let propertyTypes: PropertyType[] = [];
  const values: Record<string, any> = {};
  let selectedTag: string | null = null;

  // Fetch property types from REST API
  propertyTypes = await fetch(`/cxs/profiles/properties/targets/${target}`)
    .then(res => res.json());

  // Initialize values with defaults
  for (const propertyType of propertyTypes) {
    if (propertyType.defaultValue != null) {
      values[propertyType.metadata.id] = propertyType.defaultValue;
```

```
  }
}

// Get unique tags for filtering
const tags: string[] = [];
for (const propertyType of propertyTypes) {
  const propertyTags = propertyType.metadata.tags || [];
  for (const tag of propertyTags) {
    if (!tags.includes(tag)) {
      tags.push(tag);
    }
  }
}

// Filter property types by selected tag
let filteredTypes = propertyTypes;
if (selectedTag != null) {
  filteredTypes = propertyTypes.filter(pt =>
    (pt.metadata.tags || []).includes(selectedTag!)
  );
}

// Sort by rank
const sortedTypes = sortByRank(filteredTypes);

// Render form (pseudo-rendering - in real UI framework, this would be JSX/TSX)
console.log('Rendering form...');

// Tag filter dropdown
if (tags.length > 0) {
  console.log('Tag filter dropdown:', selectedTag);
  console.log('  Option: All properties');
  for (const tag of tags) {
    console.log(`  Option: ${tag}`);
  }
}

// Property fields
for (const propertyType of sortedTypes) {
  const fieldId = propertyType.metadata.id;
  const fieldValue = values[fieldId] ?? propertyType.defaultValue ?? '';

  console.log(`Field: ${propertyType.metadata.name} (${fieldId})`);

  if (propertyType.metadata.description) {
    console.log(`  Description: ${propertyType.metadata.description}`);
  }

  if (propertyType.protected) {
    console.log(`  Input: text, value: ${fieldValue}, readOnly: true`);
  } else {
    propertyInput(propertyType, fieldValue, (newValue) => {
      values[fieldId] = newValue;
    });
  }
}
```

```
  // Submit button
  console.log('Submit button: Save Properties');

  return { propertyTypes, values, selectedTag };
}

function propertyInput(
  propertyType: PropertyType,
  value: any,
  onChange: (value: any) => void
): void {
  // Handle numeric ranges
  if (propertyType.numericRanges && propertyType.numericRanges.length > 0) {
    console.log('  Select dropdown with numeric ranges');
    console.log('    Option: Select...');
    for (const range of propertyType.numericRanges) {
      console.log(`    Option: ${formatRange(range)} (value: ${range.key})`);
    }
    return;
  }

  // Handle multivalued properties
  if (propertyType.multivalued) {
    const valueList: any[] = Array.isArray(value) ? value : (value ? [value] : []);
    console.log('  Multi-value input container:');
    for (let index = 0; index < valueList.length; index++) {
      const val = valueList[index];
      const inputType = getInputType(propertyType.type);
      console.log(`    Input[${index}]: ${inputType}, value: ${val}`);
      // onChange would update valueList[index] = newVal, then call onChange(valueList)
    }
    console.log('    Button: Add Value');
    return;
  }

  // Regular single-value input
  const inputType = getInputType(propertyType.type);
  console.log(`  Input: ${inputType}, value: ${value}`);
}

function getInputType(valueType: string): string {
  if (valueType === 'integer' || valueType === 'long') {
    return 'number';
  } else if (valueType === 'date') {
    return 'date';
  } else if (valueType === 'boolean') {
    return 'checkbox';
  } else if (valueType === 'email') {
    return 'email';
  } else {
    return 'text';
  }
}
```

18.29.19. Related Topics

- Data Model Overview - Learn about the overall data model
- Writing Plugins - Learn how to create plugins with property types
- JSON Schemas - Learn about JSON schema support in Unomi
- GraphQL API - Learn about GraphQL property type operations

18.30. Built-in Event types

Apache Unomi comes with built-in event types, which we describe below.

18.30.1. Login event type

The login event type is used to signal an authentication event has been triggered. This event should be "secured", meaning that it should not be accepted from any location, and by default Apache Unomi will only accept this event from configured "third-party" servers (identified by their IP address and a Unomi application key).

Usually, the login event will contain information passed by the authentication server and may include user properties and any additional information. Rules may be set up to copy the information from the event into the profile, but this is not done in the default set of rules provided by Apache Unomi for security reasons. You can find an example of such a rule here: exampleLogin.json (see Login sample — the event must be sent from a trusted server-side caller).

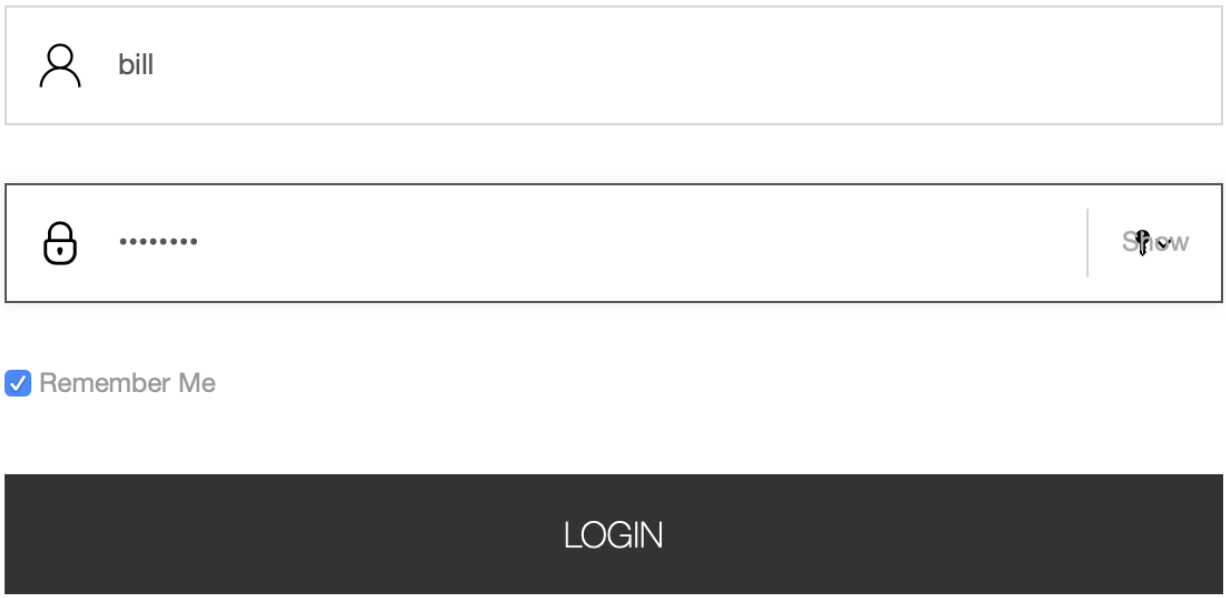
Structure overview

Based on the structure of the following object: Event

Field name	Value/description
eventType	login
source	Not used (null)
target	an Item representing the user that logged in
scope	the scope in which the user has authenticated
properties	Not used (empty)

Example

In this case, a user has logged into a site called "digitall", and his user information the following properties are associated with the active user..and perhaps show his visitor profile or user information.




```
{
  "itemId": "0b8825a6-efb8-41a6-bea5-d745b33c94cb",
  "itemType": "event",
  "scope": "digitall",
  "eventType": "login",
  "sessionId": "7b8a5f17-cdb0-4c14-b676-34c1c0de0825",
  "profileId": "f7d1f1b9-4415-4ff1-8fee-407b109364f7",
  "timeStamp": "2020-01-30T21:18:28Z",
  "properties": {},
  "source": null,
  "target": {
    "itemId": "13054a95-092d-4d7b-81f5-e4656c2ebc88",
    "itemType": "cmsUser",
    "scope": null,
    "properties": {
      "j:function": "Vice President",
      "preferredLanguage": "en",
      "j:title": "mister",
      "emailNotificationsDisabled": "true",
      "j:organization": "Acme Space",
      "j:gender": "male",
      "j:nodename": "bill",
      "j:lastName": "Galileo",
      "j:publicProperties":
"j:about,j:firstName,j:function,j:gender,j:lastName,j:organization,j:picture,j:title",
      "j:firstName": "Bill",
      "j:about": "<p> Lorem Ipsum dolor sit amet.</p> "
    }
  }
}
```

18.30.2. View event type

This event is triggered when a web page is viewed by a user. Some integrators might also want to trigger it when a single-page-application screen is displayed or when a mobile application screen is displayed.

Structure description

Based on the structure of the following object: Event

Field name	Value/description
eventType	view
source	the source for the view event, could be a web site, an application name, etc...
target	the page/screen being viewed
properties	Not used (empty)

Example

In this case a use has visited the home page of the digitall site. As this is the first page upon login, the destination and referring URL are the same.

```
{
  "itemId": "c75f50c2-ab55-4d95-be69-cbbeee180d6b",
  "itemType": "event",
  "scope": "digitall",
  "eventType": "view",
  "sessionId": "7b8a5f17-cdb0-4c14-b676-34c1c0de0825",
  "profileId": "f7d1f1b9-4415-4ff1-8fee-407b109364f7",
  "timeStamp": "2020-01-30T21:18:32Z",
  "properties": {},
  "source": {
    "itemId": "29f5fe37-28c0-48f3-966b-5353bed87308",
    "itemType": "site",
    "scope": "digitall",
    "properties": {}
  },
  "target": {
    "itemId": "f20836ab-608f-4551-a930-9796ec991340",
    "itemType": "page",
    "scope": "digitall",
    "properties": {
      "pageInfo": {
        "templateName": "home",
        "language": "en",
        "destinationURL": "http://localhost:8080/sites/digitall/home.html",
        "categories": [],
        "pageID": "f20836ab-608f-4551-a930-9796ec991340",
        "nodeType": "jnt:page",
        "pagePath": "/sites/digitall/home",
        "pageName": "Home",
        "referringURL": "http://localhost:8080/sites/digitall/home.html",
        "tags": [],
        "isContentTemplate": false
      },
      "attributes": {},
      "consentTypes": []
    }
  }
}
```


18.30.3. Form event type

This event type is used to track form submissions. These could range from login to survey form data captured and processed in Apache Unomi using rules.

Structure description

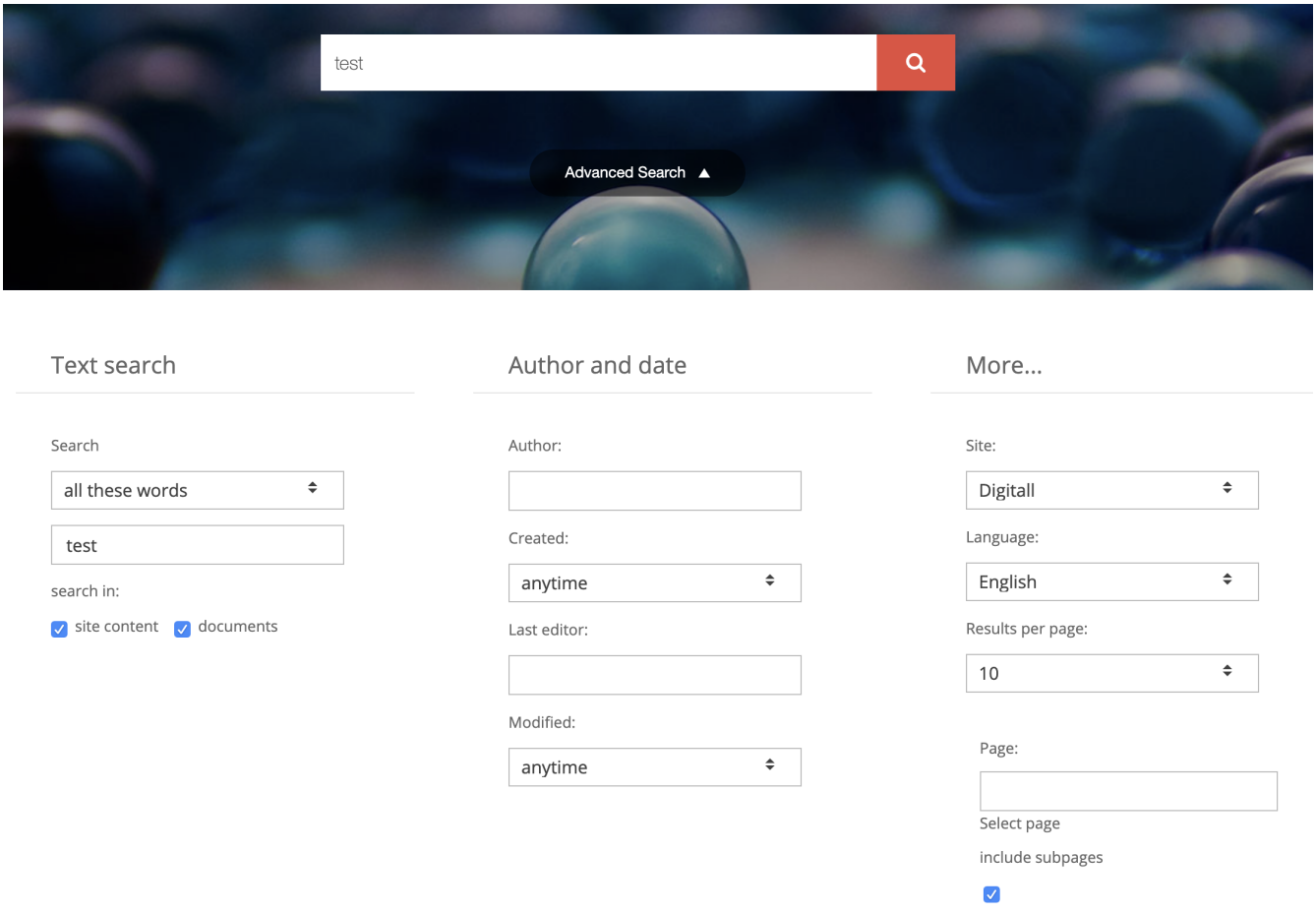
Based on the structure of the following object: Event

Field name	Value/description
eventType	form
source	the page/screen on which the form was submitted

Field name	Value/description
target	the form that was submitted (there could be more than one form on a page/screen)
properties	contain the data submitted in the form

Example

A form exists on the digitall site, and has been submitted by a visitor. In this case it was a search form that contains fields to adjust the search parameters.



```
{
  "itemId": "44177ffe-b5c8-4575-a8e5-f8aa0d4ee792",
  "itemType": "event",
  "scope": "digitall",
  "eventType": "form",
  "sessionId": "be416c08-8b9b-4611-990f-3a8bf3ed4e68",
  "profileId": "bc1e1238-a9ac-4b3a-8f63-5eec205cfcd5",
  "timeStamp": "2020-01-30T21:41:22Z",
  "properties": {
    "jcrMethodToCall": "get",
    "src_originSiteKey": "digitall",
    "src_terms[0].term": "test",
    "src_terms[0].applyFilter": "true",
    "src_terms[0].match": "all_words",
    "src_terms[0].fields.siteContent": "true",
    "src_terms[0].fields.tags": "true",
    "src_terms[0].fields.files": "true",
    "src_sites.values": "digitall",
    "src_sitesForReferences.values": "systemsite",
    "src_languages.values": "en"
  },
  "source": {
    "itemId": "97e14221-33dd-4608-82ae-9724d15d4f12",
    "itemType": "page",
    "scope": "digitall",
    "properties": {
      "pageInfo": {
        "templateName": "home",
        "language": "en",
        "destinationURL": "http://localhost:8080/sites/digitall/home/search-results.html",
        "categories": [],
        "pageID": "97e14221-33dd-4608-82ae-9724d15d4f12",
        "nodeType": "jnt:page",
        "pagePath": "/sites/digitall/home/search-results",
        "pageName": "Search Results",
        "referringURL": "http://localhost:8080/cms/edit/default/en/sites/digitall/home.html",
        "tags": [],
        "isContentTemplate": false
      },
      "attributes": {},
      "consentTypes": []
    }
  },
  "target": {
    "itemId": "searchForm",
    "itemType": "form",
    "scope": "digitall",
    "properties": {}
  }
}
```


18.30.4. Update properties event type

This event is usually used by user interfaces that make it possible to modify profile properties, for example a form where a user can edit his profile properties, or a management UI to modify.

Note that this event type is a protected event type that is only accepted from authorized source IP addresses (the tenant's authorizedIPs in normal mode, see Secured events configuration; in single-tenant compatibility mode instead an X-Unomi-Peer key valid for the provider's ipAddresses). That gate is separate from the public-versus-trusted caller distinction that governs which profile and session a request may bind to — see client-facing hardening. Once the event is accepted, the property names in add, update, addToSet and delete are checked against an allowlist. A public-key caller may only write plain dotted paths under properties (for example properties.address.city). A trusted caller (system administrator or tenant private key) may also write systemProperties, segments, scores and consents, and may target another profile through targetId. Identity and bookkeeping fields of the profile (itemId, itemType, tenantId, version, mergedWith, …) are never writable, and names using beanutils mapped or indexed syntax (a(b), a[0]), empty path segments or control characters are rejected for everyone. Refused names are skipped with a WARN log entry; the rest of the event still applies.

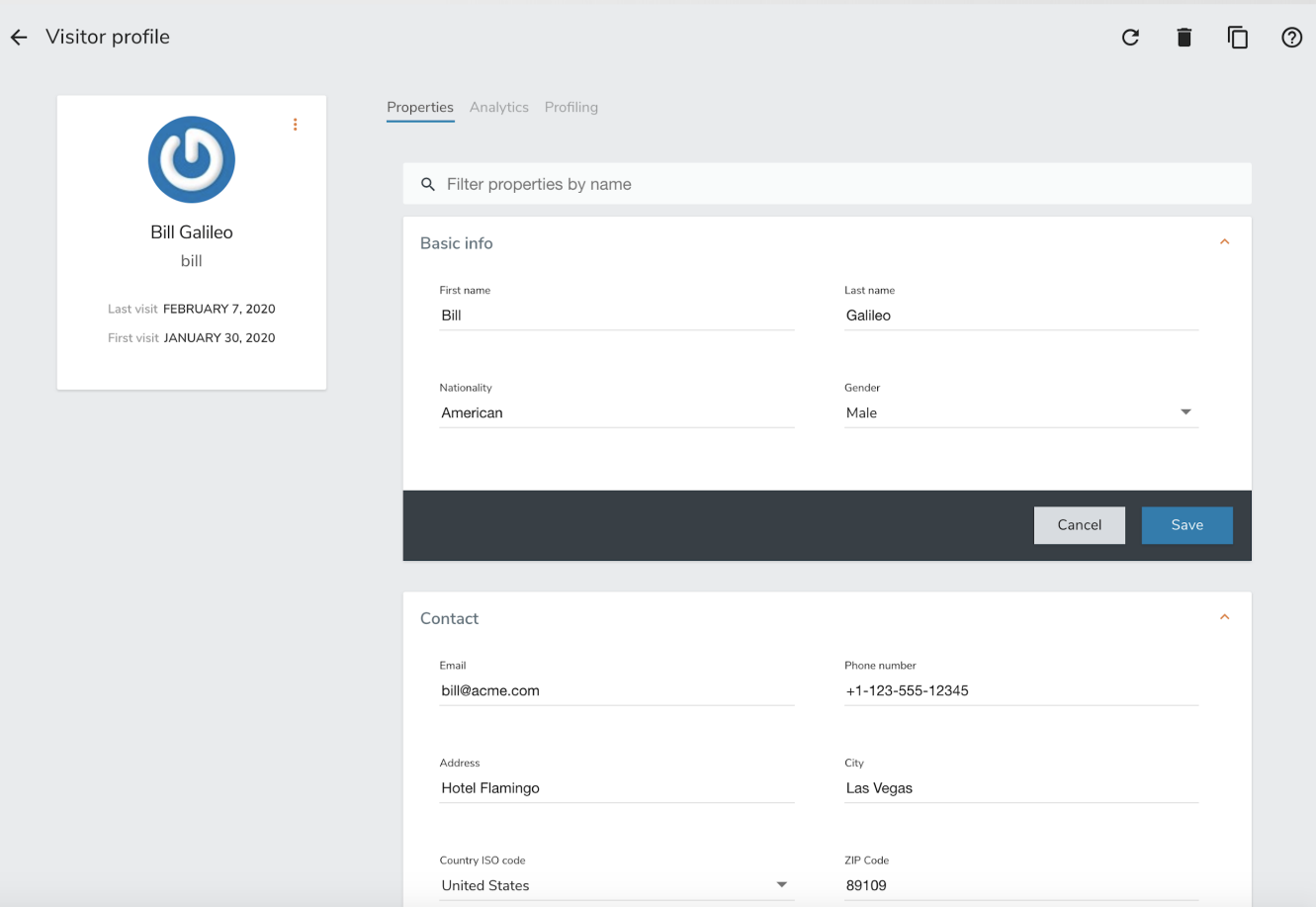
Structure definition

Based on the structure of the following object: Event

Field name	Value/description
eventType	updateProperties
source	the screen that has triggered the update to the profile properties
target	Not used (null)
properties	{ targetId: the identifier of the profile to update targetType: "profile" if updating a profile or "persona" for personas add/update/delete: properties to be added/updated or deleted on the target profile}

Example

In this example, this "updateProperties" event contains properties that must be added to the targetId profile.




```
{
  "itemId": "d8fec330-33cb-42bc-a4e2-bb48ea7ed29b",
  "itemType": "event",
  "scope": null,
  "eventType": "updateProperties",
  "sessionId": "66e63ec9-66bc-4fac-8a8a-febcc3d6cbb7",
  "profileId": "bc1e1238-a9ac-4b3a-8f63-5eec205cfcd5",
  "timeStamp": "2020-01-31T08:51:15Z",
  "properties": {
    "targetId": "f7d1f1b9-4415-4ff1-8fee-407b109364f7",
    "targetType": "profile",
    "add": {
      "properties.phoneNumber": "+1-123-555-12345",
      "properties.countryName": "US",
      "properties.city": "Las Vegas",
      "properties.address": "Hotel Flamingo",
      "properties.zipCode": "89109",
      "properties.email": "bill@acme.com"
    }
  },
  "source": {
    "itemId": "wemProfile",
    "itemType": "wemProfile",
    "scope": "digitall",
    "properties": {}
  },
  "target": null
}
```

18.30.5. Identify event type

This event type is used to add information learned about the current profile. This could be through a form that has asked the user to provide some information about himself, or it could be information sent by another system (CRM, SSO, DMP, LiveRamp or equivalent) to augment the data for the current profile.

It should be noted that, as in the case of a login event, it might be a good idea to be careful as to who and what system are allowed to send this event. Also, in order for this event to perform any modifications, an associated rule will be needed in the Unomi system to perform modifications to a profile (there is no default rule).

Event type	Available publicly	Default rule	Targeted at back-office	Can remove/update properties
identify	yes	no	no	no
updateProperties	no	yes	yes	yes

The rule of thumb is: if you need to send profile data from public system to add information to a profile you should use the identify event type and add a rule to only process the data you want to accept. If you want to add/update/delete properties in a secure manner from a known server you could use the updateProperties but you should always check first if you can't use the identify or event form event types with specific rules as this reduces greatly the potential for misuse.

Structure description

Based on the structure of the following object: Event

Field name	Value/description
eventType	identify
source	the site/application name that triggered the identify event
target	the user information contained in the event
properties	Not used (empty)

Example

In this example, an event containing additional information about the user (his nickname, favorite compiler and industry) was sent to Apache Unomi.

```
{
  "itemId": "18dfd6c7-9055-4ef0-a2eb-14c1482b4544",
  "itemType": "event",
  "scope": "myScope",
  "eventType": "identify",
  "sessionId": "928d9237-fb3d-4e53-cbee-1aeb1df7f03a",
  "profileId": "temp_023ded50-bb43-4fe2-acbc-13bfa8de16de",
  "timeStamp": "2020-01-15T14:13:25Z",
  "properties": {},
  "source": {
    "itemId": "myScope",
    "itemType": "site",
    "scope": "myScope",
    "properties": {
      "page": {
        "path": "/web-page/",
        "referrer": "http://localhost:8181/",
        "search": "",
        "title": "Apache Unomi Web Test Page",
        "url": "http://localhost:8181/web-page/"
      }
    }
  },
  "target": {
    "itemId": "null",
    "itemType": "analyticsUser",
    "scope": "myScope",
    "properties": {
      "nickname": "Amazing Grace",
      "favoriteCompiler": "A-0",
      "industry": "Computer Science"
    }
  }
}
```


18.30.6. Session created event type

The session created event is an internal event created by Apache Unomi when a new session is created. This indicates that a new visitor has interacted with a system that is using Apache Unomi to track their behavior.

Structure definition

Based on the structure of the following object: Event

Field name	Value/description
eventType	sessionCreated
source	Not used (null)
target	the Session item that was created with all its fields and properties

Field name	Value/description
properties	Not used (empty)

Example

In this example, a new session was created for a visitor coming to the digitall website. The session contains the firstVisit property. It may be augmented over time with more information including location.


```
{
  "itemId": "b3f5486f-b317-4182-9bf4-f497271e5363",
  "itemType": "event",
  "scope": "digitall",
  "eventType": "sessionCreated",
  "sessionId": "be416c08-8b9b-4611-990f-3a8bf3ed4e68",
  "profileId": "bc1e1238-a9ac-4b3a-8f63-5eec205cfcd5",
  "timeStamp": "2020-01-30T21:13:26Z",
  "properties": {},
  "source": null,
  "target": {
    "itemId": "be416c08-8b9b-4611-990f-3a8bf3ed4e68",
    "itemType": "session",
    "scope": "digitall",
    "profileId": "bc1e1238-a9ac-4b3a-8f63-5eec205cfcd5",
    "profile": {
      "itemId": "bc1e1238-a9ac-4b3a-8f63-5eec205cfcd5",
      "itemType": "profile",
      "properties": {
        "firstVisit": "2020-01-30T21:13:26Z"
      },
      "systemProperties": {},
      "segments": [],
      "scores": null,
      "mergedWith": null,
      "consents": {}
    },
    "properties": {},
    "systemProperties": {},
    "timeStamp": "2020-01-30T21:13:26Z",
    "lastEventDate": null,
    "size": 0,
    "duration": 0
  }
}
```


18.30.7. Goal event type

A goal event is triggered when the current profile (visitor) reaches a goal.

Structure definition

Based on the structure of the following object: Event

Field name	Value/description
eventType	goal
source	the Event that triggered the goal completion
target	the Goal item that was reached
properties	Not used (empty)

Example

In this example, a visitor has reached a goal by viewing a page called “sub-home” on the site “digitall” (event source). This goal event had the goal object as a target. The goal object (see Goal object later in this document) has a start event of creating a new session and a target event of a page view on the page “sub-home”.


```
{
  "itemId": "9fa70519-382d-412b-82ea-99b5989fbd0d",
  "itemType": "event",
  "scope": "digitall",
  "eventType": "goal",
  "sessionId": "42bd3fde-5fe9-4df6-8ae6-8550b8b06a7f",
  "profileId": "3ec46b2c-fbaa-42d5-99df-54199c807fc8",
  "timeStamp": "2017-05-29T23:49:16Z",
  "properties": {},
  "source": {
    "itemId": "aadcd86c-9431-43c2-bdc3-06683ac25927",
    "itemType": "event",
    "scope": "digitall",
    "eventType": "view",
    "sessionId": "42bd3fde-5fe9-4df6-8ae6-8550b8b06a7f",
    "profileId": "3ec46b2c-fbaa-42d5-99df-54199c807fc8",
    "timeStamp": "2017-05-29T23:49:16Z",
    "properties": {},
    "source": {
      "itemId": "6d5f4ae3-30c9-4561-81f3-06f82af7da1e",
      "itemType": "site",
      "scope": "digitall",
      "properties": {}
    },
    "target": {
      "itemId": "67dfc299-9b74-4264-a865-aebdc3482539",
      "itemType": "page",
      "scope": "digitall",
      "properties": {
        "pageInfo": {
          "language": "en",
          "destinationURL": "https://acme.com/home/sub-home.html",
          "pageID": "67dfc299-9b74-4264-a865-aebdc3482539",
          "pagePath": "/sites/digitall/home/sub-home",
```

```
            "pageName": "sub-home",
            "referringURL": "https://acme.com/home/perso-on-profile-past-event-page.html"
          },
          "category": {},
          "attributes": {}
        }
      }
    },
    "target": {
      "itemId": "_v4ref2mxg",
      "itemType": "goal",
      "startEvent": {
        "parameterValues": {},
        "type": "sessionCreatedEventCondition"
      },
      "targetEvent": {
        "parameterValues": {
          "pagePath": "/sites/digitall/home/sub-home"
        },
        "type": "pageViewEventCondition"
      },
      "campaignId": null,
      "metadata": {
        "id": "_v4ref2mxg",
        "name": "sub-home-visit",
        "description": "",
        "scope": "digitall",
        "tags": [
          "pageVisitGoal"
        ],
        "enabled": true,
        "missingPlugins": false,
        "hidden": false,
        "readOnly": false
      }
    }
  }
```


18.30.8. Modify consent event type

Consent type modification events are used to tell Unomi that consents were modified. A built-in rule will update the current profile with the consent modifications contained in the event. Consent events may be sent directly by a current profile to update their consents on the profile.

Structure definition

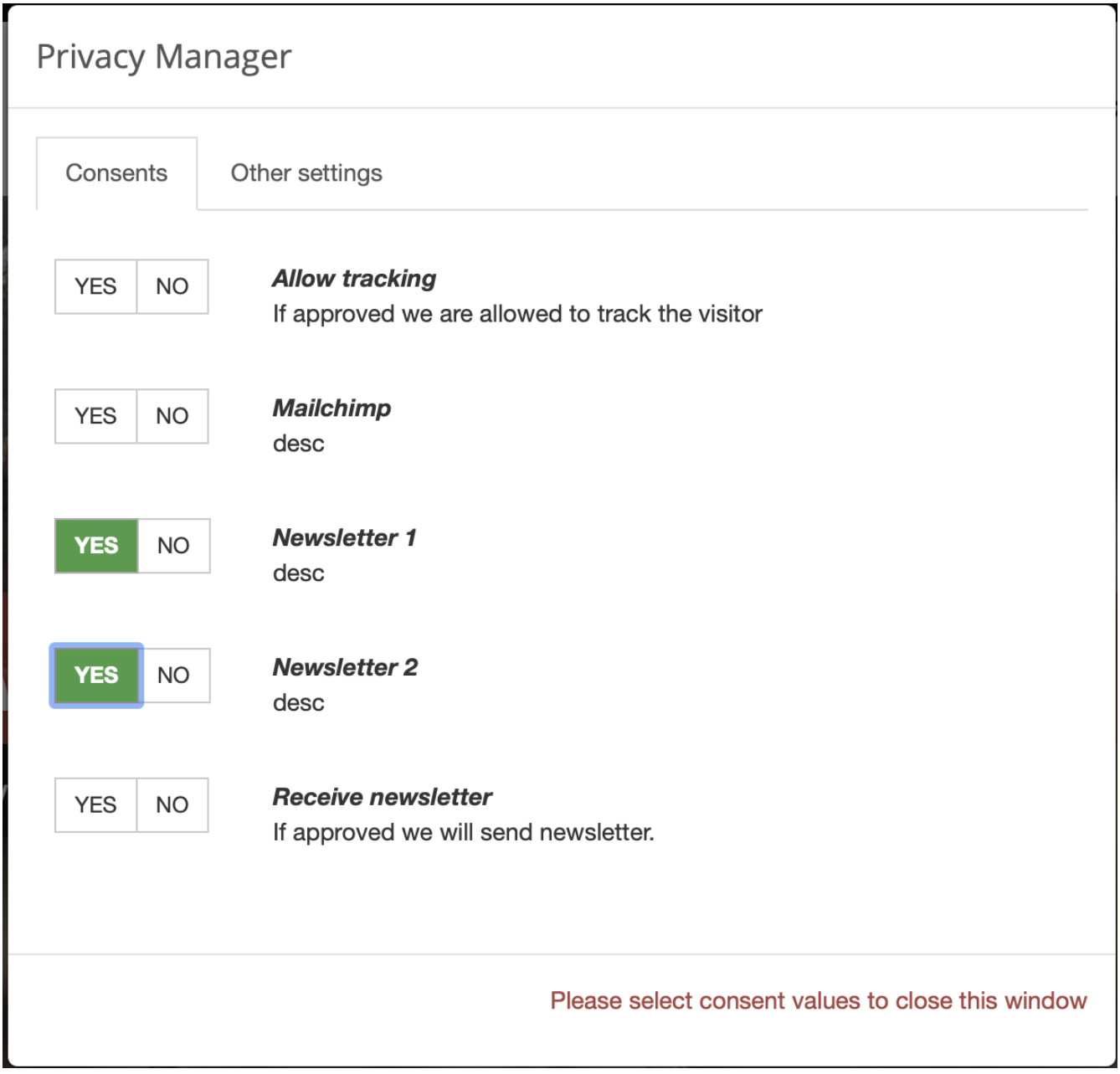
Based on the structure of the following object: Event

Field name	Value/description
eventType	modifyConsent

Field name	Value/description
source	the page that has triggered the update the consents and that contains the different consent types the current profile could grant or deny
target	The consent that was modified
properties	The consent's new value. See the Consent object type for more information.

Example

In this example, a user-generated a consent modification when visiting the home page, possibly by interacting with a consent form that captured his preferences. Different consent types were present on the page and he decided to GRANT the "tracking" consent.




```
{
```

```
"scope": "digitall",
"eventType": "modifyConsent",
"source": {
  "itemType": "page",
  "scope": "digitall",
  "itemId": "f20836ab-608f-4551-a930-9796ec991340",
  "properties": {
    "pageInfo": {
      "pageID": "f20836ab-608f-4551-a930-9796ec991340",
      "nodeType": "jnt:page",
      "pageName": "Home",
      "pagePath": "/sites/digitall/home",
      "templateName": "home",
      "destinationURL": "http://localhost:8080/sites/digitall/home.html",
      "referringURL": "http://localhost:8080/cms/render/default/en/sites/digitall/home.html",
      "language": "en",
      "categories": [],
      "tags": [],
      "isContentTemplate": false
    },
    "attributes": {},
    "consentTypes": [
      {
        "typeIdentifier": "tracking",
        "activated": true,
        "title": "Allow tracking",
        "description": "If approved we are allowed to track the visitor"
      },
      {
        "typeIdentifier": "newsletter1",
        "activated": true,
        "title": "Newsletter 1",
        "description": "desc"
      },
      {
        "typeIdentifier": "newsletter2",
        "activated": true,
        "title": "Newsletter 2",
        "description": "desc"
      },
      {
        "typeIdentifier": "newsletter",
        "activated": true,
        "title": "Receive newsletter",
        "description": "If approved we will send newsletter."
      }
    ]
  }
},
"target": {
  "itemType": "consent",
  "scope": "digitall",
  "itemId": "tracking"
},
"properties": {
  "consent": {
    "scope": "digitall",
```

```
        "typeIdentifier": "tracking",
        "status": "GRANTED",
        "statusDate": "2020-01-31T20:10:00.463Z",
        "revokeDate": "2022-01-30T20:10:00.463Z"
      }
    }
}
```

18.31. Built-in condition types

Apache Unomi ships with built-in condition types in the base plugin and optional extensions. For a hands-on introduction with examples, see Conditions guide. For save-time parameter rules (required, allowedValues, exclusive groups), see Condition validation.

18.31.1. Condition type descriptor shape

```
{
 "metadata": {
  "id": "booleanCondition",
  "name": "booleanCondition",
  "description": "Combines multiple conditions with a logical operator",
  "systemTags": ["logical", "profileCondition", "eventCondition"],
  "readOnly": true
 },
 "conditionEvaluator": "booleanConditionEvaluator",
 "queryBuilder": "booleanConditionQueryBuilder",
 "parameters": [
  {
   "id": "operator",
   "type": "string",
   "defaultValue": "and",
   "validation": { "required": true, "allowedValues": ["and", "or"] }
  },
  {
   "id": "subConditions",
   "type": "Condition",
   "multivalued": true,
   "validation": { "required": true }
  }
 ]
}
```

Parameters may include a validation object (see Condition validation). Real descriptors in the base plugin use this heavily — for example pastEventCondition requires an eventCondition sub-condition tagged eventCondition.

Each condition type provides:

- conditionEvaluator — real-time evaluation against profile, session, and/or event
- queryBuilder — converts the condition to an Elasticsearch or OpenSearch query (Unomi 3.0+ uses

generic *QueryBuilder IDs)

Some types use parentCondition to specialize a base type (for example eventTypeCondition extends eventPropertyCondition).

18.31.2. Plugin registration (Unomi 3.0+)

New plugins use **Declarative Services** (@Component). Legacy Blueprint XML remains in some shipped bundles during migration; do not copy it for new code. In particular, the built-in BooleanConditionEvaluator, BooleanConditionESQueryBuilder, and BooleanConditionOSQueryBuilder classes are themselves still registered via Blueprint XML today — the snippets below illustrate how a **new** condition type should be registered, not the current state of those specific built-in classes.

Condition evaluator (see plugins/advanced-conditions):

```
@Component(service = ConditionEvaluator.class, property =
{"conditionEvaluatorId=myCustomConditionEvaluator"})
public class MyCustomConditionEvaluator implements ConditionEvaluator {
    // ...
}
```

Elasticsearch query builder (in your ES-specific bundle):

```
@Component(service = ConditionESQueryBuilder.class, property =
{"queryBuilderId=myCustomConditionQueryBuilder"})
public class MyCustomConditionESQueryBuilder implements ConditionESQueryBuilder {
    // ...
}
```

OpenSearch query builder (in your OS-specific bundle):

```
@Component(service = ConditionOSQueryBuilder.class, property =
{"queryBuilderId=myCustomConditionQueryBuilder"})
public class MyCustomConditionOSQueryBuilder implements ConditionOSQueryBuilder {
    // ...
}
```

Use the same queryBuilderId in JSON, Elasticsearch, and OpenSearch registrations. See Writing plugins for module layout and POM setup.

18.31.3. Catalog — base plugin

Condition type ID	Category	Parent	Key parameters / notes
booleanCondition	Event	—	Combines multiple conditions with a logical operator. Params: operator, subConditions
eventPropertyCondition	Event	—	Params: propertyName, comparisonOperator, propertyValue, propertyValueInteger, propertyValueDouble, propertyValueDate, propertyValueDateExpr, propertyValues
eventTypeCondition	Event	eventPropertyCondition	Params: eventTypeId
formEventCondition	Event	booleanCondition	Params: formId, pagePath
geoLocationByPointSessionCondition	Session	—	Params: type, rectLatitudeNE, rectLongitudeNE, rectLatitudeSW, rectLongitudeSW, circleLatitude, circleLongitude, distance
geoLocationSessionCondition	Session	booleanCondition	Params: country, admin1, admin2, city
goalMatchCondition	Profile	profilePropertyCondition	Params: goalId, comparisonOperator
idsCondition	Other	—	Params: ids, match
matchAllCondition	Event	—	Params: —
modifyAnyConsentEventCondition	Event	eventTypeCondition	Used to match any consent modification. Params: —
modifyConsentEventCondition	Event	booleanCondition	Used to match status modifications on a specific consent. Params: consentTypeId, consentStatus
nestedCondition	Session	—	Evaluates a condition on nested items within a profile or session property. Params: path, subCondition
newVisitorCondition	Session	booleanCondition	Params: since
notCondition	Event	—	Params: subCondition

Condition type ID	Category	Parent	Key parameters / notes
pastEventCondition	Profile	—	Evaluates past events that occurred within a specified time range. Params: eventCondition, operator, numberOfDays, fromDate, toDate, minimumEventCount, maximumEventCount
profileAliasesPropertyCondition	Other	—	Params: propertyName, comparisonOperator, propertyValue
profilePropertyCondition	Profile	—	Evaluates a profile property against various comparison criteria. Params: propertyName, comparisonOperator, propertyValue, propertyValueInteger, propertyValueDouble, propertyValueDate, propertyValueDateExpr, propertyValues
profileSegmentCondition	Profile	profilePropertyCondition	Params: segments, matchType
profileUpdatedEventCondition	Event	eventTypeCondition	Params: —
profileUserListCondition	Profile	profilePropertyCondition	Params: lists, matchType
returningVisitorCondition	Session	booleanCondition	Params: since
scoringCondition	Profile	profilePropertyCondition	Params: scoringPlanId, scoreValue, comparisonOperator
sessionCreatedEventCondition	Event	eventTypeCondition	Params: —
sessionDurationCondition	Session	sessionPropertyCondition	Params: minimumDuration, maximumDuration
sessionPropertyCondition	Session	—	Params: propertyName, comparisonOperator, propertyValue, propertyValueInteger, propertyValueDouble, propertyValueDate, propertyValueDateExpr, propertyValues

Condition type ID	Category	Parent	Key parameters / notes
topicPropertyCondition	Other	—	Params: propertyName, comparisonOperator, propertyValue
updatePropertiesEventCondition	Event	eventTypeCondition	Params: —
videoViewEventCondition	Event	booleanCondition	Params: videoId, pagePath

18.31.4. Catalog — extensions

Condition type ID	Source	Key parameters / notes
cdpSessionEventCondition	graphql	Params: —
hoverEventCondition	hover-event (not deployed by default)	Matches a hover on the given target id or page path. Params: targetId, targetPath (set at least one; with neither set, it matches nothing)
sourceEventPropertyCondition	advanced-conditions	Params: id, path, scope, type
updateConsentEventCondition	graphql	Params: —
updateListsEventCondition	graphql	Params: —
userListPropertyCondition	graphql	Params: propertyName, comparisonOperator, propertyValue, propertyValues

The hover-event plugin is built with Unomi but is not part of any Karaf feature, so hoverEventCondition is not available on a default install. Deploy the bundle to use it, for example from the Karaf console:

```
bundle:install -s mvn:org.apache.unomi/unomi-plugins-hover-event/${project.version}
```

18.31.5. Source locations

Condition descriptors are JSON files under META-INF/cxs/conditions/ in each plugin, for example:

- plugins/baseplugin/src/main/resources/META-INF/cxs/conditions/
- plugins/advanced-conditions/src/main/resources/META-INF/cxs/conditions/
- plugins/hover-event/src/main/resources/META-INF/cxs/conditions/

GraphQL-specific condition types ship with the GraphQL feature (graphql/cxs-impl/…).

Custom condition types can be added in your own plugins; see Writing plugins.

18.32. Built-in action types

Actions run when a rule matches. Each action type references an actionExecutor OSGi service.

18.32.1. Action type descriptor shape

```
{
 "metadata": {
  "id": "setPropertyAction",
  "name": "setPropertyAction",
  "systemTags": ["profileTags", "availableToEndUser"],
  "readOnly": true
 },
 "actionExecutor": "setProperty",
 "parameters": [
  { "id": "setPropertyName", "type": "string" },
  { "id": "setPropertyValue", "type": "string" }
 ]
}
```

18.32.2. OSGi registration example

```
<bean id="setPropertyActionImpl"
class="org.apache.unomi.plugins.baseplugin.actions.SetPropertyAction"/>
<service ref="setPropertyActionImpl" interface="org.apache.unomi.api.actions.ActionExecutor">
   <service-properties>
      <entry key="actionExecutorId" value="setProperty"/>
   </service-properties>
</service>
```

18.32.3. Catalog

Action type ID	Source	Executor	Parameters / description
addToListsAction	lists-extension	addToLists	Params: listIdentifiers
allEventToProfilePropertiesAction	baseplugin	allEventToProfileProperties	Params: —
cdpSessionEventAction	graphql	cdp_sessionEvent	Params: state, sessionId, scope
copyPropertiesAction	baseplugin	copyProperties	Params: setPropertyStrategy, rootProperty, mandatoryPropTypeSystemTag
evaluateProfileAgeAction	baseplugin	evaluateProfileAge	Params: —

Action type ID	Source	Executor	Parameters / description
evaluateProfileSegmentsAction	baseplugin	evaluateProfileSegments	Params: —
evaluateVisitPropertiesAction	baseplugin	evaluateVisitProperties	Params: —
eventToProfilePropertyAction	baseplugin	eventToProfileProperty	Params: eventPropertyName, profilePropertyName
incrementPropertyAction	baseplugin	incrementProperty	Params: propertyName, propertyTarget, storeInSession
mergeProfilesOnPropertyAction	baseplugin	mergeProfilesOnProperty	Params: mergeProfilePropertyName, forceEventProfileAsMaster
modifyConsentAction	baseplugin	modifyConsent	Modify a profile consent. Params: —
requestHeaderToProfilePropertyAction	request	requestHeaderToProfileProperty	Params: requestHeaderName, profilePropertyName, sessionPropertyName, storeAsList
requestParameterToProfilePropertyAction	request	requestParameterToProfileProperty	Params: requestParameterName, profilePropertyName, sessionPropertyName, storeAsList
sendEventAction	baseplugin	sendEvent	Params: eventType, eventProperties
sendMailAction	mail	sendMail	Params: notificationType, notificationTypeId, notifyOncePerProfile, from, to, cc, bcc, subject, template
setEventOccurenceCountAction	baseplugin	setEventOccurenceCount	Params: pastEventCondition
setPropertyAction	baseplugin	setProperty	Params: setPropertyName, setPropertyValue, setPropertyValueBoolean, setPropertyValueInteger, setPropertyValueMultiple, setPropertyValueCurrentEventTimestamp, setPropertyValueCurrentDate, setPropertyStrategy, storeInSession
setRemoteHostInfoAction	request	setRemoteHostInfo	Params: —
sfdcCreateOrUpdateLeadAction	salesforce-connector	sfdcCreateOrUpdateLead	Params: —
sfdcUpdateProfileFromLeadAction	salesforce-connector	sfdcUpdateProfileFromLead	Params: —

Action type ID	Source	Executor	Parameters / description
updateConsentEventAction	graphql	updateConsent	Params: type, status, lastUpdate, expiration
updateListsEventAction	graphql	updateLists	Params: join_lists, leave_lists
updatePropertiesAction	baseplugin	updateProperties	Update multiple properties in profile/persona. Params: —
weatherUpdateAction	weather-update	weatherUpdate	This action will retrieve the weather associated with the resolved location of the user (from his IP address). Params: —

18.32.4. Common action groups

Profile maintenance — setPropertyAction, updatePropertiesAction, incrementPropertyAction, copyPropertiesAction, eventToProfilePropertyAction, allEventToProfilePropertiesAction, mergeProfilesOnPropertyAction

Segmentation & scoring — evaluateProfileSegmentsAction, evaluateProfileAgeAction, evaluateVisitPropertiesAction

Events & consent — sendEventAction, modifyConsentAction, setEventOccurenceCountAction

HTTP request — requestHeaderToProfilePropertyAction, requestParameterToProfilePropertyAction, setRemoteHostInfoAction (request plugin)

Notifications — sendMailAction (mail plugin)

Integrations — addToListsAction, Salesforce actions, weatherUpdateAction, GraphQL CDP actions (when GraphQL feature is installed)

18.32.5. Source locations

- plugins/baseplugin/src/main/resources/META-INF/cxs/actions/
- plugins/mail/src/main/resources/META-INF/cxs/actions/
- plugins/request/src/main/resources/META-INF/cxs/actions/
- Extension plugins under extensions/*/src/main/resources/META-INF/cxs/actions/

List action types at runtime: GET /cxs/definitions/actions (tenant private key or admin).

See Writing plugins to implement custom actions.

18.33. Updating Events Using the Context Servlet

One of the use cases that needed to be supported by Unomi is the ability to build a user profile based on Internal System events or Change Data Capture which usally transported through internal messaging queues such as Kafka.

This can easily achieved using the KafkaInjector module built in within Unomi.

But, as streaming system usually operates in at-least-once semantics, we need to have a way to guarantee we wont have duplicate events in the system.

18.33.1. Authentication Requirements

To update events in Unomi, you must authenticate as either:

- A tenant administrator
- The system administrator

This authentication requirement ensures proper access control and security when modifying event data.

18.33.2. Solution

One of the solutions to this scenario is to have the ability to control and pass in the eventId property from outside of Unomi, Using an authorized 3rd party. This way whenever an event with the same itemId will be processed once again he wont be appended to list of events, but will be updated.

Here is an example of a request contains the itemdId

```
curl -X POST http://localhost:8181/cxs/context.json \
-H "Content-Type: application/json" \
--user "TENANT_ID:PRIVATE_KEY" \
-d @- <<'EOF'
{
   "events":[
      {
         "itemId": "exampleEventId",
         "eventType":"view",
         "scope": "example",
         "properties" : {
          "firstName" : "example"
         }
      }
   ]
}
EOF
```

Make sure to authenticate as either a tenant administrator or system administrator using Basic Authentication, and verify that the eventType is in the list of allowed events.

18.33.3. Defining Rules

Another use case we support is the ability to define a rule on the above mentioned events. If we have a rule that increment a property on profile level, we would want the action to be executed only once per event id. this can be achieved by adding "raiseEventOnlyOnce": false to the rule definition.

```
curl -X POST http://localhost:8181/cxs/context.json \
-H "Content-Type: application/json" \
-H "X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY" \
-d @- <<'EOF'
{
 "metadata": {
  "id": "updateNumberOfOrders",
  "name": "update number of orders on orderCreated eventType",
  "description": "update number of orders on orderCreated eventType"
 },
 "raiseEventOnlyOnce": false,
 "condition": {
  "type": "eventTypeCondition",
  "parameterValues": {
   "eventTypeId": "orderCreated"
  }
 },
 "actions": [
  {
   "parameterValues": {
    "setPropertyName": "properties.nbOfOrders",
    "setPropertyValue": "script::profile.properties.?nbOfOrders != null ?
(profile.properties.nbOfOrders + 1) : 1",
    "storeInSession": false
   },
   "type": "setPropertyAction"
  }
 ]
}
EOF
```

18.34. Unomi Web Tracker reference

In this section of the documentation, more details are provided about the web tracker provided by Unomi.

18.34.1. Configuring the tracker for your Unomi server

The same tracker works with Unomi 2.x, 3.x and 4.x. Only its configuration changes: on a multi-tenant Unomi 4 server, set apiKey in wemInitConfig to the tenant's public API key.

Unomi server	apiKey **in** wemInitConfig	**Data goes to**
2.x or 3.x	Leave it out	The server (no tenants)

Unomi server	apiKey in wemInitConfig	Data goes to
4.x in single-tenant compatibility mode	Leave it out (a key is ignored)	Tenant default
4.x multi-tenant	The tenant's **public** API key	That tenant

```
"wemInitConfig": {
  "contextServerUrl": "https://cdp.example.com",
  ...
  "apiKey": "YOUR_PUBLIC_API_KEY"
}
```

The tracker sends the key in the X-Unomi-Api-Key header on context.json and eventcollector requests. Without a key, a Unomi 4 server that is not in compatibility mode answers 401.

- Use the tenant's **public** key only, never its private key: the page configuration is visible to every visitor. The public key identifies the tenant; the tracker does not need a tenant ID.
- A custom header makes the browser send a CORS preflight request before each request when Unomi is on another origin. Unomi 2.x and 3.x servers accept the header and ignore it, so a page that sets a key still works with them.
- Create the tenant and obtain its plaintext keys with POST /cxs/tenants/{id}/apikeys before integrating a site. See Multi-tenancy.

apiKey and sendCredentials need version 2.0.0 or later of the tracker (the apache-unomi-tracker NPM package). Earlier versions do not send the key. Keep single-tenant compatibility mode on until those pages use tracker 2.0. See Upgrading the tracker from 1.x to 2.0.

18.34.2. Upgrading the tracker from 1.x to 2.0

Tracker 2.0 replaces tracker 1.x. It works with Unomi 2.x, 3.x and 4.x.

A page that already works keeps the same configuration, on the same Unomi server. Unomi 2.x, Unomi 3.x, and Unomi 4 in single-tenant compatibility mode need no new setting. A multi-tenant Unomi 4 server needs tracker 2.0.0 or later and apiKey, as described above. Tracker 1.x cannot send that key, so the server answers 401.

Where the script comes from matters:

- The copy Unomi serves at /tracker/unomi-web-tracker.js is updated when you upgrade the server. Unomi 4 serves tracker 2.0.
- A site that installed the apache-unomi-tracker package must upgrade that package itself (apache-unomi-tracker@2).
- A file copied from an older Unomi stays on tracker 1.x until you replace it. Leave compatibility mode on until you do.

What a site may notice comes from fixes. The configuration does not change:

- An empty page attributes object is no longer sent. Unomi 2.4 and later rejected those events, so page views and clicks from older examples are stored again. Attributes that contain values are still sent.
- Every click on a tracked element is sent, including a click on an image, an icon or some text inside it. Tracker 1.x sent only the first click on each element, and often nothing for a click on a child.
- The page's own click handlers run once per click. They ran twice.
- A link or a form still works when Unomi cannot be reached, when it never answers, and when tracking is turned off. A tracked form is submitted with the button that was used, including a button named submit.
- A missing, blank or zero timeoutInMilliseconds uses 1.5 seconds. With tracker 1.x every request failed at once.

apiKey and sendCredentials are new and optional. sendCredentials is described under Cookie/session handling.

The tracker project has the full list and the configuration for each Unomi version:

- Upgrading from 1.x
- Release notes

18.34.3. Custom events

In order to be able to use your own custom events with the web tracker, you must first declare them in Unomi so that they are properly recognized and validated by the /context.json or /eventcollector endpoints.

Declaring JSON schema

The first step is to declare a JSON schema for your custom event type. Here's an example of such a declaration:

```
{
  "$id": "https://unomi.apache.org/schemas/json/events/click/1-0-0",
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "self": {
    "vendor": "org.apache.unomi",
    "target": "events",
    "name": "click",
    "format": "jsonschema",
    "version": "1-0-0"
  },
  "title": "ClickEvent",
  "type": "object",
  "allOf": [
    {
      "$ref": "https://unomi.apache.org/schemas/json/event/1-0-0"
    }
  ],
  "properties": {
    "source": {
      "$ref": "https://unomi.apache.org/schemas/json/items/page/1-0-0"
    },
    "target": {
      "$ref": "https://unomi.apache.org/schemas/json/item/1-0-0"
    }
  },
  "unevaluatedProperties": false
}
```

The above example comes from a built-in event type that is already declared in Unomi but that illustrates the structure of a JSON schema. It is not however the objective of this section of the documentation to go into the details of how to declare a JSON schema, instead, we recommend you go to the corresponding section of the documentation.

Sending event from tracker

In the Unomi web tracker, you can use the following function to send an event to Unomi:

```
        /**
         * This function will send an event to Apache Unomi
         * @param {object} event The event object to send, you can build it using
wem.buildEvent(eventType, target, source)
         * @param {function} successCallback optional, will be executed in case of success
         * @param {function} errorCallback optional, will be executed in case of error
         * @return {undefined}
         */
        collectEvent: function (event, successCallback = undefined, errorCallback = undefined)
```

As you can see this function is quite straight forward to use. There are also helper functions to build events, such as :

```
        /**
         * This function return the basic structure for an event, it must be adapted to your need
         *
         * @param {string} eventType The name of your event
         * @param {object} [target] The target object for your event can be build with
wem.buildTarget(targetId, targetType, targetProperties)
         * @param {object} [source] The source object for your event can be build with
wem.buildSource(sourceId, sourceType, sourceProperties)
         * @returns {object} the event
         */
        buildEvent: function (eventType, target, source)

        /**
         * This function return an event of type form
         *
         * @param {string} formName The HTML name of id of the form to use in the target of the event
         * @param {HTMLFormElement} form optional HTML form element, if provided will be used to
extract the form fields and populate the form event
         * @returns {object} the form event
         */
        buildFormEvent: function (formName, form = undefined)

        /**
         * This function return the source object for a source of type page
         *
         * @returns {object} the target page
         */
        buildTargetPage: function ()

        /**
         * This function return the source object for a source of type page
         *
         * @returns {object} the source page
         */
        buildSourcePage: function ()

        /**
         * This function return the basic structure for the target of your event
         *
         * @param {string} targetId The ID of the target
         * @param {string} targetType The type of the target
         * @param {object} [targetProperties] The optional properties of the target
         * @returns {object} the target
         */
        buildTarget: function (targetId, targetType, targetProperties = undefined)

        /**
         * This function return the basic structure for the source of your event
         *
         * @param {string} sourceId The ID of the source
         * @param {string} sourceType The type of the source
         * @param {object} [sourceProperties] The optional properties of the source
         * @returns {object} the source
         */
        buildSource: function (sourceId, sourceType, sourceProperties = undefined)
```

Here's an example of using these helper functions and the collectEvent function alltogether:

```
var clickEvent = wem.buildEvent('click',
   wem.buildTarget('buttonId', 'button'),
   wem.buildSourcePage());

wem.collectEvent(clickEvent, function (xhr) {
   console.info('Click event successfully collected.');
}, function (xhr) {
   console.error('Could not send click event.');
});
```

Sending multiple events

In some cases, especially when multiple events must be sent fast and the order of the events is critical for rules to be properly executed, it is better to use another function called collectEvents that will batch the sending of events to Unomi in a single HTTP request. Here's the signature of this function:

```
/**
 * This function will send the events to Apache Unomi
 *
 * @param {object} events Javascript object { events: [event1, event2] }
 * @param {function} successCallback optional, will be executed in case of success
 * @param {function} errorCallback optional, will be executed in case of error
 * @return {undefined}
 */
collectEvents: function (events, successCallback = undefined, errorCallback = undefined)
```

This function is almost exactly the same as the collectEvent method except that it takes an events array instead of a single one. The events in the array may be of any mixture of types.

Extending existing events

An alternative to defining custom event types is to extend existing event types. This, for example, can be used to add new properties to the built-in view event type.

For more information about event type extensions, please read the JSON schema section of this documentation.

18.34.4. Integrating with tag managers

Integrating with tag managers such as Google Tag Manager is an important part of the way trackers can be added to an existing site. Unomi's web tracker should be pretty easy to integrate with such tools: you simply need to insert the script tag to load the script and then another tag to initialize it and map any tag manager variables you want.

Personalization scripts should however be modified to check for the existence of the tracker object in the page because tag managers might deactivate scripts based on conditions such as GDPR approval, cookie preferences, ...

18.34.5. Cookie/session handling

In order to track profiles, an identifier must be stored in the browser so that subsequent requests can keep a reference to the visitor's profile. Also, a session identifier must be generated to group the current visitor interactions.

Unomi's web tracker uses 3 cookies in the tracker by default:

- server profile ID, called context-profile-id by default, that is sent from the Unomi server
- web tracker profile ID, called wem-profile-id by default (this is a copy of the server profile ID that can be managed by the tracker directly)
- web tracker session ID, called wem-session-id by default

It is possible to change the name of these cookie by passing the following properties to the start's initialization:

```
"wemInitConfig": {
   ...
   "contextServerCookieName": "context-profile-id",
   "trackerSessionIdCookieName": "unomi-tracker-test-session-id",
   "trackerProfileIdCookieName": "unomi-tracker-test-profile-id"
}
```

Please note however that the contextServerCookieName will also have to be changed in the server configuration in order for it to work. See the Configuration section for details on how to do this.

The tracker sends requests with cookies. When the website and Unomi are on different origins, another subdomain included (www.example.com and cdp.example.com), list the website origin on the server (UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS) or in the tenant's publicCorsAllowedOrigins property; otherwise the browser blocks the responses and Unomi does not use the visitor cookie. A website served from the same origin as Unomi, through a reverse proxy, needs no list. A website on another site also needs HTTPS and SameSite=None; see Websites on other origins. In single-tenant compatibility mode, embedding context.js with a <script> tag from another origin is served without the visitor cookie (a new anonymous profile on every page view) unless the operator turns on org.apache.unomi.public.cors.matchReferer; see Embedding context.js with a script tag. If the deployment does not need the visitor cookie, set sendCredentials to false in wemInitConfig (tracker 2.0.0 or later). Requests are then sent without cookies and work with the default server settings, but every request starts a new anonymous profile. Unomi 2.x and 3.x servers instead find the profile through the session, so it lasts for the session.

For session tracking, it is important to provide a value for the cookie otherwise the tracker will not initialize (a message is displayed in the console that explains that the session cookie is missing). Here is the code excerpt from the initialization code used in the tutorial that creates the initial cookie value.

```
    // generate a new session
    if
(unomiWebTracker.getCookie(unomiTrackerTestConf.wemInitConfig.trackerSessionIdCookieName)
== null) {

unomiWebTracker.setCookie(unomiTrackerTestConf.wemInitConfig.trackerSessionIdCookieName,
unomiWebTracker.generateGuid(), 1);
    }
```

Note that this is just an example, you could very well customize this code to create session IDs another way.

18.34.6. JavaScript API

The JavaScript API for the web tracker is in the tracker source: https://github.com/apache/unomi-tracker/blob/main/src/apache-unomi-tracker.js

How to install it, how to configure it for each Unomi version, and what changed in 2.0 are in the tracker project: README. The short version for an existing site is Upgrading the tracker from 1.x to 2.0 above.

Please note that only the functions that do NOT start with an underscore should be used. The ones that start with an underscore are not considered part of the public API and could change or even be removed at any point in the future.

18.35. Implementing a JavaScript Tracker

This guide explains how to implement a basic JavaScript tracker for Apache Unomi from scratch. This will help you understand the underlying concepts and API interactions, whether you're building a custom tracker or integrating Apache Unomi into an existing application.



Apache Unomi provides an official web tracker (see Unomi Web Tracker reference) that handles many of these details automatically, including the API key (see Configuring the tracker for your Unomi server). This guide is useful if you need a custom implementation or want to understand how the tracker works internally.

18.35.1. Prerequisites

Before implementing a tracker, you need:

- An Apache Unomi instance running and accessible
- A tenant created with a public API key (see Request examples for tenant creation)
- Basic knowledge of JavaScript and HTTP requests
- Understanding of how profile tracking works (see How profile tracking works)

18.35.2. Basic Tracker Structure

A basic JavaScript tracker needs to:

1. **Manage session and profile identifiers** - Generate and store session IDs, handle profile ID cookies
2. **Make context requests** - Call /cxs/context.json to retrieve profile/session data
3. **Send events** - Send user interaction events to Apache Unomi
4. **Handle personalization** - Request and apply personalized content
5. **Manage cookies** - Read and write cookies for profile tracking

Here's a minimal tracker structure:

```
(function() {
    'use strict';

    // Configuration
    const CONFIG = {
        contextServerUrl: 'http://localhost:8181',
        apiKey: 'YOUR_PUBLIC_API_KEY',
        scope: 'mydigital',
        sessionCookieName: 'unomi-session-id'
        // No profile cookie name here: the profile ID cookie is HttpOnly and
        // cannot be read from JavaScript. Use response.profileId instead.
    };

    // Tracker object
    const tracker = {
        // Session management
        getSessionId: function() { /* ... */ },
        setSessionId: function(sessionId) { /* ... */ },

        // Cookie utilities
        getCookie: function(name) { /* ... */ },
        setCookie: function(name, value, days) { /* ... */ },

        // Context requests
        getContext: function(options, callback) { /* ... */ },

        // Event tracking
        trackEvent: function(event, callback) { /* ... */ },
        trackEvents: function(events, callback) { /* ... */ },

        // Personalization
        personalize: function(personalizations, callback) { /* ... */ },

        // Initialization
        init: function() { /* ... */ }
    };

    // Expose tracker globally
    window.unomiTracker = tracker;
})();
```

18.35.3. Cookie Management

The tracker needs to read and write cookies to maintain the session ID and handle the profile ID cookie set by Apache Unomi.

```
// Cookie utilities
getCookie: function(name) {
    const value = `; ${document.cookie}`;
    const parts = value.split(`; ${name}=`);
    if (parts.length === 2) {
        return parts.pop().split(';').shift();
    }
    return null;
},

setCookie: function(name, value, days) {
    const expires = new Date();
    expires.setTime(expires.getTime() + (days * 24 * 60 * 60 * 1000));
    document.cookie = `${name}=${value};expires=${expires.toUTCString()};path=/`;
},

deleteCookie: function(name) {
    document.cookie = `${name}=;expires=Thu, 01 Jan 1970 00:00:00 UTC;path=/;`;
}
```

18.35.4. Session Management

The tracker needs to generate and maintain a session ID. If no session ID exists, generate a new one:

```
// Generate a UUID v4
// Uses crypto.randomUUID() if available (most secure, modern browsers)
// Falls back to crypto.getRandomValues() (secure, widely supported)
// Fails closed if neither is available: session IDs must never be guessable
generateUUID: function() {
    // Use crypto.randomUUID() if available (Chrome 92+, Firefox 95+, Safari 15.4+, Node.js 19+)
    if (typeof crypto !== 'undefined' && crypto.randomUUID) {
        return crypto.randomUUID();
    }

    // Use crypto.getRandomValues() if available (widely supported, secure)
    if (typeof crypto !== 'undefined' && crypto.getRandomValues) {
        return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
            const r = crypto.getRandomValues(new Uint8Array(1))[0] % 16;
            const v = c === 'x' ? r : (r & 0x3 | 0x8);
            return v.toString(16);
        });
    }

    // No cryptographically secure source available: fail closed.
    // Do NOT fall back to Math.random() — its output is predictable and a
    // guessable session ID lets an untrusted caller take over a visitor's session.
    throw new Error(
        'Unomi tracker: no cryptographically secure random source available ' +
        '(neither crypto.randomUUID nor crypto.getRandomValues). This browser is ' +
        'unsupported: install a Web Crypto polyfill, or use a browser that provides ' +
        'crypto.getRandomValues.'
    );
},

// Get or create session ID
getSessionId: function() {
    let sessionId = this.getCookie(CONFIG.sessionCookieName);
    if (!sessionId) {
        sessionId = this.generateUUID();
        this.setSessionId(sessionId);
    }
    return sessionId;
},

setSessionId: function(sessionId) {
    this.setCookie(CONFIG.sessionCookieName, sessionId, 1); // 1 day expiration
}
```

18.35.5. Making Context Requests

The core of the tracker is making requests to /cxs/context.json. This endpoint:

- Automatically creates or loads profiles and sessions
- Processes events
- Resolves personalization

- Returns profile and session data

```
// Make a context request
getContext: function(options, callback) {
    const sessionId = this.getSessionId();
    const url =
`${CONFIG.contextServerUrl}/cxs/context.json?sessionId=${encodeURIComponent(sessionId)}`;

    // Build request payload
    const payload = {
        source: {
            itemId: options.sourceId || window.location.pathname,
            itemType: options.sourceType || 'page',
            scope: CONFIG.scope
        }
    };

    // Add optional parameters
    if (options.requiredProfileProperties) {
        payload.requiredProfileProperties = options.requiredProfileProperties;
    }
    if (options.requiredSessionProperties) {
        payload.requiredSessionProperties = options.requiredSessionProperties;
    }
    if (options.requireSegments) {
        payload.requireSegments = true;
    }
    if (options.requireScores) {
        payload.requireScores = true;
    }
    if (options.events) {
        payload.events = options.events;
    }
    if (options.filters) {
        payload.filters = options.filters;
    }
    if (options.personalizations) {
        payload.personalizations = options.personalizations;
    }

    // Make the request
    const xhr = new XMLHttpRequest();
    xhr.open('POST', url, true);
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.setRequestHeader('X-Unomi-Api-Key', CONFIG.apiKey);
    xhr.withCredentials = true; // Important for cookies

    xhr.onreadystatechange = function() {
        if (xhr.readyState === 4) {
            if (xhr.status === 200) {
                try {
                    const response = JSON.parse(xhr.responseText);
                    if (callback) {
                        callback(null, response);
                    }
                } catch (e) {
```

```
                if (callback) {
                    callback(new Error('Failed to parse response: ' + e.message), null);
                }
            }
        } else {
            if (callback) {
                callback(new Error('Request failed with status: ' + xhr.status), null);
            }
        }
    }
};

xhr.send(JSON.stringify(payload));
}
```

18.35.6. Retrieving Profile and Session Properties

To retrieve profile and session properties, include them in the context request:

```
// Get all profile and session properties
tracker.getContext({
    requireSegments: true,
    requireScores: true,
    requiredProfileProperties: ['*'], // '*' means all properties
    requiredSessionProperties: ['*']
}, function(error, response) {
    if (error) {
        console.error('Failed to get context:', error);
        return;
    }

    console.log('Profile ID:', response.profileId);
    console.log('Session ID:', response.sessionId);
    console.log('Profile Properties:', response.profileProperties);
    console.log('Session Properties:', response.sessionProperties);
    console.log('Profile Segments:', response.profileSegments);
    console.log('Profile Scores:', response.profileScores);
});
```

You can also request specific properties:

```
// Get specific profile properties
tracker.getContext({
    requiredProfileProperties: ['firstName', 'lastName', 'email', 'pageViewCount']
}, function(error, response) {
    if (error) {
        console.error('Failed to get context:', error);
        return;
    }

    const profile = response.profileProperties || {};
    console.log('User:', profile.firstName, profile.lastName);
    console.log('Email:', profile.email);
    console.log('Page Views:', profile.pageViewCount);
});
```

18.35.7. Sending Events

Events are sent as part of the context request. You can send a single event or multiple events in one request:

```
// Build an event object
buildEvent: function(eventType, target, source) {
    const event = {
        eventType: eventType,
        scope: CONFIG.scope
    };

    // Set source if provided, otherwise use default based on event type
    if (source) {
        event.source = source;
    } else if (eventType === 'view') {
        // For view events, source should typically be a site
        event.source = {
            itemType: 'site',
            scope: CONFIG.scope,
            itemId: window.location.hostname || 'default-site'
        };
    } else {
        // For other events, source defaults to the current page
        event.source = {
            itemType: 'page',
            scope: CONFIG.scope,
            itemId: window.location.pathname
        };
    }

    // Set target if provided
    if (target) {
        event.target = target;
    }

    return event;
},

// Track a single event
trackEvent: function(event, callback) {
    this.getContext({
        events: [event]
    }, callback);
},

// Track multiple events (more efficient)
trackEvents: function(events, callback) {
    this.getContext({
        events: events
    }, callback);
}
```

Example: Tracking a page view

```
// Track a page view event
const viewEvent = tracker.buildEvent('view', {
    itemType: 'page',
    scope: CONFIG.scope,
    itemId: window.location.pathname,
    properties: {
        pageInfo: {
            pageName: document.title,
            destinationURL: window.location.href,
            referringURL: document.referrer
        }
    }
}, {
    itemType: 'site',
    scope: CONFIG.scope,
    itemId: window.location.hostname || 'default-site'
});

tracker.trackEvent(viewEvent, function(error, response) {
    if (error) {
        console.error('Failed to track view event:', error);
        return;
    }
    console.log('View event tracked. Processed events:', response.processedEvents);
});
```

Example: Tracking a click event

```
// Track a click on a button
function trackButtonClick(buttonId, buttonText) {
    const clickEvent = tracker.buildEvent('click', {
        itemType: 'button',
        scope: CONFIG.scope,
        itemId: buttonId,
        properties: {
            text: buttonText
        }
    });

    tracker.trackEvent(clickEvent, function(error, response) {
        if (error) {
            console.error('Failed to track click:', error);
            return;
        }
        console.log('Click event tracked');
    });
}

// Use it
document.getElementById('myButton').addEventListener('click', function() {
    trackButtonClick('myButton', 'Click Me');
});
```

Example: Tracking a form submission

```
// Track a form submission
function trackFormSubmission(formId, formData) {
    const formEvent = {
        eventType: 'form',
        scope: CONFIG.scope,
        source: {
            itemType: 'page',
            scope: CONFIG.scope,
            itemId: window.location.pathname
        },
        target: {
            itemType: 'form',
            scope: CONFIG.scope,
            itemId: formId,
            properties: formData
        }
    };

    tracker.trackEvent(formEvent, function(error, response) {
        if (error) {
            console.error('Failed to track form:', error);
            return;
        }
        console.log('Form event tracked');
    });
}

// Use it
document.getElementById('myForm').addEventListener('submit', function(e) {
    e.preventDefault();
    const formData = {
        formName: this.id,
        formFields: {}
    };

    // Extract form field values
    Array.from(this.elements).forEach(function(element) {
        if (element.name) {
            formData.formFields[element.name] = element.value;
        }
    });

    trackFormSubmission(this.id, formData);
});
```

18.35.8. Using Filters

Filters allow you to evaluate conditions against the profile and session to determine which content should be displayed. Filters return boolean results indicating whether conditions match.

```
// Request filter evaluation
tracker.getContext({
    filters: [
        {
            id: 'premium-user-filter',
            filters: [
                {
                    condition: {
                        type: 'profileSegmentCondition',
                        parameterValues: {
                            segments: ['premium-users']
                        }
                    }
                }
            ]
        },
        {
            id: 'new-visitor-filter',
            filters: [
                {
                    condition: {
                        type: 'profilePropertyCondition',
                        parameterValues: {
                            propertyName: 'properties.firstVisit',
                            comparisonOperator: 'exists'
                        }
                    }
                }
            ]
        }
    ]
}, function(error, response) {
    if (error) {
        console.error('Failed to evaluate filters:', error);
        return;
    }

    const results = response.filteringResults || {};
    if (results['premium-user-filter']) {
        console.log('User is a premium user');
        // Show premium content
    }
    if (results['new-visitor-filter']) {
        console.log('User is a new visitor');
        // Show welcome message
    }
});
```

18.35.9. Using Personalization

Personalization allows you to request content variants and have Apache Unomi select the most appropriate one based on the profile and session state.

```
// Request personalization
```

```
tracker.getContext({
    personalizations: [
        {
            id: 'homepage-hero',
            strategy: 'matching-first',
            strategyOptions: {
                fallback: 'default-hero'
            },
            contents: [
                {
                    id: 'premium-user-hero',
                    filters: [
                        {
                            condition: {
                                type: 'profileSegmentCondition',
                                parameterValues: {
                                    segments: ['premium-users']
                                }
                            }
                        }
                    ]
                },
                {
                    id: 'new-visitor-hero',
                    filters: [
                        {
                            condition: {
                                type: 'profilePropertyCondition',
                                parameterValues: {
                                    propertyName: 'properties.firstVisit',
                                    comparisonOperator: 'exists'
                                }
                            }
                        }
                    ]
                },
                {
                    id: 'default-hero'
                }
            ]
        }
    ]
}, function(error, response) {
    if (error) {
        console.error('Failed to get personalization:', error);
        return;
    }

    // Get selected content ID(s)
    // Note: personalizations is deprecated but still available
    // It returns a List<String>, so get the first element for single-selection strategies
    const contentIds = response.personalizations['homepage-hero'];
    const selectedContentId = contentIds && contentIds.length > 0 ? contentIds[0] : null;

    // Alternatively, use personalizationResults for more detailed information
    const personalizationResult = response.personalizationResults['homepage-hero'];
    if (personalizationResult && personalizationResult.contentIds &&
```

```
personalizationResult.contentIds.length > 0) {
        const selectedId = personalizationResult.contentIds[0];
        console.log('Selected content:', selectedId);
        console.log('Strategy used:', personalizationResult.strategy);
        applyPersonalizedContent('homepage-hero', selectedId);
    } else if (selectedContentId) {
        console.log('Selected content:', selectedContentId);
        applyPersonalizedContent('homepage-hero', selectedContentId);
    } else {
        console.log('No content selected, using fallback');
        applyPersonalizedContent('homepage-hero', 'default-hero');
    }
});

function applyPersonalizedContent(personalizationId, contentId) {
    // Your logic to apply the personalized content
    const element = document.getElementById(personalizationId);
    if (element) {
        // Example: Update content based on selected ID
        if (contentId === 'premium-user-hero') {
            element.innerHTML = '<h1>Welcome, Premium Member!</h1>';
        } else if (contentId === 'new-visitor-hero') {
            element.innerHTML = '<h1>Welcome! Sign up today!</h1>';
        } else {
            element.innerHTML = '<h1>Welcome!</h1>';
        }
    }
}
```

18.35.10. Complete Example: Page Load with Personalization

Here's a complete example that loads context, tracks a page view, and applies personalization on page load:

```
// Initialize tracker on page load
tracker.init = function() {
    const sessionId = this.getSessionId();

    // Build page view event
    const viewEvent = this.buildEvent('view', {
        itemType: 'page',
        scope: CONFIG.scope,
        itemId: window.location.pathname,
        properties: {
            pageInfo: {
                pageName: document.title,
                destinationURL: window.location.href,
                referringURL: document.referrer
            }
        }
    });

    // Request context with event and personalization
    this.getContext({
```

```
    events: [viewEvent],
    requireSegments: true,
    requiredProfileProperties: ['firstName', 'lastName', 'pageViewCount'],
    personalizations: [
      {
        id: 'homepage-hero',
        strategy: 'matching-first',
        strategyOptions: {
          fallback: 'default-hero'
        },
        contents: [
          {
            id: 'premium-user-hero',
            filters: [
              {
                condition: {
                  type: 'profileSegmentCondition',
                  parameterValues: {
                    segments: ['premium-users']
                  }
                }
              }
            ]
          },
          {
            id: 'new-visitor-hero',
            filters: [
              {
                condition: {
                  type: 'profilePropertyCondition',
                  parameterValues: {
                    propertyName: 'properties.firstVisit',
                    comparisonOperator: 'exists'
                  }
                }
              }
            ]
          },
          {
            id: 'default-hero'
          }
        ]
      }
    ]
  }, function(error, response) {
    if (error) {
      console.error('Failed to initialize tracker:', error);
      return;
    }

    // Log profile information
    console.log('Profile ID:', response.profileId);
    console.log('Session ID:', response.sessionId);
    console.log('Profile Segments:', response.profileSegments);

    // Apply personalization
    // Get the first content ID from the list (for single-selection strategies like 'matching-first')
```

```
        // Note: personalizations returns a List<String>, so get the first element
        const contentIds = response.personalizations && response.personalizations['homepage-hero'];
        const selectedContentId = contentIds && contentIds.length > 0 ? contentIds[0] : 'default-hero';
        applyPersonalizedContent('homepage-hero', selectedContentId);

        // Update UI with profile data
        if (response.profileProperties) {
            const profile = response.profileProperties;
            if (profile.firstName) {
                updateWelcomeMessage(profile.firstName, profile.lastName);
            }
            if (profile.pageViewCount) {
                updatePageViewCount(profile.pageViewCount);
            }
        }
    });
};

function applyPersonalizedContent(personalizationId, contentId) {
    const element = document.getElementById(personalizationId);
    if (!element) return;

    const contentMap = {
        'premium-user-hero': '<div class="hero premium"><h1>Welcome, Premium
Member!</h1><p>Thank you for your loyalty.</p></div>',
        'new-visitor-hero': '<div class="hero new-visitor"><h1>Welcome!</h1><p>Sign up today and get
10% off your first purchase.</p></div>',
        'default-hero': '<div class="hero default"><h1>Welcome!</h1><p>Discover our amazing
products.</p></div>'
    };

    element.innerHTML = contentMap[contentId] || contentMap['default-hero'];
}

function updateWelcomeMessage(firstName, lastName) {
    const element = document.getElementById('welcome-message');
    if (element) {
        element.textContent = `Welcome back, ${firstName} ${lastName}!`;
    }
}

function updatePageViewCount(count) {
    const element = document.getElementById('page-view-count');
    if (element) {
        element.textContent = `You've viewed ${count} pages`;
    }
}

// Initialize when DOM is ready
if (document.readyState === 'loading') {
    document.addEventListener('DOMContentLoaded', function() {
        tracker.init();
    });
} else {
    tracker.init();
}
```

18.35.11. Advanced Features

Batch Event Tracking

For better performance, batch multiple events into a single request:

```
// Track multiple events in one request
const events = [
    tracker.buildEvent('view', { /* ... */ }),
    tracker.buildEvent('click', { /* ... */ }),
    tracker.buildEvent('form', { /* ... */ })
];

tracker.trackEvents(events, function(error, response) {
    if (error) {
        console.error('Failed to track events:', error);
        return;
    }
    console.log('Tracked', response.processedEvents, 'events');
});
```

Event Queue

Implement an event queue to batch events and send them periodically or when a threshold is reached:

```
// Event queue implementation
const eventQueue = {
    queue: [],
    maxSize: 10,
    flushInterval: 5000, // 5 seconds

    add: function(event) {
        this.queue.push(event);
        if (this.queue.length >= this.maxSize) {
            this.flush();
        }
    },

    flush: function() {
        if (this.queue.length === 0) return;

        const events = this.queue.slice();
        this.queue = [];

        tracker.trackEvents(events, function(error, response) {
            if (error) {
                console.error('Failed to flush events:', error);
                // Re-add events to queue on error
                eventQueue.queue = events.concat(eventQueue.queue);
            } else {
                console.log('Flushed', response.processedEvents, 'events');
            }
        });
    },

    start: function() {
        setInterval(function() {
            eventQueue.flush();
        }, this.flushInterval);
    }
};

// Start automatic flushing
eventQueue.start();

// Add events to queue
eventQueue.add(tracker.buildEvent('click', { /* ... */ }));
```

Error Handling and Retry Logic

Implement robust error handling with retry logic:

```
// Enhanced context request with retry logic
getContextWithRetry: function(options, callback, maxRetries) {
    maxRetries = maxRetries || 3;
    let retries = 0;

    const attempt = () => {
        this.getContext(options, function(error, response) {
            if (error && retries < maxRetries) {
                retries++;
                console.warn('Request failed, retrying...', retries);
                setTimeout(attempt, 1000 * retries); // Exponential backoff
            } else {
                callback(error, response);
            }
        });
    };

    attempt();
}
```

Profile ID Synchronization

The profile ID is set by Apache Unomi in a cookie. By default the cookie is HttpOnly, so browser JavaScript **cannot** read it via document.cookie. Always prefer the profileId field in the JSON response. The browser still **sends** the cookie on subsequent requests when credentials/cookies are included.

```
// Enhanced context request that handles profile ID from the response
getContext: function(options, callback) {
    // ... existing code ...

    xhr.onreadystatechange = function() {
        if (xhr.readyState === 4) {
            if (xhr.status === 200) {
                try {
                    const response = JSON.parse(xhr.responseText);

                    // Authoritative profile id for application logic:
                    // response.profileId (cookie may be HttpOnly and unreadable from JS)
                    if (response.profileId) {
                        // store in memory / your app state if needed — do not rely on document.cookie
                    }

                    if (callback) {
                        callback(null, response);
                    }
                } catch (e) {
                    if (callback) {
                        callback(new Error('Failed to parse response: ' + e.message), null);
                    }
                }
            } else {
                if (callback) {
                    callback(new Error('Request failed with status: ' + xhr.status), null);
                }
            }
        }
    };

    // ... rest of existing code ...
}
```

Consent Management

Handle user consent for tracking:

```
// Check if user has consented to tracking
hasConsent: function(consentType) {
    // Check localStorage or cookie for consent
    const consent = localStorage.getItem('unomi-consent-' + consentType);
    return consent === 'true';
},

// Request context only if consent is given
getContextWithConsent: function(options, callback) {
    if (!this.hasConsent('tracking')) {
        console.warn('User has not consented to tracking');
        if (callback) {
            callback(new Error('Consent required'), null);
        }
        return;
    }

    this.getContext(options, callback);
}
```

Cross-Domain Tracking

The visitor cookie is set by Unomi, not by page script: it is HttpOnly by default and belongs to Unomi's host (or the parent domain set in org.apache.unomi.profile.cookie.domain). Page code cannot create or move it to the website's domain, and a body profileId is ignored for public callers.

When the website and Unomi are on different origins:

- send requests with credentials (xhr.withCredentials = true or fetch(..., {credentials: 'include'}));
- have the website origin listed, by the operator in UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS or in the tenant's publicCorsAllowedOrigins property;
- for a website on another site (another registrable domain), also serve Unomi over HTTPS with SameSite=None, knowing that browsers increasingly block such third-party cookies.

Serving Unomi from the website's own origin through a reverse proxy avoids all of this. See Websites on other origins.

Performance Optimization

Optimize performance by:

- **Debouncing events**: Don't send every single interaction immediately
- **Batching requests**: Combine multiple operations into one request
- **Caching context**: Cache profile data and only refresh when needed

```
// Debounce function
debounce: function(func, wait) {
    let timeout;
    return function executedFunction(...args) {
        const later = () => {
            clearTimeout(timeout);
            func(...args);
        };
        clearTimeout(timeout);
        timeout = setTimeout(later, wait);
    };
},

// Debounced event tracking
trackEventDebounced: this.debounce(function(event) {
    tracker.trackEvent(event, function(error, response) {
        if (error) {
            console.error('Failed to track event:', error);
        }
    });
}, 1000), // Wait 1 second before sending

// Context caching
contextCache: {
    data: null,
    timestamp: null,
    ttl: 60000, // 1 minute

    get: function() {
        if (this.data && this.timestamp && (Date.now() - this.timestamp) < this.ttl) {
            return this.data;
        }
        return null;
    },

    set: function(data) {
        this.data = data;
        this.timestamp = Date.now();
    },

    clear: function() {
        this.data = null;
        this.timestamp = null;
    }
}
```

18.35.12. Complete Tracker Implementation

Here's a complete, production-ready tracker implementation combining all the concepts:

```
(function() {
    'use strict';

    const CONFIG = {
```

```
    contextServerUrl: 'http://localhost:8181',
    apiKey: 'YOUR_PUBLIC_API_KEY',
    scope: 'mydigital',
    sessionCookieName: 'unomi-session-id',
    // No profile cookie name here: the profile ID cookie is HttpOnly and
    // cannot be read from JavaScript. Use response.profileId instead.
    eventQueueSize: 10,
    eventQueueInterval: 5000
};

const tracker = {
    // Cookie management
    getCookie: function(name) {
        const value = `; ${document.cookie}`;
        const parts = value.split(`; ${name}=`);
        if (parts.length === 2) {
            return parts.pop().split(';').shift();
        }
        return null;
    },

    setCookie: function(name, value, days, domain) {
        const expires = new Date();
        expires.setTime(expires.getTime() + (days * 24 * 60 * 60 * 1000));
        let cookie = `${name}=${value};expires=${expires.toUTCString()};path=/`;
        if (domain) {
            cookie += `;domain=${domain}`;
        }
        document.cookie = cookie;
    },

    // UUID generation
    // Uses crypto.randomUUID() if available (most secure, modern browsers)
    // Falls back to crypto.getRandomValues() (secure, widely supported)
    // Fails closed if neither is available: session IDs must never be guessable
    generateUUID: function() {
        // Use crypto.randomUUID() if available (Chrome 92+, Firefox 95+, Safari 15.4+, Node.js 19+)
        if (typeof crypto !== 'undefined' && crypto.randomUUID) {
            return crypto.randomUUID();
        }

        // Use crypto.getRandomValues() if available (widely supported, secure)
        if (typeof crypto !== 'undefined' && crypto.getRandomValues) {
            return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
                const r = crypto.getRandomValues(new Uint8Array(1))[0] % 16;
                const v = c === 'x' ? r : (r & 0x3 | 0x8);
                return v.toString(16);
            });
        }

        // No cryptographically secure source available: fail closed.
        // Do NOT fall back to Math.random() — its output is predictable and a
        // guessable session ID lets an untrusted caller take over a visitor's session.
        throw new Error(
            'Unomi tracker: no cryptographically secure random source available ' +
            '(neither crypto.randomUUID nor crypto.getRandomValues). This browser is ' +
            'unsupported: install a Web Crypto polyfill, or use a browser that provides ' +
```

```
            'crypto.getRandomValues.'
        );
    },

    // Session management
    getSessionId: function() {
        let sessionId = this.getCookie(CONFIG.sessionCookieName);
        if (!sessionId) {
            sessionId = this.generateUUID();
            this.setSessionId(sessionId);
        }
        return sessionId;
    },

    setSessionId: function(sessionId) {
        this.setCookie(CONFIG.sessionCookieName, sessionId, 1);
    },

    // Event building
    buildEvent: function(eventType, target, source) {
        return {
            eventType: eventType,
            scope: CONFIG.scope,
            source: source || {
                itemType: 'page',
                scope: CONFIG.scope,
                itemId: window.location.pathname
            },
            target: target
        };
    },

    // Context request
    getContext: function(options, callback) {
        const sessionId = this.getSessionId();
        const url =
`${CONFIG.contextServerUrl}/cxs/context.json?sessionId=${encodeURIComponent(sessionId)}`;

        const payload = {
            source: {
                itemId: options.sourceId || window.location.pathname,
                itemType: options.sourceType || 'page',
                scope: CONFIG.scope
            }
        };

        if (options.requiredProfileProperties) {
            payload.requiredProfileProperties = options.requiredProfileProperties;
        }
        if (options.requiredSessionProperties) {
            payload.requiredSessionProperties = options.requiredSessionProperties;
        }
        if (options.requireSegments) {
            payload.requireSegments = true;
        }
        if (options.requireScores) {
            payload.requireScores = true;
```

```
    }
    if (options.events) {
        payload.events = options.events;
    }
    if (options.filters) {
        payload.filters = options.filters;
    }
    if (options.personalizations) {
        payload.personalizations = options.personalizations;
    }

    const xhr = new XMLHttpRequest();
    xhr.open('POST', url, true);
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.setRequestHeader('X-Unomi-Api-Key', CONFIG.apiKey);
    xhr.withCredentials = true;

    xhr.onreadystatechange = function() {
        if (xhr.readyState === 4) {
            if (xhr.status === 200) {
                try {
                    const response = JSON.parse(xhr.responseText);
                    if (callback) {
                        callback(null, response);
                    }
                } catch (e) {
                    if (callback) {
                        callback(new Error('Failed to parse response: ' + e.message), null);
                    }
                }
            } else {
                if (callback) {
                    callback(new Error('Request failed with status: ' + xhr.status), null);
                }
            }
        }
    };

    xhr.send(JSON.stringify(payload));
},

// Event tracking
trackEvent: function(event, callback) {
    this.getContext({
        events: [event]
    }, callback);
},

trackEvents: function(events, callback) {
    this.getContext({
        events: events
    }, callback);
},

// Initialization
init: function(config) {
    // Merge custom config
```

```
        if (config) {
            Object.assign(CONFIG, config);
        }

        // Initialize session
        this.getSessionId();

        // Track page view
        const viewEvent = this.buildEvent('view', {
            itemType: 'page',
            scope: CONFIG.scope,
            itemId: window.location.pathname,
            properties: {
                pageInfo: {
                    pageName: document.title,
                    destinationURL: window.location.href,
                    referringURL: document.referrer
                }
            }
        });

        this.trackEvent(viewEvent, function(error, response) {
            if (error) {
                console.error('Failed to track page view:', error);
            } else {
                console.log('Page view tracked');
            }
        });
    }
};

// Expose tracker
window.unomiTracker = tracker;

// Auto-initialize if DOM is ready
if (document.readyState === 'loading') {
    document.addEventListener('DOMContentLoaded', function() {
        tracker.init();
    });
} else {
    tracker.init();
}
})();
```

18.35.13. Best Practices

1. **Always use HTTPS in production** - Update contextServerUrl to use HTTPS
2. **Handle errors gracefully** - Don't break the user experience if tracking fails
3. **Respect user privacy** - Implement consent management
4. **Optimize performance** - Batch events, cache context, debounce rapid interactions
5. **Test thoroughly** - Test with different browsers, network conditions, and privacy settings

6. **Monitor and log** - Log errors and track success rates
7. **Keep session IDs secure** - Use cryptographically secure UUID generation (see UUID generation section below)
8. **Handle cookie restrictions** - Some browsers block third-party cookies; prefer serving Unomi from the website's origin (see Websites on other origins)

18.35.14. UUID Generation Security

Security Considerations for UUID Generation

Session IDs are security-relevant: a session ID is a bearer identifier, so anyone who can predict one can take over the corresponding visitor session. The implementation in this guide therefore uses only cryptographically secure sources, and **fails closed** when none is available:

1. **crypto.randomUUID()** (Preferred): Available in modern browsers (Chrome 92+, Firefox 95+, Safari 15.4+, Edge 92+). This is the most secure option and generates RFC 4122-compliant UUIDs using cryptographically secure random number generation.
2. **crypto.getRandomValues()** (Fallback): Available in all modern browsers (IE 11+, Chrome 11+, Firefox 21+, Safari 5.1+). Uses the Web Crypto API to generate cryptographically secure random values. This is secure and widely compatible.
3. **No secure source throw**: if neither Web Crypto entry point exists, generateUUID() throws an Error instead of returning an identifier. There is deliberately **no Math.random() fallback**: Math.random() is not cryptographically secure, its output is predictable, and returning such a value would silently hand untrusted callers a guessable session ID. Refusing to track is the safer outcome.



Recommendations: * For production applications, ensure your minimum browser support includes browsers with crypto.getRandomValues() support (essentially all browsers from 2013+) * Serve your pages over HTTPS. crypto.randomUUID() is restricted to secure contexts, so over plain HTTP the tracker falls back to crypto.getRandomValues(), which is available in non-secure contexts too. HTTPS is worth doing on its own merits - a session identifier travelling in clear text is the larger problem - but plain HTTP alone does not reach the fail-closed branch on a current browser * If you must support browsers without the Web Crypto API (IE 10 and below), install a Web Crypto polyfill — do not reintroduce a Math.random() fallback * Never use Math.random() for security-sensitive identifiers like session IDs or authentication tokens * Handle the thrown error in your integration (for example, disable tracking and log a warning) rather than letting it break unrelated page scripts * The profile ID is always generated server-side by Apache Unomi using secure UUID generation, so client-side UUID generation is only needed for session IDs

Browser Compatibility: * crypto.randomUUID(): Chrome 92+, Firefox 95+, Safari 15.4+, Edge 92+ (2021+), secure contexts only * crypto.getRandomValues(): All modern browsers (2013+), available in secure and non-secure contexts * Anything older, or any non-secure context: unsupported — the tracker throws rather than generating a weak session ID

The implementation automatically uses the best available secure method, and refuses to generate a session ID when there is none.

18.35.15. Next Steps

- Learn about How profile tracking works to understand the full flow
- Review the official Unomi Web Tracker for a production-ready implementation
- Explore Request examples for more API usage patterns
- Check Built-in event types for available event types
- Review Built-in condition types for filter conditions

Chapter 19. Integration samples

19.1. Samples

Apache Unomi provides the following integration demos in samples/. These are scenario walkthroughs, not maintained plugin templates — for plugin development see Writing plugins.

- Twitter integration
- Login integration
- Weather update extension

19.2. Login sample

This sample shows how to integrate a **server-side** login with Apache Unomi so that mergeProfilesOnPropertyAction can merge profiles on email (or another merge key).

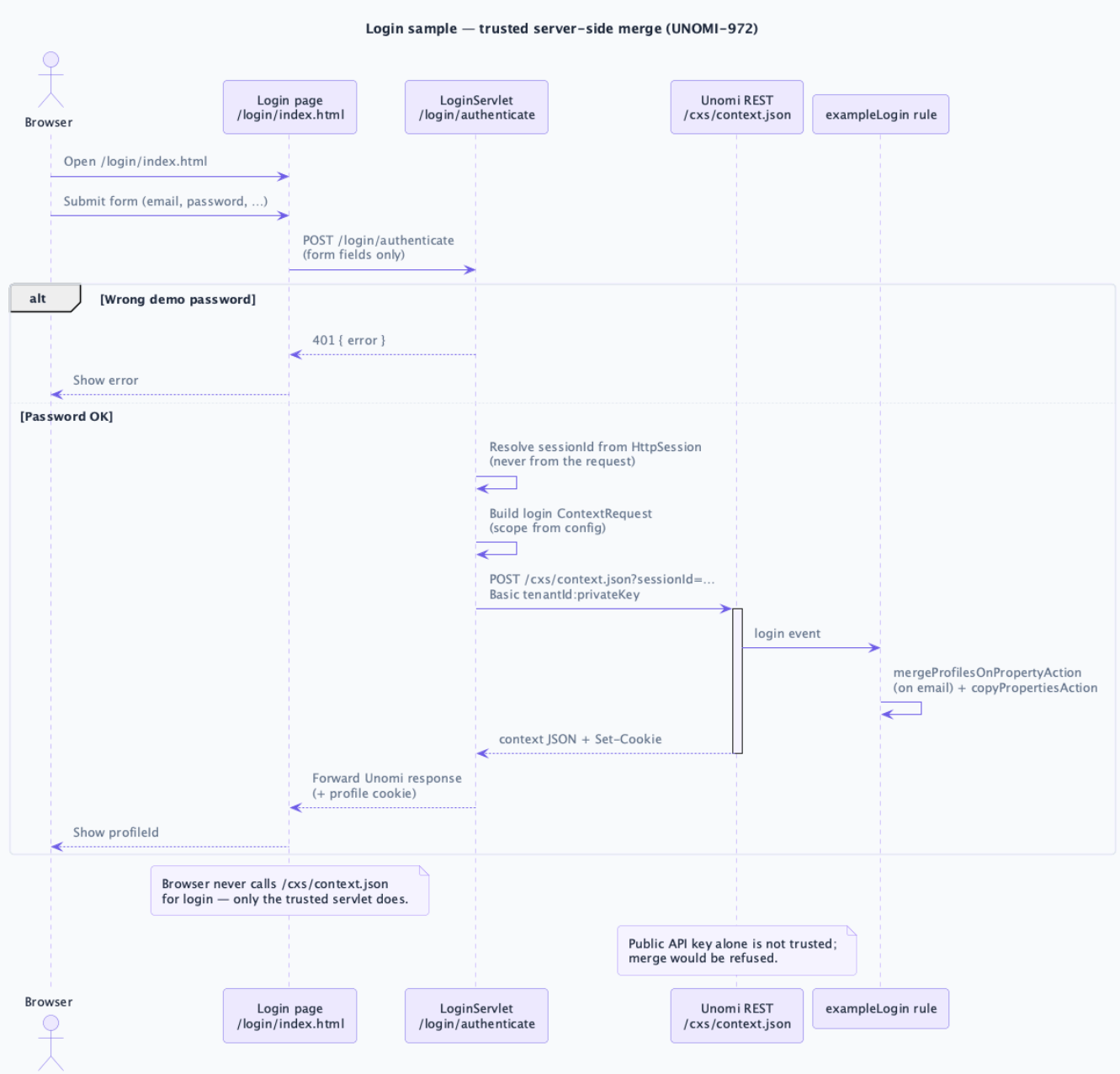
19.2.1. Why server-side?

Under Unomi 4.0 hardening (UNOMI-972 / client-facing hardening), cross-profile merge requires a **trusted** caller (system administrator or tenant private key). A browser call with only a public API key is not trusted — merge is refused.

A real authentication server must validate the password, then call Unomi. This sample mimics that with a small DS servlet inside Unomi (/login/authenticate).

19.2.2. How it works

1. Open http://localhost:8181/login/index.html and submit the form (the password you set as demoPassword in the sample configuration).
2. The page posts to /login/authenticate (same host) — it never posts a login event to /cxs/context.json from JavaScript.
3. LoginServlet checks the demo password, then POST`s a `login event to /cxs/context.json using Basic auth (tenantId:privateKey — a tenant private key is the only credential the sample accepts).
4. The session id is generated by the servlet and kept on the browser's own HttpSession — it is deliberately **not** read from the form. See Never forward client-supplied identifiers.
5. The bundled rule exampleLogin (META-INF/cxs/rules/exampleLogin.json) runs mergeProfilesOnPropertyAction on target.properties.email plus copyPropertiesAction.



Source: samples/login-integration/ (OSGi Declarative Services, no Blueprint).

19.2.3. Never forward client-supplied identifiers

Moving the login event server-side is only half of the fix. A trusted caller is allowed to adopt whatever profile owns the sessionId it passes, and to merge profiles on whatever identifier the event carries. A proxy that takes those values from its own untrusted callers and replays them under trusted credentials hands that power straight back to the browser: anyone who guesses another visitor's session id could rebind or merge that visitor's profile.

LoginServlet therefore derives the session id from state it controls — a UUID stored on the browser's container HttpSession — and never reads it from a request parameter. The consequence is that the anonymous profile the browser built up through the public tracker before logging in is **not** linked by this sample: doing so requires forwarding a browser-chosen identifier (the profile cookie or the tracker's session id) under trusted credentials, which your server must only do once it has established that the identifier belongs to the caller. Apply the same rule in your own integration:

- Derive the sessionId from your authenticated server-side session, not from the request body.
- Derive the merge identifier (here, the email) from the account you just authenticated, not from a form field the caller chose.
- Treat every other Unomi identifier the same way: if the value came from the caller, it must not be forwarded under credentials the caller does not hold.



The sample's same-origin check is a lightweight stand-in for CSRF protection, and demoPassword is a single shared secret standing in for a user directory. A real integration should use a per-session CSRF token and authenticate each user against your identity provider. The sample ships no default demoPassword for the same reason Unomi ships no default admin password: a credential published in source is a credential everyone has.

19.2.4. Build

From the Unomi source tree:

```
mvn -pl samples/login-integration -am install -DskipTests
```

19.2.5. Set up

Build the sample, then run the setup script on the Unomi host. It creates the tenant and scope, issues a tenant private key, generates a demo password, configures the bundle and starts it:

```
mvn -pl samples/login-integration -am install -DskipTests

export UNOMI_ROOT_PASSWORD='your-admin-password'
./samples/login-integration/setup.sh
```

KARAF_HOME must point at the Unomi install that is **actually running** and serving UNOMI_URL — the script writes into that install's etc/ and deploy/ directories, so configuring a different copy would have no effect. Set it explicitly unless you have exactly one built distribution in the source tree:

```
cd samples/login-integration
KARAF_HOME=../../package/target/unomi-4.0.0-SNAPSHOT ./setup.sh
```

The script validates this before changing anything, and refuses to continue if the directory does not exist, does not look like a Unomi install, is not writable, or belongs to an instance that is stopped (it checks karaf.pid against the running process, so a stale pid file from a previous run is caught too). If you have more than one built distribution under package/target, it will not guess — set KARAF_HOME.

The script waits until the sample page answers before reporting success, then prints the page URL and the **generated demo password** — copy it, it is not stored anywhere you can read it back. The private key is never printed. Re-running is safe: it reuses an existing tenant and scope and issues a fresh key.

It needs curl and jq, and filesystem access to the Unomi install — it writes etc/org.apache.unomi.samples.login.cfg (mode 600, since it holds the private key) and copies the bundle into deploy/, both of which Karaf picks up within a second. KARAF_HOME is auto-detected in the source tree; set it otherwise. No Karaf console, SSH or console credential is involved. Override UNOMI_URL, UNOMI_TENANT_ID, UNOMI_SCOPE or DEMO_PASSWORD if the defaults do not suit; ./setup.sh --help lists them.



The script uses the system administrator credential because creating a tenant is an operator action. The servlet never sees it — it receives only the scoped tenant private key the script issues. A tenant private key is the **only** credential the sample accepts: a system administrator password would also satisfy the merge gate, but it grants far more than this sample needs and is not scoped to a single tenant, so the servlet deliberately refuses to use one.

Setting it up by hand

If you would rather not run the script, the equivalent steps are: create the tenant and a scope, POST /cxs/tenants/<tenant>/apikeys?type=PRIVATE and keep the returned plainTextKey (it is shown once), then from the Karaf console:

```
bundle:install mvn:org.apache.unomi/login-integration-sample/${project.version}
config:edit org.apache.unomi.samples.login
config:property-set tenantId default
config:property-set scope default
config:property-set privateKey <plainTextKey>
config:property-set demoPassword <choose-a-password>
config:update
bundle:start <bundle-id>
```

Either way, the bundle reports on activation whether it is usable. A configured sample logs, at INFO:

```
Login sample ready - open http://localhost:8181/login/index.html (tenantId=default, scope=default)
```

A sample still missing something logs a WARN naming exactly what to set. Correct it and run config:update again — the servlet re-reads its configuration and reprints the status without a restart.

Ensure the tenant allows login events from the servlet's source IP (typically 127.0.0.1 when calling localhost) via tenant authorized IPs.

19.2.6. Test profile merge

1. Open http://localhost:8181/login/index.html
2. Log in with email alice@example.com and your configured demoPassword. Note the profileId in the success message.
3. Clear the context-profile-id cookie for localhost (or use a private browser window) so Unomi would otherwise create a new anonymous profile.

4. Log in again with the **same** email. Expect the **same** master profileId (merge on email).
5. Optional: log in with a different email — expect a different profile.

If you see Unable to resolve a tenant, create the tenant and ensure tenantId matches the tenant the private key belongs to — the key's tenant is what Unomi authenticates against. If schema validation rejects the scope, create the scope (see above). If you see a configuration error from /login/authenticate, the servlet has no trusted credentials yet — set privateKey as shown above and run config:update.

19.2.7. Related

- Client-facing hardening (4.0)
- How profile tracking works
- Rule JSON: https://github.com/apache/unomi/blob/master/samples/login-integration/src/main/resources/META-INF/cxs/rules/exampleLogin.json

19.3. Twitter sample

19.3.1. Overview

We will examine how a simple HTML page can interact with Unomi to enrich a user's profile. The use case we will follow is a rather simple one: we use a Twitter button to record the number of times the visitor tweeted (as a tweetNb profile integer property) as well as the URLs they tweeted from (as a tweetedFrom multi-valued string profile property). A javascript script will use the Twitter API to react to clicks on this button and update the user profile using a ContextServlet request triggering a custom event. This event will, in turn, trigger a Unomi action on the server implemented using a Unomi plugin, a standard extension point for the server.

Building the tweet button samples

In your local copy of the Unomi repository and run:

```
cd samples/tweet-button-plugin
mvn clean install
```

This will compile and create the OSGi bundle that can be deployed on Unomi to extend it.

Deploying the tweet button samples

In standard Karaf fashion, you will need to copy the samples bundle to your Karaf deploy directory.

If you are using the packaged version of Unomi (as opposed to deploying it to your own Karaf version), you can simply run, assuming your current directory is samples/tweet-button-plugin and that you uncompressed the archive in the directory it was created:

```
cp target/tweet-button-plugin-4.0.0-SNAPSHOT.jar ../../package/target/unomi-4.0.0-SNAPSHOT/deploy
```

Testing the samples

You can now go to http://localhost:8181/twitter/index.html to test the samples code. The page is very simple, you will see a Twitter button, which, once clicked, will open a new window to tweet about the current page. The original page should be updated with the new values of the properties coming from Unomi. Additionnally, the raw JSON response is displayed.

We will now explain in greater details some concepts and see how the example works.

19.3.2. Interacting with the context server

There are essentially two modalities to interact with the context server, reflecting different types of Unomi users: context server clients and context server integrators.

Context server clients are usually web applications or content management systems. They interact with Unomi by providing raw, uninterpreted contextual data in the form of events and associated metadata. That contextual data is then processed by the context server to be fed to clients once actionable. In that sense context server clients are both consumers and producers of contextual data. Context server clients will mostly interact with Unomi using a single entry point called the ContextServlet, requesting context for the current user and providing any triggered events along the way.

On the other hand, **context server integrators** provide ways to feed more structured data to the context server either to integrate with third party services or to provide analysis of the uninterpreted data provided by context server clients. Such integration will mostly be done using Unomi's API either directly using Unomi plugins or via the provided REST APIs. However, access to REST APIs is restricted due for security reasons, requiring privileged access to the Unomi server, making things a little more complex to set up.

For simplicity's sake, this document will focus solely on the first use case and will interact only with the context servlet.

19.3.3. Retrieving context information from Unomi using the context servlet

Unomi provides two ways to retrieve context: either as a pure JSON object containing strictly context information or as a couple of JSON objects augmented with javascript functions that can be used to interact with the Unomi server using the <context server base URL>/cxs/context.json or <context server base URL>/context.js URLs, respectively.

Below is an example of asynchronously loading the initial context using the javascript version, assuming a default Unomi install running on http://localhost:8181:

```
// Load context from Unomi asynchronously
(function (document, elementToCreate, id) {
    var js, fjs = document.getElementsByTagName(elementToCreate)[0];
    if (document.getElementById(id)) return;
    js = document.createElement(elementToCreate);
    js.id = id;
    js.src = 'http://localhost:8181/cxs/context.js';
    fjs.parentNode.insertBefore(js, fjs);
}(document, 'script', 'context'));
```

This initial context results in a javascript file providing some functions to interact with the context server from javascript along with two objects: a cxs object containing information about the context for the current user and a digitalData object that is injected into the browser's window object (leveraging the Customer Experience Digital Data Layer standard). Note that this last object is not under control of the context server and clients are free to use it or not. Our example will not make use of it.

On the other hand, the cxs top level object contains interesting contextual information about the current user:

```
{
  "profileId":<identifier of the profile associated with the current user>,
  "sessionId":<identifier of the current user session>,
  "profileProperties":<requested profile properties, if any>,
  "sessionProperties":<requested session properties, if any>,
  "profileSegments":<segments the profile is part of if requested>,
  "filteringResults":<result of the evaluation of content filters>,
  "personalizations":<result of the evaluation of personalization filters>,
  "trackedConditions":<tracked conditions in the source page, if any>
}
```

We will look at the details of the context request and response later.

19.4. Example

19.4.1. HTML page

The code for the HTML page with our Tweet button can be found at https://github.com/apache/unomi/blob/master/wab/src/main/webapp/index.html.

This HTML page is fairly straightforward: we create a tweet button using the Twitter API while a Javascript script performs the actual logic.

19.4.2. Javascript

Globally, the script loads both the twitter widget and the initial context asynchronously (as shown previously). This is accomplished using fairly standard javascript code and we won't look at it here. Using the Twitter API, we react to the tweet event and call the Unomi server to update the user's profile with the required information, triggering a custom tweetEvent event. This is accomplished using a

contextRequest function which is an extended version of a classic AJAX request:

```
function contextRequest(successCallback, errorCallback, payload) {
    var data = JSON.stringify(payload);
    // if we don't already have a session id, generate one
    var sessionId = cxs.sessionId || generateUUID();
    var url = 'http://localhost:8181/cxs/context.json?sessionId=' + sessionId;
    var xhr = new XMLHttpRequest();
    var isGet = data.length < 100;
    if (isGet) {
        xhr.withCredentials = true;
        xhr.open("GET", url + "&payload=" + encodeURIComponent(data), true);
    } else if ("withCredentials" in xhr) {
        xhr.open("POST", url, true);
        xhr.withCredentials = true;
    } else if (typeof XDomainRequest != "undefined") {
        xhr = new XDomainRequest();
        xhr.open("POST", url);
    }
    xhr.onreadystatechange = function () {
        if (xhr.readyState != 4) {
            return;
        }
        if (xhr.status ==== 200) {
            var response = xhr.responseText ? JSON.parse(xhr.responseText) : undefined;
            if (response) {
                cxs.sessionId = response.sessionId;
                successCallback(response);
            }
        } else {
            console.log("contextserver: " + xhr.status + " ERROR: " + xhr.statusText);
            if (errorCallback) {
                errorCallback(xhr);
            }
        }
    };
    xhr.setRequestHeader("Content-Type", "text/plain;charset=UTF-8"); // Use text/plain to avoid CORS
preflight
    if (isGet) {
        xhr.send();
    } else {
        xhr.send(data);
    }
}
```

There are a couple of things to note here:

- If we specify a payload, it is expected to use the JSON format so we stringify it and encode it if passed as a URL parameter in a GET request.
- We need to make a CORS request since the Unomi server is most likely not running on the same host than the one from which the request originates. The specific details are fairly standard and we will not explain them here. In Unomi 4.0 the page's origin must be listed in UNOMI_PUBLIC_CORS_ALLOWED_ORIGINS for the request to carry the visitor cookie (see Websites

on other origins).

- We need to either retrieve (from the initial context we retrieved previously using cxs.sessionId) or generate a session identifier for our request since Unomi currently requires one.
- We're calling the ContextServlet using the default install URI, specifying the session identifier: http://localhost:8181/cxs/context.json?sessionId=sessionId. This URI requests context from Unomi, resulting in an updated cxs object in the javascript global scope. The context server can reply to this request either by returning a JSON-only object containing solely the context information as is the case when the requested URI is context.json. However, if the client requests context.js then useful functions to interact with Unomi are added to the cxs object in addition to the context information as depicted above.
- We don't need to provide any authentication at all to interact with this part of Unomi since we only have access to read-only data (as well as providing events as we shall see later on). If we had been using the REST API, we would have needed to provide authentication information as well.

Context request and response structure

The interesting part, though, is the payload. This is where we provide Unomi with contextual information as well as ask for data in return. This allows clients to specify which type of information they are interested in getting from the context server as well as specify incoming events or content filtering or property/segment overrides for personalization or impersonation. This conditions what the context server will return with its response.

Let's look at the context request structure:

```
{
  "sessionId" : <optional session identifier>,
  "source": <Item source of the context request>,
  "events": <optional array of events to trigger>,
  "requiredProfileProperties": <optional array of property identifiers>,
  "requiredSessionProperties": <optional array of property identifiers>,
  filters: <optional array of filters to evaluate>,
  "personalizations": <optional array of personalizations to evaluate>,
  "profileOverrides": <optional profile containing segments,scores or profile properties to override>,
    - segments: <optional array of segment identifiers>,
    - profileProperties: <optional map of property name / value pairs>,
    - scores: <optional map of score id / value pairs>
  "sessionPropertiesOverrides": <optional map of property name / value pairs>,
  "requireSegments": <boolean, whether to return the associated segments>
}
```

We will now look at each part in greater details.

Source

A context request payload needs to at least specify some information about the source of the request in the form of an Item (meaning identifier, type and scope plus any additional properties we might have to provide), via the source property of the payload. Of course the more information can be provided about the source, the better.

Filters

A client wishing to perform content personalization might also specify filtering conditions to be evaluated by the context server so that it can tell the client whether the content associated with the filter should be activated for this profile/session. This is accomplished by providing a list of filter definitions to be evaluated by the context server via the filters field of the payload. If provided, the evaluation results will be provided in the filteringResults field of the resulting cxs object the context server will send.

Here is an example of a filter request:

```
curl --location --request POST 'http://localhost:8181/cxs/context.json' \
--header 'Content-Type: application/json' \
--header 'X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY' \
--header 'Cookie: JSESSIONID=48C8AFB3E18B8E3C93C2F4D5B7BD43B7; context-profile-id=01060c4c-a055-4c8f-9692-8a699d0c434a' \
--data-raw '{
    "source": null,
    "requireSegments": false,
    "requiredProfileProperties": null,
    "requiredSessionProperties": null,
    "events": null,
    "filters": [
        {
            "id" : "filter1",
            "filters" : [
                {
                    "condition": {
                    "parameterValues": {
                        "propertyName": "properties.gender",
                        "comparisonOperator": "equals",
                        "propertyValue": "male"
                    },
                    "type": "profilePropertyCondition"
                }
                }
            ]
        }
    ],
    "personalizations": null,
    "profileOverrides": null,
    "sessionPropertiesOverrides": null,
    "sessionId": "demo-session-id"
}'
```

And here's the result:

```
{
   "profileId": "01060c4c-a055-4c8f-9692-8a699d0c434a",
   "sessionId": "demo-session-id",
   "profileProperties": null,
   "sessionProperties": null,
   "profileSegments": null,
   "filteringResults": {
      "filter1": false
   },
   "processedEvents": 0,
   "personalizations": null,
   "trackedConditions": [],
   "anonymousBrowsing": false,
   "consents": {}
}
```


As we can see, the filter1 filter we sent in our request, in this example, evaluated to false for the current profile, so we can use that result to perform any customization for the current profile, in this case use the fact that he is male.

Personalizations

Filters make it possible to evaluate conditions against a profile in real-time, but for true personalization it is needed to have a more powerful mechanism: strategies. Sometimes we want to provide multiple variants that each have their own conditions and we want to know which is the best variant to use for the current profile. This can be achieved with the personalizations structure in the ContextRequest.

Here is an example of a personalizations request:

```
curl --location --request POST 'http://localhost:8181/cxs/context.json' \
--header 'Content-Type: application/json' \
--header 'X-Unomi-Api-Key: YOUR_PUBLIC_API_KEY' \
--header 'Cookie: JSESSIONID=48C8AFB3E18B8E3C93C2F4D5B7BD43B7; context-profile-id=01060c4c-a055-4c8f-9692-8a699d0c434a' \
--data-raw '{
    "source": null,
    "requireSegments": false,
    "requiredProfileProperties": null,
    "requiredSessionProperties": [ "unomiControlGroups" ],
    "events": null,
    "filters": null,
    "personalizations": [
      {
        "id": "gender-test",
        "strategy": "matching-first",
        "strategyOptions": {
          "fallback": "var2",
          "controlGroup" : {
            "percentage" : 10.0,
            "displayName" : "Gender test control group",
            "path" : "/gender-test",
            "storeInSession" : true
          }
        },
        "contents": [
          {
            "id": "var1",
            "filters": [
              {
                "appliesOn": null,
                "condition": {
                  "parameterValues": {
                    "propertyName": "properties.gender",
                    "comparisonOperator": "equals",
                    "propertyValue": "male"
                  },
                  "type": "profilePropertyCondition"
                },
                "properties": null
              }
            ],
            "properties": null
          },
          {
            "id": "var2",
            "filters": null,
            "properties": null
          }
        ]
      }
    ],
    "profileOverrides": null,
    "sessionPropertiesOverrides": null,
    "sessionId": "demo-session-id"
}'
```

In the above example, we basically setup two variants : var1 and var2 and setup the var2 to be the fallback variant in case no variant is matched. We could of course specify more than a variant. The strategy indicates to the personalization service how to calculate the "winning" variant. In this case the strategy matching-first will return variants that match the current profile. We also use the controlGroups option to specify that we want to have a control group for this personalization. The 10.0 percentage value represents 10% (0.0 to 100.0) of traffic that will be assigned randomly to the control group. The control group will be stored in the profile and the session of the visitors if they were assigned to it. We also specify that we store the control group in the session (by default it is stored in the profile)

Currently the following strategies are available:

- matching-first: will return the variant IDs that match the current profile (using the initial content order)
- random: will return a shuffled list of variant IDs (ignoring any conditions)
- score-sorted: allows to sort the variants based on scores associated with the filtering conditions, effectively sorting them by the highest scoring condition first.

Here is the result of the above example:


```
{
  "profileId": "01060c4c-a055-4c8f-9692-8a699d0c434a",
  "sessionId": "demo-session-id",
  "profileProperties": null,
  "sessionProperties": {
    "unomiControlGroups": [
      {
        "id": "previousPerso",
        "displayName": "Previous perso",
        "path": "/home/previousPerso.html",
        "timeStamp": "2021-12-15T13:52:38Z"
      }
    ]
  },
  "profileSegments": null,
  "filteringResults": null,
  "processedEvents": 0,
  "personalizations": {
    "gender-test": [
      "var2"
    ]
  },
  "trackedConditions": [
  ],
  "anonymousBrowsing": false,
  "consents": {}
}
```


In the above example we can see the profile and session were assigned to other control groups but not the current one (the ids are different).

Overrides

It is also possible for clients wishing to perform user impersonation to specify properties or segments to override the proper ones so as to emulate a specific profile, in which case the overridden value will temporarily replace the proper values so that all rules will be evaluated with these values instead of the proper ones. The segments (array of segment identifiers), profileProperties (maps of property name and associated object value) and scores (maps of score id and value) all wrapped in a profileOverrides object and the sessionPropertiesOverrides (maps of property name and associated object value) fields allow to provide such information. Providing such overrides will, of course, impact content filtering results and segments matching for this specific request.

Controlling the content of the response

The clients can also specify which information to include in the response by setting the requireSegments property to true if segments the current profile matches should be returned or provide an array of property identifiers for requiredProfileProperties or requiredSessionProperties fields to ask the context server to return the values for the specified profile or session properties, respectively. This information is provided by the profileProperties, sessionProperties and profileSegments fields of the context server response.

Additionally, the context server will also returns any tracked conditions associated with the source of the context request. Upon evaluating the incoming request, the context server will determine if there are any rules marked with the trackedCondition tag and which source condition matches the source of the incoming request and return these tracked conditions to the client. The client can use these tracked conditions to learn that the context server can react to events matching the tracked condition and coming from that source. This is, in particular, used to implement form mapping (a solution that allows clients to update user profiles based on values provided when a form is submitted).

Events

Finally, the client can specify any events triggered by the user actions, so that the context server can process them, via the events field of the context request.

Default response

If no payload is specified, the context server will simply return the minimal information deemed necessary for client applications to properly function: profile identifier, session identifier and any tracked conditions that might exist for the source of the request.

Context request for our example

Now that we've seen the structure of the request and what we can expect from the context response, let's examine the request our component is doing.

In our case, our source item looks as follows: we specify a scope for our application (unomi-tweet-button-samples), specify that the item type (i.e. the kind of element that is the source of our event) is a page (which corresponds, as would be expected, to a web page), provide an identifier (in our case, a Base-64 encoded version of the page's URL) and finally, specify extra properties (here, simply a url property corresponding to the page's URL that will be used when we process our event in our Unomi

extension).

```
var scope = 'unomi-tweet-button-samples';
var itemId = btoa(window.location.href);
var source = {
   itemType: 'page',
   scope: scope,
   itemId: itemId,
   properties: {
      url: window.location.href
   }
};
```

We also specify that we want the context server to return the values of the tweetNb and tweetedFrom profile properties in its response. Finally, we provide a custom event of type tweetEvent with associated scope and source information, which matches the source of our context request in this case.

```
var contextPayload = {
   source: source,
   events: [
      {
         eventType: 'tweetEvent',
         scope: scope,
         source: source
      }
   ],
   requiredProfileProperties: [
      'tweetNb',
      'tweetedFrom'
   ]
};
```

The tweetEvent event type is not defined by default in Unomi. This is where our Unomi plugin comes into play since we need to tell Unomi how to react when it encounters such events.

Unomi plugin overview

In order to react to tweetEvent events, we will define a new Unomi rule since this is exactly what Unomi rules are supposed to do. Rules are guarded by conditions and if these conditions match, the associated set of actions will be executed. In our case, we want our new incrementTweetNumber rule to only react to tweetEvent events and we want it to perform the profile update accordingly: create the property types for our custom properties if they don't exist and update them. To do so, we will create a custom incrementTweetNumberAction action that will be triggered any time our rule matches. An action is some custom code that is deployed in the context server and can access the Unomi API to perform what it is that it needs to do.

Rule definition

Let's look at how our custom incrementTweetNumber rule is defined:

```
{
 "metadata": {
  "id": "smp:incrementTweetNumber",
  "name": "Increment tweet number",
  "description": "Increments the number of times a user has tweeted after they click on a tweet
button"
 },
 "raiseEventOnlyOnceForSession": false,
 "condition": {
  "type": "eventTypeCondition",
  "parameterValues": {
   "eventTypeId": "tweetEvent"
  }
 },
 "actions": [
  {
   "type": "incrementTweetNumberAction",
   "parameterValues": {}
  }
 ]
}
```

Rules define a metadata section where we specify the rule name, identifier and description.

When rules trigger, a specific event is raised so that other parts of Unomi can react to it accordingly. We can control how that event should be raised. Here we specify that the event should be raised each time the rule triggers and not only once per session by setting raiseEventOnlyOnceForSession to false, which is not strictly required since that is the default. A similar setting (raiseEventOnlyOnceForProfile) can be used to specify that the event should only be raised once per profile if needed.

We could also specify a priority for our rule in case it needs to be executed before other ones when similar conditions match. This is accomplished using the priority property. We're using the default priority here since we don't have other rules triggering on `tweetEvent`s and don't need any special ordering.

We then tell Unomi which condition should trigger the rule via the condition property. Here, we specify that we want our rule to trigger on an eventTypeCondition condition. Unomi can be extended by adding new condition types that can enrich how matching or querying is performed. The condition type definition file specifies which parameters are expected for our condition to be complete. In our case, we use the built-in event type condition that will match if Unomi receives an event of the type specified in the condition's eventTypeId parameter value: tweetEvent here.

Finally, we specify a list of actions that should be performed as consequences of the rule matching. We only need one action of type incrementTweetNumberAction that doesn't require any parameters.

Action definition

Let's now look at our custom incrementTweetNumberAction action type definition:

```
{
  "id": "incrementTweetNumberAction",
  "actionExecutor": "incrementTweetNumber",
  "systemTags": [
    "event"
  ],
  "parameters": []
}
```

We specify the identifier for the action type, a list of systemTags if needed: here we say that our action is a consequence of events using the event tag. Our actions does not require any parameters so we don't define any.

Finally, we provide a mysterious actionExecutor identifier: incrementTweetNumber.

Action executor definition

The action executor references the actual implementation of the action as defined in our blueprint definition:

```
<blueprint xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
        xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0
http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

   <reference id="profileService" interface="org.apache.unomi.api.services.ProfileService"/>

   <!-- Action executor -->
   <service id="incrementTweetNumberAction"
interface="org.apache.unomi.api.actions.ActionExecutor">
      <service-properties>
         <entry key="actionExecutorId" value="incrementTweetNumber"/>
      </service-properties>
      <bean
class="org.apache.unomi.examples.unomi_tweet_button_plugin.actions.IncrementTweetNumberAct
ion">
         <property name="profileService" ref="profileService"/>
      </bean>
   </service>
</blueprint>
```

In standard Blueprint fashion, we specify that we will need the profileService defined by Unomi and then define a service of our own to be exported for Unomi to use. Our service specifies one property: actionExecutorId which matches the identifier we specified in our action definition. We then inject the profile service in our executor and we're done for the configuration side of things!

Action executor implementation

Our action executor definition specifies that the bean providing the service is implemented in the org.apache.unomi.samples.tweet_button_plugin.actions .IncrementTweetNumberAction class. This class

implements the Unomi ActionExecutor interface which provides a single int execute(Action action, Event event) method: the executor gets the action instance to execute along with the event that triggered it, performs its work and returns an integer status corresponding to what happened as defined by public constants of the EventService interface of Unomi: NO_CHANGE, SESSION_UPDATED or PROFILE_UPDATED.

Let's now look at the implementation of the method:

```
final Profile profile = event.getProfile();
Integer tweetNb = (Integer) profile.getProperty(TWEET_NB_PROPERTY);
List<String> tweetedFrom = (List<String>) profile.getProperty(TWEETED_FROM_PROPERTY);

if (tweetNb ==== null || tweetedFrom ==== null) {
   // create tweet number property type
   PropertyType propertyType = new PropertyType(new Metadata(event.getScope(),
TWEET_NB_PROPERTY, TWEET_NB_PROPERTY, "Number of times a user tweeted"));
   propertyType.setValueTypeId("integer");
   service.createPropertyType(propertyType);

   // create tweeted from property type
   propertyType = new PropertyType(new Metadata(event.getScope(), TWEETED_FROM_PROPERTY,
TWEETED_FROM_PROPERTY, "The list of pages a user tweeted from"));
   propertyType.setValueTypeId("string");
   propertyType.setMultivalued(true);
   service.createPropertyType(propertyType);

   tweetNb = 0;
   tweetedFrom = new ArrayList<>();
}

profile.setProperty(TWEET_NB_PROPERTY, tweetNb + 1);
final String sourceURL = extractSourceURL(event);
if (sourceURL != null) {
   tweetedFrom.add(sourceURL);
}
profile.setProperty(TWEETED_FROM_PROPERTY, tweetedFrom);

return EventService.PROFILE_UPDATED;
```

It is fairly straightforward: we retrieve the profile associated with the event that triggered the rule and check whether it already has the properties we are interested in. If not, we create the associated property types and initialize the property values.

> Note that it is not an issue to attempt to create the same property type multiple times as Unomi will not add a new property type if an identical type already exists.

Once this is done, we update our profile with the new property values based on the previous values and the metadata extracted from the event using the extractSourceURL method which uses our url property that we've specified for our event source. We then return that the profile was updated as a result of our action and Unomi will properly save it for us when appropriate. That's it!

For reference, here's the extractSourceURL method implementation:

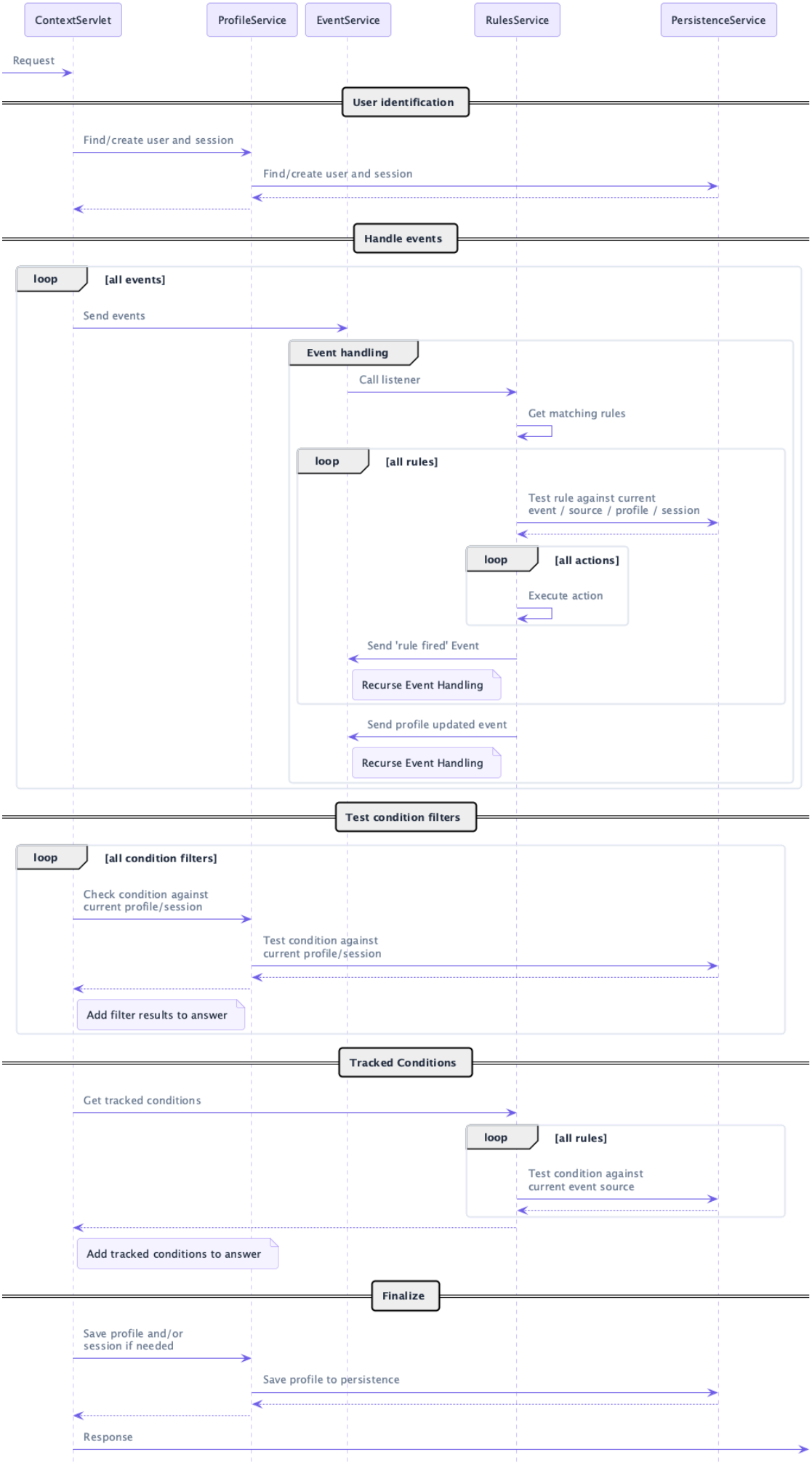
```
private String extractSourceURL(Event event) {
    final Item sourceAsItem = event.getSource();
    if (sourceAsItem instanceof CustomItem) {
        CustomItem source = (CustomItem) sourceAsItem;
        final String url = (String) source.getProperties().get("url");
        if (url != null) {
            return url;
        }
    }

    return null;
}
```

19.5. Conclusion

We have seen a simple example how to interact with Unomi using a combination of client-side code and Unomi plugin. Hopefully, this provided an introduction to the power of what Unomi can do and how it can be extended to suit your needs.

19.6. Annex

Here is an overview of how Unomi processes incoming requests to the ContextServlet.


ContextServlet
ProfileService
EventService
RulesService
PersistenceService
Request
User identification
Find/create user and session
Find/create user and session
Handle events
loop
[all events]
Send events
Event handling
Call listener
Get matching rules
loop
[all rules]
Test rule against current
event / source / profile / session
loop
[all actions]
Execute action
Send 'rule fired' Event
Recurse Event Handling
Send profile updated event
Recurse Event Handling
Test condition filters
loop
[all condition filters]
Check condition against
current profile/session
Test condition against
current profile/session
Add filter results to answer
Tracked Conditions
Get tracked conditions
loop
[all rules]
Test condition against
current event source
Add tracked conditions to answer
Finalize
Save profile and/or
session if needed
Save profile to persistence
Response

19.7. Weather update sample

The **weather-update** extension enriches the visitor session with weather properties based on the resolved geo-location (typically from IP lookup). It calls the OpenWeatherMap API and stores values such as temperature, “feels like”, wind speed, and wind direction on the session.

Source and packaging live under extensions/weather-update/ in the Unomi repository. See also the extension README: https://github.com/apache/unomi/blob/master/extensions/weather-update/README.md

19.7.1. Prerequisites

- Apache Unomi running with a tenant and API keys (see Multi-tenancy)
- An OpenWeatherMap account and API key: https://home.openweathermap.org/api_keys
- GeoIP / location resolution configured so sessions have a usable location (see Geonames / GeoIP sections in Configuration)

19.7.2. Install the extension

From the Karaf shell (adjust the version to match your Unomi build):

```
feature:repo-add mvn:org.apache.unomi/unomi-weather-update-karaf-kar/4.0.0-SNAPSHOT/xml/features
feature:install unomi-weather-update-karaf-kar
```

19.7.3. Configure the OpenWeatherMap API key

Edit etc/org.apache.unomi.weatherUpdate.cfg:

```
weatherUpdate.apiKey=YOUR_OPENWEATHERMAP_API_KEY
```

19.7.4. What it updates

The extension registers session property types under the weather tag (for example weatherTemp, weatherLike, weatherWindSpeed, weatherWindDirection) and an action that fills them when a suitable event/rule fires.

Wire a rule that runs the weather-update action on session start or first view (using the tenant private API key or JAAS), then verify session properties via POST /cxs/profiles/search / session APIs or a context request with requiredSessionProperties.

19.7.5. Related

- Twitter sample — another extension-driven action example

- Built-in action types — action plugin patterns

Chapter 20. Connectors

20.1. Connectors

Apache Unomi provides the following connectors:

- Salesforce CRM connector

20.1.1. Call for contributors

We are looking for help with the development of additional connectors. Any contribution (large or small) is more than welcome. Feel free to discuss this in our mailing list.

20.2. Salesforce Connector

This connectors makes it possible to push and pull data to/from the Salesforce CRM. It can copy information between Apache Unomi profiles and Salesforce Leads.



On Unomi **4.0**, profile and rule APIs used by the connector operate in a **tenant** context. Ensure the Unomi side uses tenant private-key or JAAS credentials with an explicit tenant (see Multi-tenancy). The connector's Salesforce OAuth configuration is unchanged.

20.2.1. Getting started

Salesforce account setup

1. Create a new developer account here:

```
https://developer.salesforce.com/signup
```

2. Create a new Connected App, by going into Setup -> App Manager and click "Create Connected App"
3. In the settings, make sure you do the following:

```
Enable OAuth settings -> Activated
Enable for device flow -> Activated (no need for a callback URL)
Add all the selected OAuth scopes you want (or put all of them)
Make sure Require Secret for Web Server flow is activated
```

4. Make sure you retrieve the following information once you have created the app in the API (Enable OAuth Settings):

```
Consumer key
Consumer secret (click to see it)
```

5. You must also retrieve your user's security token, or create it if you don't have one already. To do this simply click on your user at the top right, select "Settings", the click on "Reset my security token". You will receive an email with the security token.

Apache Unomi setup

1. You are now ready to configure the Apache Unomi Salesforce Connector. In the etc/unomi.custom.system.properties file add/change the following settings:

```
org.apache.unomi.sfdc.user.username=${env:UNOMI_SFDC_USER_USERNAME:-}
org.apache.unomi.sfdc.user.password=${env:UNOMI_SFDC_USER_PASSWORD:-}
org.apache.unomi.sfdc.user.securityToken=${env:UNOMI_SFDC_USER_SECURITYTOKEN:-}
org.apache.unomi.sfdc.consumer.key=${env:UNOMI_SFDC_CONSUMER_KEY:-}
org.apache.unomi.sfdc.consumer.secret=${env:UNOMI_SFDC_CONSUMER_SECRET:-}
```

Deployment from Maven repository

In this procedure we assume you have access to a Maven repository that contains a compiled version of the Salesforce connector. If this is not the case or you prefer to deploy using a KAR bundle, see the KAR deployment instructions instead.

1. Connect to the Apache Unomi Karaf Shell using :

```
ssh -p 8102 karaf@localhost (the password is your configured UNOMI_ROOT_PASSWORD)
```

2. Deploy into Apache Unomi using the following commands from the Apache Karaf shell:

```
feature:repo-add mvn:org.apache.unomi/unomi-salesforce-connector-karaf-
kar/${project.version}/xml/features
feature:install unomi-salesforce-connector-karaf-kar
```

Deployment using KAR bundle

If you have a KAR bundle (for example after building from source in the extensions/salesforce-connector/karaf-kar/target directory), you can follow these steps to install :

1. Ensure that Apache Karaf and Apache Unomi are started
2. Execute the following command in karaf: feature:install unomi-salesforce-connector-karaf-kar
3. The installation is done !

Testing the connector

1. You can then test the connection to Salesforce by accessing the following URLs:

```
https://localhost:9443/cxs/sfdc/version
https://localhost:9443/cxs/sfdc/limits
```

   The first URL will give you information about the version of the connectors, so this makes it easy to check that the plugin is properly deployed, started and the correct version. The second URL will actually make a request to the Salesforce REST API to retrieve the limits of the Salesforce API.

   Both URLs are password protected by the Apache Unomi (Karaf) password: the karaf user and the UNOMI_ROOT_PASSWORD you configured at startup. No default password is shipped, and etc/users.properties only references the configured value rather than containing it.

You can now use the connectors's defined actions in rules to push or pull data to/from the Salesforce CRM. You can find more information about rules in the Data Model and the Getting Started pages.

20.2.2. Properties

To define how Salesforce attributes will be mapped to Unomi profile properties, edit the following entry using the pattern below :

```
org.apache.unomi.sfdc.fields.mappings=${env:UNOMI_SFDC_FIELDS_MAPPINGS:-
email<=>Email,firstName<=>FirstName,lastName<=>LastName,company<=>Company,phoneNumbe
r<=>Phone,jobTitle<=>Title,city<=>City,zipCode<=>PostalCode,address<=>Street,sfdcStatus<=>Status,
sfdcRating<=>Rating}
```

Please note that Salesforce needs the company and the last name to be set, otherwise the lead won't be created. An identifier needs to be set as well. The identifier will be used to map the Unomi profile to the Salesforce lead. By default, the email is set as the identifier, meaning that if a lead in Salesforce and a profile in Unomi have the same email, they'll be considered as the same person.

```
org.apache.unomi.sfdc.fields.mappings.identifier=${env:UNOMI_SFDC_FIELDS_MAPPINGS_IDENTIF
IER:-email<=>Email}
```

20.2.3. Hot-deploying updates to the Salesforce connector (for developers)

If you followed all the steps in the Getting Started section, you can upgrade the Salesforce connectors by using the following steps:

1. Compile the connectors using:

```
cd extensions/salesforce-connector
mvn clean install
```

2. Login to the Unomi Karaf Shell using:

```
ssh -p 8102 karaf@localhost (the password is your configured UNOMI_ROOT_PASSWORD)
```

3. Execute the following commands in the Karaf shell

```
feature:repo-refresh
feature:uninstall unomi-salesforce-connector-karaf-feature
feature:install unomi-salesforce-connector-karaf-feature
```

4. You can then check that the new version is properly deployed by accessing the following URL and checking the build date:

```
https://localhost:9443/cxs/sfdc/version
```

(if asked for credentials, use the karaf user and your configured UNOMI_ROOT_PASSWORD)

20.2.4. Using the Salesforce Workbench for testing REST API

The Salesforce Workbench contains a REST API Explorer that is very useful to test requests. You may find it here :

```
https://workbench.developerforce.com/restExplorer.php
```

20.2.5. Setting up Streaming Push queries

Using the Salesforce Workbench, you can setting streaming push queries (Queries->Streaming push topics) such as the following example:

```
Name: LeadUpdates
Query : SELECT Id,FirstName,LastName,Email,Company FROM Lead
```

20.2.6. Executing the unit tests

Before running the tests, make sure you have completed all the steps above, including the streaming push queries setup.

By default the unit tests will not run as they need proper Salesforce credentials to run. To set this up create a properties file like the following one:

test.properties

```
#
# Licensed to the Apache Software Foundation (ASF) under one or more
# contributor license agreements.  See the NOTICE file distributed with
# this work for additional information regarding copyright ownership.
# The ASF licenses this file to You under the Apache License, Version 2.0
# (the "License"); you may not use this file except in compliance with
# the License.  You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
sfdc.user.username=YOUR_USER_NAME
sfdc.user.password=YOUR_PASSWORD
sfdc.user.securityToken=YOUR_USER_SECURITY_TOKEN
sfdc.consumer.key=CONNECTED_APP_CONSUMER_KEY
sfdc.consumer.secret=CONNECTED_APP_SECRET
```

and then use the following command line to reference the file:

```
cd extensions/salesforce-connector
mvn clean install -DsfdcProperties=../test.properties
```

(in case you're wondering the ../ is because the test is located in the services sub-directory)

Chapter 21. Developers

21.1. Building

Apache Unomi 4.x provides a convenient build.sh script that simplifies the build process with enhanced error handling, preflight validation, and deployment options. See the Using the build.sh script section below for details. Alternatively, you can build using Maven directly as described in this section.

21.1.1. Initial Setup

1. Install J2SE 17 SDK (or later), which can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html
2. Make sure that your JAVA_HOME environment variable is set to the newly installed JDK location, and that your PATH includes %JAVA_HOME%\bin (windows) or $JAVA_HOME$/bin (unix).
3. Install Maven 3.9.8 (or later), which can be downloaded from http://maven.apache.org/download.html. Make sure that your PATH includes the MVN_HOME/bin directory.

21.1.2. Building

Unix/Linux/macOS

1. Get the code: git clone https://github.com/apache/unomi.git
2. Change to the top level directory of Apache Unomi source distribution.
3. Run

```
$> mvn clean install
```

This will compile Apache Unomi and run all of the tests in the Apache Unomi source distribution. Alternatively, you can run

```
$> mvn -P \!integration-tests clean install
```

This will compile Apache Unomi without running the tests and takes less time to build.



On a non-English Windows env, the Asciidoctor Maven Plugin may fail to generate manuals due to an encoding conversion issue. To solve this issue, we recommend setting the **file.encoding** system property to *UTF-8* like the following examples before issuing the commands shown above.

```
> set MAVEN_OPTS=-Dfile.encoding=UTF-8
or
> set MAVEN_OPTS=-Dfile.encoding=UTF-8 -Xmx2048m
...
```

4. The distributions will be available under "package/target" directory.

21.1.3. Using the build.sh script

Apache Unomi 4.x provides a unified build.sh script that simplifies the build, deployment, and execution process. This script replaces multiple legacy build scripts and provides enhanced error handling, colorized output, and comprehensive environment validation.

The build.sh script offers the following features:

- **Preflight validation**: Automatically checks for required tools (Java, Maven, GraphViz), system resources (memory, disk space), Maven settings, and port availability
- **Flexible build options**: Support for skipping tests, running integration tests, using OpenSearch, setting Unomi distribution, debug modes, and more
- **Distribution management**: The --distribution parameter allows you to specify which Unomi distribution to use (e.g., unomi-distribution-opensearch, unomi-distribution-elasticsearch). When used with --use-opensearch, the distribution is automatically set to unomi-distribution-opensearch if not explicitly specified. Conversely, if a distribution containing "opensearch" is specified, OpenSearch is automatically enabled for integration tests.
- **Deployment helpers**: Automatically copies KAR packages to Karaf deploy directory and purges caches
- **Enhanced output**: Colorized, structured output with progress indicators (respects NO_COLOR environment variable)

To use the script:

1. Get the code: git clone https://github.com/apache/unomi.git
2. Change to the top level directory of Apache Unomi source distribution.
3. Make the script executable (if needed): chmod +x build.sh
4. Run the script with your desired options:

```
./build.sh --help
```

   This will display all available options and usage examples.

Common usage examples:

- Build with integration tests using OpenSearch:

```
./build.sh --integration-tests --use-opensearch
```

- Build with a specific Unomi distribution (automatically enables OpenSearch for tests if distribution contains "opensearch"):

```
./build.sh --integration-tests --distribution unomi-distribution-opensearch
```

- Build with OpenSearch using explicit distribution (both parameters work together):

```
./build.sh --integration-tests --use-opensearch --distribution unomi-distribution-opensearch-graphql
```

- Build in debug mode:

```
./build.sh --debug --debug-port 5006 --debug-suspend
```

- Deploy to specific Karaf instance:

```
./build.sh --deploy --karaf-home ~/apache-karaf
```

- Build without Karaf and auto-start OpenSearch:

```
./build.sh --no-karaf --auto-start opensearch
```

- Run a single integration test:

```
./build.sh --integration-tests --single-test org.apache.unomi.itests.graphql.GraphQLEventIT
```

- Debug a single integration test:

```
./build.sh --integration-tests --single-test org.apache.unomi.itests.graphql.GraphQLEventIT --it-debug --it-debug-suspend
```

- Run without colored output:

```
NO_COLOR=1 ./build.sh
```

or

```
export NO_COLOR=1
./build.sh
```

For a complete list of options and examples, run ./build.sh --help.

21.1.4. Updating the website

Use the top-level generate-manual.sh script to generate and publish the documentation website.

```
./generate-manual.sh publish <svn_user> <svn_pass>
./generate-manual.sh simulate <svn_user> <svn_pass>
```

Modes:

- publish: generates all documentation and publishes to Apache SVN
  - Generates exactly 2 versions (latest + stable)
  - Publishes HTML manual to $SVN_WEBSITE_BASE/manual and API docs from master
  - Uploads release packages (PDF/ZIP) to Apache Dist SVN for non-master branches
  - Removes old versions automatically on the website
- simulate: dry-run; prints the commands without making changes

Requirements:

- Java 17+, Maven 3.6+, Git, SVN client, bc
- Access to the master and the stable branch configured in generate-manual-config.sh

Outputs and locations:

- Staging directories under target/generated-docs/ (created by the build)
- Website content committed to $SVN_WEBSITE_BASE/manual/<version>
- Release artifacts (PDF/ZIP + signatures/checksums) committed to Apache Dist SVN at $SVN_DIST_BASE/<version>

Notes:

- The script sources optional generate-manual-config.sh and shell-utils.sh for configuration/utilities.
- Javadoc aggregation is attempted; if it fails (toolchain mismatch), the rest still publishes.

21.1.5. JGitFlow

You can use the JGitFlow Maven plugin to work with feature, hotfix and other types of branches.

For example, to start a feature branch, simply use:

```
mvn jgitflow:feature-start
```

This will prompt you for the feature name, and then create a feature branch and update all the POMs to have a version that contains the feature name. This makes it easier to integrate with continuous integration systems to generate builds for the feature branch.

Once the feature is completed you can use:

```
mvn jgitflow:feature-finish
```

To merge the branch into master.

21.1.6. Installing a Search Engine

Apache Unomi 4.x does not embed a search engine. Install Elasticsearch 9.x or OpenSearch 3.x as a standalone service. See Migrate from 2.x to 3.0 and Migrate from 3.0 to 4.0 for version requirements.

Option 1: Using Elasticsearch

1. Download Elasticsearch 9.4.3 from: https://www.elastic.co/downloads/past-releases/elasticsearch-9-4-3
2. Uncompress the downloaded package into a directory
3. In the config/elasticsearch.yml file, uncomment and modify the following line:

```
cluster.name: contextElasticsearch
```

+ 4. Launch the server using:

```
bin/elasticsearch (Mac, Linux)
bin\elasticsearch.bat (Windows)
```

Option 2: Using OpenSearch

The recommended way to run OpenSearch is using Docker Compose:

1. Create a docker-compose.yml file with the following content:

```
version: '3.8'
services:
  opensearch-node1:
    image: opensearchproject/opensearch:3.7.0
    environment:
      - cluster.name=opensearch-cluster
      - node.name=opensearch-node1
      - discovery.type=single-node
      - bootstrap.memory_lock=true
      - "OPENSEARCH_JAVA_OPTS=-Xms512m -Xmx512m"
      - OPENSEARCH_INITIAL_ADMIN_PASSWORD=${OPENSEARCH_INITIAL_ADMIN_PASSWORD:-
MyStrongPassw0rd!}
    ulimits:
      memlock:
        soft: -1
        hard: -1
      nofile:
        soft: 65536
        hard: 65536
    volumes:
      - opensearch-data1:/usr/share/opensearch/data
    ports:
      - 9200:9200
      - 9600:9600

volumes:
  opensearch-data1:
```

+ 2. Set up your Docker host environment: * **macOS & Windows**: In Docker *Preferences* > *Resources*, set RAM to at least 4 GB * **Linux**: Ensure vm.max_map_count is set to at least 262144

+ 3. Start OpenSearch using:

```
docker-compose up
```

Verify Installation

Check that your search engine is up and running by accessing: http://localhost:9200

For OpenSearch with security enabled, you may need to use https and provide the default credentials: - Username: admin - Password: admin (or the value of OPENSEARCH_INITIAL_ADMIN_PASSWORD if set)

21.1.7. Starting Unomi

After your search engine is running, you can start Unomi using the appropriate command:

```
# For Elasticsearch
unomi:start elasticsearch

# For OpenSearch
unomi:start opensearch
```



Make sure to specify which start features configuration you're using with the unomi:start command. The parameter determines which set of features and bundles are installed and started. Using just unomi:start without specifying the configuration is deprecated.

21.1.8. Deploying the generated binary package

The "package" sub-project generates a pre-configured Apache Karaf installation that is the simplest way to get started. Simply uncompress the package/target/unomi-VERSION.tar.gz (for Linux or Mac OS X) or package/target/unomi-VERSION.zip (for Windows) archive into the directory of your choice.

Apache Unomi ships no default administrator password, and bin/karaf refuses to start until one is set, so export both passwords first:

```
export UNOMI_ROOT_PASSWORD='choose-a-strong-password'
export UNOMI_HEALTHCHECK_PASSWORD='choose-a-strong-health-password'
```

You can then start the server simply by using the command on UNIX/Linux/MacOS X :

```
./bin/karaf
```

or on Windows shell :

```
set UNOMI_ROOT_PASSWORD=choose-a-strong-password
set UNOMI_HEALTHCHECK_PASSWORD=choose-a-strong-health-password
bin\karaf.bat
```



On Windows, karaf.bat does not check the exit code of setenv.bat, so a missing password is reported but startup continues anyway with an account whose password is empty. Set both variables before launching, and verify with curl -i -u "karaf:" http://localhost:8181/cxs/tenants, which must return 401.

You will then need to launch (only on the first Karaf start) the Apache Unomi packages using the following Apache Karaf shell command:

```
unomi:start
```

21.1.9. Deploying into an existing Karaf server

This is only needed if you didn't use the generated package. Also, this is the preferred way to install a development environment if you intend to re-deploy the context server KAR iteratively.

Additional requirements: * Apache Karaf 4.4.x (Unomi 4.0 ships with 4.4.11), Java 17+ — http://karaf.apache.org

Before deploying, make sure that you have Apache Karaf properly installed. Depending of your usage, you may also have to increase the memory size by adjusting the following environment values in the bin/setenv(.bat) files (at the end of the file):

```
MY_DIRNAME=`dirname $0`
MY_KARAF_HOME=`cd "$MY_DIRNAME/.."; pwd`
export KARAF_OPTS="$KARAF_OPTS -Xmx3G"
```

Install the WAR support and CXF into Karaf by doing the following in the Karaf command line:

```
feature:repo-add cxf-jaxrs 3.6.x (match the CXF version pinned in the Unomi Karaf features)
feature:repo-add mvn:org.apache.unomi/unomi-kar/VERSION/xml/features
feature:install unomi-kar
```

Create a new $MY_KARAF_HOME/etc/org.apache.cxf.osgi.cfg file and put the following property inside :

```
org.apache.cxf.servlet.context=/cxs
```

If all went smoothly, you should be able to access the context script here : http://localhost:8181/cxs/cluster . You should be able to login as karaf with the password you set in UNOMI_ROOT_PASSWORD and see basic server information. If not something went wrong during the install.

21.1.10. Installing GraphViz for Manual Generation

The manual project uses PlantUML diagrams which require GraphViz to be installed. Here's how to install it:

On macOS using Homebrew

```
brew install graphviz
```

On Linux (Debian/Ubuntu)

```
sudo apt-get install graphviz
```

On Linux (RHEL/CentOS/Fedora)

```
sudo dnf install graphviz
```

On Windows

1. Download the installer from https://graphviz.org/download/
2. Run the installer
3. Add the GraphViz bin directory to your PATH

Building the Manual

You can build the manual directly from the manual directory:

```
cd manual
mvn clean install
```

The build script will automatically detect and configure the GraphViz path. If you need to set it manually, you can:

1. Set it via environment variable:

```
export GRAPHVIZ_DOT=/path/to/dot
mvn clean install
```

+ 2. Or specify it directly to Maven:

```
mvn clean install -Dgraphviz.dot.path=/path/to/dot
```

The generated documentation will be available in: - HTML: target/generated-docs/html/latest/ - PDF: target/generated-docs/pdf/latest/

21.1.11. JDK Selection on Mac OS X

You might need to select the JDK to run the tests in the itests subproject. In order to do so you can list the installed JDKs with the following command :

```
/usr/libexec/java_home -V
```

which will output something like this :

```
Matching Java Virtual Machines (3):
   17.0.x, x86_64: "OpenJDK 17"    /Library/Java/JavaVirtualMachines/openjdk-17.jdk/Contents/Home
   1.8.0_181, x86_64:  "Java SE 8" /Library/Java/JavaVirtualMachines/jdk1.8.0_181.jdk/Contents/Home
   1.7.0_80, x86_64:   "Java SE 7" /Library/Java/JavaVirtualMachines/jdk1.7.0_80.jdk/Contents/Home

/Library/Java/JavaVirtualMachines/openjdk-11.jdk/Contents/Home
```

You can then select the one you want using :

```
export JAVA_HOME=`/usr/libexec/java_home -v 11.0.5`
```

and then check that it was correctly referenced using:

```
java -version
```

which should give you a result such as this:

```
openjdk version "11.0.5" 2019-10-15
OpenJDK Runtime Environment (build 11.0.5+10)
OpenJDK 64-Bit Server VM (build 11.0.5+10, mixed mode)
```

21.1.12. Running the integration tests

The integration tests are not executed by default to make build time minimal, but it is recommended to run the integration tests at least once before using the server to make sure that everything is ok in the build. Another way to use these tests is to run them from a continuous integration server such as Jenkins, Apache Gump, Atlassian Bamboo or others.

Note : the integration tests require a JDK **17** or more recent !

To run the tests simply activate the following profile:

```
mvn -P integration-tests clean install
```

Selecting the Search Engine for Integration Tests

By default, integration tests target Elasticsearch. To run them against OpenSearch, you can use either:

- **Using the build script** (recommended):

  ```
  ./build.sh --integration-tests --use-opensearch
  ```

  Or specify the distribution directly (automatically enables OpenSearch if distribution contains

"opensearch"):

```
./build.sh --integration-tests --distribution unomi-distribution-opensearch
```

- **Using Maven directly**: Set the activation property used by the OpenSearch profile (no additional -P flags are required):

  ```
  mvn -P integration-tests -Duse.opensearch=true clean install
  ```

  You can also set the Unomi distribution system property:

  ```
  mvn -P integration-tests -Dunomi.distribution=unomi-distribution-opensearch clean install
  ```

21.1.13. Testing with an example page

A default test page is provided at the following URL:

```
http://localhost:8181/index.html
```

This test page loads /cxs/context.js, which retrieves or creates a user context. On Unomi 4.0, public context endpoints require a tenant public API key (X-Unomi-Api-Key) unless single-tenant compatibility mode is enabled. Create a tenant and regenerate keys first (see 5-minute quickstart or Multi-tenancy). The page also contains an experimental Facebook Login integration that does not yet save the context back to the server.

21.2. Platform upgrades

This section is for Apache Unomi **maintainers** upgrading runtime platform components: Apache Karaf, JDK, CXF/Jackson/Jetty, Elasticsearch and OpenSearch clients, Log4j, integration tests, and application dependency hygiene.

Procedures are derived from git history (UNOMI-216, 491, 876, 899, 901, 828, 921, 875) and Unomi 4.0 platform upgrade work, cross-checked against Apache JIRA (see JIRA reference).

21.2.1. Overview and shared rules

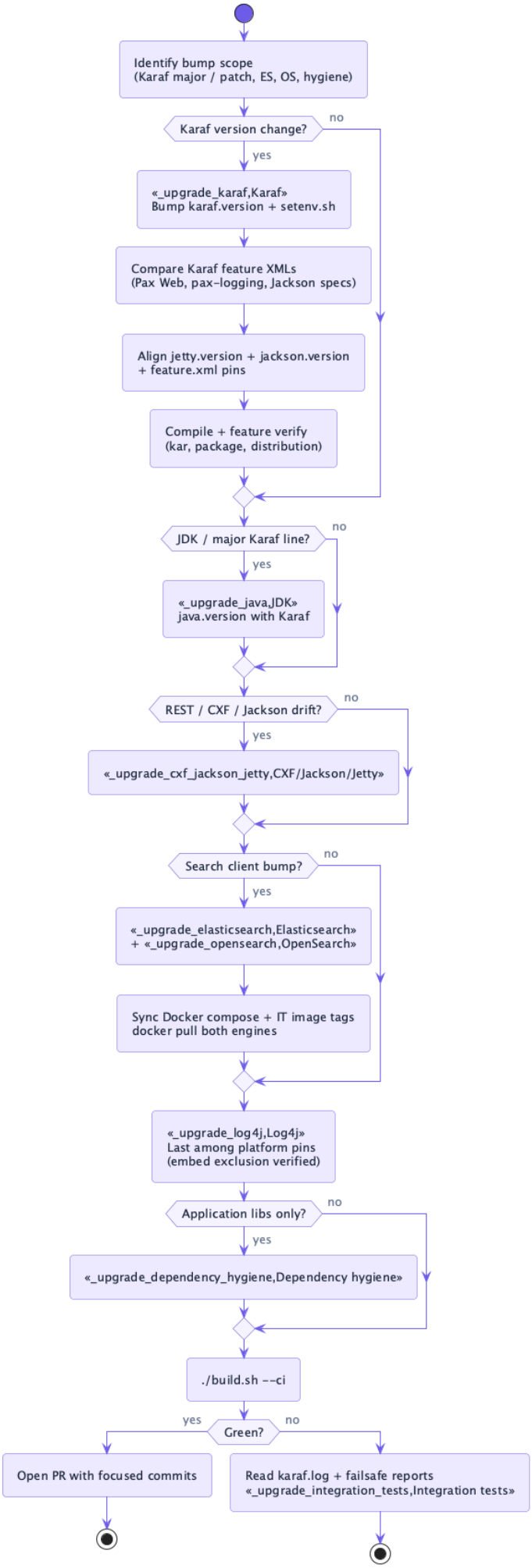
Guide index

Component	Guide	When to use
Overview	This section	PR planning, shared OSGi rules, version pins, upgrade ordering

Component	Guide	When to use
Apache Karaf	Karaf upgrade	Major vs patch Karaf bumps, feature verify, framework cascade
JDK	JDK upgrade	Java version changes (with major Karaf migrations)
CXF / Jackson / Jetty	CXF/Jackson/Jetty	Libraries that cascade from Karaf but are Unomi-pinned
Elasticsearch	Elasticsearch upgrade	elasticsearch.version bump, OSGi embed, Java API changes
OpenSearch	OpenSearch upgrade	opensearch.version bump, Docker tag verification
Log4j	Log4j upgrade	${log4j.version} — always with persistence embed checks
Integration tests	Integration tests	IT/Docker failures, Pax Exam static init, log reading
Dependency hygiene	Dependency hygiene	npm, kafka, commons — after platform pins are stable
JIRA reference	JIRA reference	Epic UNOMI-875, blockers, ticket-to-guide map, acceptance criteria

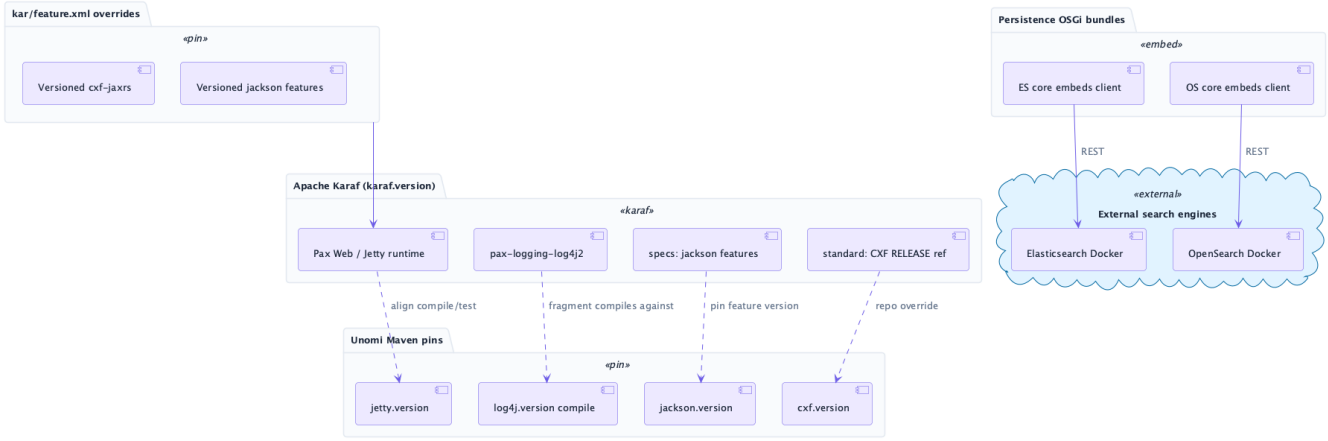
Platform upgrade workflow

Recommended order when bumping multiple platform components in one release cycle:


Identify bump scope
(Karaf major / patch, ES, OS, hygiene)
Karaf version change?
no
yes
«_upgrade_karaf,Karaf»
Bump karaf.version + setenv.sh
Compare Karaf feature XMLs
(Pax Web, pax-logging, Jackson specs)
Align jetty.version + jackson.version
+ feature.xml pins
Compile + feature verify
(kar, package, distribution)
JDK / major Karaf line?
no
yes
«_upgrade_java,JDK»
java.version with Karaf
REST / CXF / Jackson drift?
no
yes
«_upgrade_cxf_jackson_jetty,CXF/Jackson/Jetty»
Search client bump?
no
yes
«_upgrade_elasticsearch,Elasticsearch»
+ «_upgrade_opensearch,OpenSearch»
Sync Docker compose + IT image tags
docker pull both engines
«_upgrade_log4j,Log4j»
Last among platform pins
(embed exclusion verified)
Application libs only?
no
yes
«_upgrade_dependency_hygiene,Dependency hygiene»
./build.sh --ci
yes
Green?
no
Open PR with focused commits
Read karaf.log + failsafe reports
«_upgrade_integration_tests,Integration tests»

Platform component relationships

Karaf is the hub: most runtime libraries come from Karaf features, while Unomi Maven properties pin compile-time and feature-verify alignment.



Current version pins (Unomi 4.0 baseline)

Property	Value	Primary files
java.version	17	Root pom.xml, package/pom.xml (javase)
karaf.version	4.4.11	pom.xml, setenv.sh, itests/pom.xml, */karaf-kar/pom.xml
cxf.version	3.6.8	pom.xml, bom/pom.xml, kar/.../feature.xml
jackson.version	2.18.3	pom.xml, bom/pom.xml, kar/.../feature.xml
jetty.version	9.4.58.v20250814	pom.xml, graphql/karaf-feature/pom.xml, itests/pom.xml
elasticsearch.version	9.4.3	pom.xml, bom/pom.xml, Docker ES compose files
elasticsearch.test.version	9.4.3	pom.xml, IT Docker ES image in itests/pom.xml
opensearch.version	3.7.0	pom.xml, bom/pom.xml, OS Docker compose, IT OS image
opensearch.rest.client.version	3.7.0	pom.xml (keep in sync with opensearch.version)
log4j.version	2.24.0	pom.xml (compile-time for fragments; runtime from pax-logging)



Maven client version, IT Docker image, and docker/ compose tags must match for each search engine. Verify with docker pull before opening a pull request.



${log4j.version}, ${jackson.version}, and ${jetty.version} are **Unomi Maven pins**. Runtime versions for logging and HTTP come from Karaf/pax-logging/Pax Web — reconcile after every Karaf bump.

Shared OSGi rules (Elasticsearch and OpenSearch persistence)

1. **Embed** the client and HTTP/JSON stack: Embed-Dependency *;scope=compile|runtime, Embed-Transitive true.
2. **Exclude log4j from embed:** !groupId=org.apache.logging.log4j;artifactId=* (see Log4j upgrade).
3. **Optional Import-Package** for codecs (brotli, zstd, netty) after httpclient5 5.6 — do **not** add compile Maven dependencies for native codecs.
4. **Provided** for platform libraries: slf4j-api, Jackson 2 (com.fasterxml.jackson.*), commons, Unomi API modules.
5. **Export** Elasticsearch query DSL types from the core bundle only; the conditions bundle imports exports — do not re-embed the client.
6. Verify OSGi resolution, not just compilation:

```
mvn -pl persistence-elasticsearch/core,persistence-opensearch/core install -DskipTests
mvn -pl package -DskipTests package
./build.sh --ci
```

Recommended pull request structure

Split platform work into focused commits when possible:

Slice	Contents	Rationale
Karaf + cascade libs	karaf.version, setenv.sh, jetty.version, jackson.version, feature.xml pins, compile-fix POMs	Distribution-wide; isolate from search clients
ES + OS + Docker	Version pins, persistence POMs, compose files, Java API fixes	Large OSGi and Docker surface
Log4j last	${log4j.version} after embed exclusion verified	Log4j-only bump breaks ES bundle without embed fix
Hygiene	npm, kafka, commons (Track E1/E2)	Lower OSGi risk once platform is green

Git history reference

Ticket / PR	Commit	Era	Key lesson
UNOMI-216	b873b35a2	Karaf 4.1.7	Property placeholders in etc require Karaf >= 4.1.6
UNOMI-491	dda24b66e	Karaf 4.2.11	Do not duplicate Jackson in WAB; reactivate features after graph changes

Ticket / PR	Commit	Era	Key lesson
UNOMI-876	da9c57fd2	Karaf 4.4.8 + JDK 17	feature.xml v1.6; platform deps provided; ~187 POM changes; WAB/Pax Web
UNOMI-899	8cff171ef	CXF 3.6.8	Karaf Jackson features; OpenAPI/Swagger path; pin CXF version in feature.xml
UNOMI-901	d3d2d16c4	ES 9 client	Export co.elastic.clients.*; optional httpclient5 imports
UNOMI-828	078ebceda	OpenSearch module	Lean embed + optional imports; dual-engine IT matrix
UNOMI-921	5a5aedba7	Docker IT	Replace elasticsearch-maven-plugin with docker-maven-plugin
UNOMI-919	a0a08e4ad	Startup feature	Boot feature graph changes — update package verify lists
PR2 #795	ef10dfe15	log4j 2.24	Embed without exclusion breaks Felix — reverted
PR3 #796	545f6b8af…	Karaf 4.4.11 / ES 9.4 / OS 3.7	slf4j-api; Jackson/Jetty cascade; codec optional imports; IT lazy JSON mapper (no static init in probe)

Pre-merge checklist

- Compared Karaf feature XMLs for Pax Web, pax-logging, Jackson specs (patch/minor Karaf bumps)
- ${jetty.version} matches Pax Web Jetty bundles for ${karaf.version}
- Jackson features version-pinned in feature.xml; ${jackson.version} exists in Karaf specs
- ./build.sh --ci green (or unit tests + both IT profiles)
- After Karaf bump: no static initializers in BaseIT that touch Jackson/Unomi types before features are up; IT JSON via lazy instance getObjectMapper() (see Pax Exam probe & static init)
- docker pull succeeds for ES and OS IT images at pinned versions
- ES/OS core manifests: no embedded log4j / brotli4j / zstd-jni JARs
- setenv.sh KARAF_VERSION matches pom.xml
- Docker README and compose tags match version properties

Quick diagnostics

```
rg
'karaf\.version|jackson\.version|jetty\.version|cxf\.version|elasticsearch\.version|opensearch\.vers
ion|log4j\.version' pom.xml setenv.sh kar/src/main/feature/feature.xml
mvn -pl package -DskipTests package 2>&1 | rg -i 'Unable to resolve|BUILD FAILURE'
docker pull docker.elastic.co/elasticsearch/elasticsearch:9.4.3
docker pull opensearchproject/opensearch:3.7.0
```

21.2.2. JIRA reference (platform upgrades)

This section maps Apache Unomi JIRA tickets to the upgrade runbooks in this chapter. Use it when you need **why** a constraint exists (Cellar removal, CXF 3.x ceiling, ES client decoupling) or when tracing work under the Unomi 3 epic.

JIRA: UNOMI project

Unomi 3 epic

UNOMI-875 (**Unomi Version 3**) groups most 3.x platform and product work. Karaf was initially deferred from the first 3.0 release because the cascade (Java 17, CXF, Cellar removal, WAB/Pax Web) would delay shipping again — see comments on UNOMI-898.

Platform-relevant subtasks include:

Ticket	Summary	Upgrade guide
UNOMI-876	Karaf 4.4.7+ (major migration)	Karaf, JDK
UNOMI-877	Remove Karaf Cellar / Hazelcast PersistenceService clustering	Karaf (prerequisite)
UNOMI-828	OpenSearch persistence module	OpenSearch
UNOMI-901	HLRC elasticsearch-java client	Elasticsearch
UNOMI-921	Docker-based ES in ITs (replace elasticsearch-maven-plugin)	Integration tests
UNOMI-887	Enhanced build.sh	Integration tests
UNOMI-884	V3 **data** migration scripts (audit fields, multi-tenancy)	Manual migration chapter — **not** Maven version pins
UNOMI-829	General dependency upgrade strategy	Dependency hygiene



UNOMI-884 covers **index/data migration** for existing deployments. Platform upgrade guides cover **runtime and client versions** (Karaf, ES/OS clients, Log4j). Do not conflate the two.

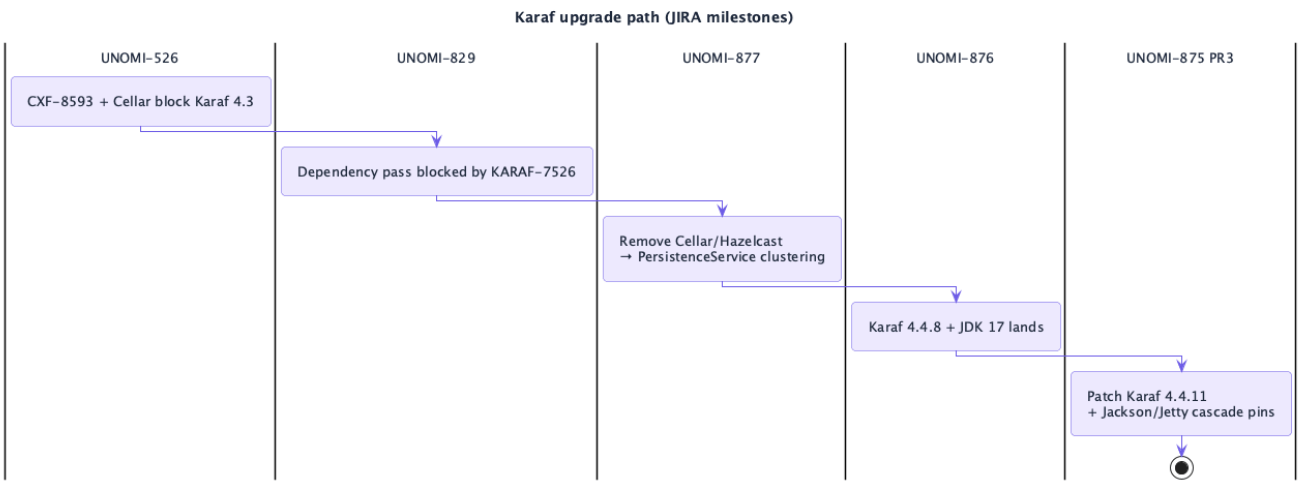
Ticket map by component

Ticket	Status (2026)	Topic	Maintainer takeaway
UNOMI-526	Closed	Karaf 4.3 / CXF 3.4.5 / Jackson 2.12	Blocked by CXF-8593 and Karaf Cellar incompatibility with Karaf 4.3
UNOMI-829	Closed	Dependency hygiene	30-minute timebox per library; maintenance-branch strategy; Karaf blocked by KARAF-7526 until Cellar removed
UNOMI-838	Closed	CXF + jakarta.servlet	Superseded by Karaf 4.4 migration; GraphQL still on javax — see UNOMI-667
UNOMI-851	Closed	ES HLRC 7.17	Duplicate of UNOMI-901; shipped as ES 9 + elasticsearch-java
UNOMI-898	Closed	Karaf 4.2.15 4.4.7	Duplicate of UNOMI-876
UNOMI-70	Closed	ES 5.x externalization	Start of external ES server; no embedded ES in modern Unomi
UNOMI-225	Closed	ES 7 / REST HLRC	Predecessor path to UNOMI-901
UNOMI-531	Open	IT matrix for multiple ES versions	Partially addressed by Docker IT properties; see Integration tests
UNOMI-667	In Progress	GraphQL OSGi / java.io imports	Future hygiene risk when bumping graphql-java — not a Karaf blocker today
UNOMI-889	Sub-task of 875	Remove OGNL scripting	Called out in UNOMI-876 Karaf plan (Camel 4.x line)

Historical blockers (Karaf)

Karaf upgrade prerequisites (timeline)

Historical JIRA blockers explain why Karaf 4.3+ waited until Unomi 3 clustering changed:



Before Unomi 3 could move to Karaf 4.3+, two recurring blockers appeared in JIRA:

1. **Karaf Cellar** — KARAF-7526 / Cellar 4.2.1 not compatible with Karaf 4.3 (UNOMI-526, UNOMI-829). Resolved by UNOMI-877 (PersistenceService-based clustering).
2. **CXF OSGi** — CXF-8593 blocked Jackson/CXF/Karaf alignment on the 4.3 line. Unomi 3 landed on Karaf 4.4 with **CXF 3.6.x** pinned in feature.xml (UNOMI-899).

UNOMI-876 documents a critical decision repeated in JIRA comments: **CXF 4.x removes OSGi support** and is unsuitable for Karaf. Target CXF 3.6.5+ (Unomi currently pins 3.6.8), not CXF 4.

UNOMI-876 planned vs shipped

Serge Huber's Karaf upgrade report on UNOMI-876 compares Karaf 4.2.15 vs 4.4.7 binary contents. Use it as a **checklist**, then verify against the **current** ${karaf.version} feature XMLs (see Karaf).

Component	JIRA target (4.4.7 era)	Unomi 4.0 baseline	Notes
Java	17	17	Shipped — commit da9c57fd2
Karaf	4.4.7 → later 4.4.11	4.4.11	Patch bumps need cascade pins (PR3)
CXF	3.6.5 (not 4.x)	3.6.8	Repo override in feature.xml
Pax Web / Jetty	8.0.30 / Jetty 9.4.x	Pax Web 8.0.35 / Jetty 9.4.58	Align ${jetty.version} after each Karaf bump
Camel	4.9.0 (bundled with Karaf 4.4.7)	**2.23.1** in extensions/router/pom.xml	Router extension still ships Camel 2.x OSGi routes; Karaf may expose Camel 4 features — do not assume JIRA matrix matches router code without checking
Jackson	~2.15.x in report	2.18.3 pinned	Pin <feature version="..."> on 4.4.11+
Servlet API	4.0.1 (javax, not jakarta)	javax.servlet in WAB/Pax Web 8	UNOMI-838 jakarta migration deferred

Elasticsearch client strategy (UNOMI-901)

UNOMI-901 acceptance criteria: remove HLRC, adopt co.elastic.clients:elasticsearch-java, refactor persistence code, green ITs.

JIRA comment (decoupling option): if JDK 17 + Karaf upgrade is too heavy for a **2.x** maintenance branch, Elastic's ES 8 Java API client (Java 8+) can talk to ES 9 clusters as an interim step — but **Karaf upgrades belong on Unomi 3**, not 2.x.

For Unomi 3 platform work, treat ES client, IT Docker tag, and compose files as one atomic bump (Elasticsearch).

Dependency upgrade strategy (UNOMI-829)

When doing non-platform library bumps (Dependency hygiene), follow the process documented on UNOMI-829:

1. Prefer removing unused libraries over upgrading.
2. Timebox each failed upgrade to **30 minutes**; if harder, file a follow-up and move on.
3. Prefer latest release; if breaking, use latest on same major maintenance branch and open a ticket for the next major.
4. Do **not** mix complex upgrades (Karaf, CXF, GraphQL OSGi) into a bulk hygiene pass.

Integration test tickets

Ticket	Intent	Runbook
UNOMI-921	Replace elasticsearch-maven-plugin; remove default_template workaround; Docker parity with OpenSearch	Integration tests
UNOMI-937	Stabilize ITs; align GitHub Actions with build.sh	Integration tests
UNOMI-944	IT developer tooling (archival, diff, live Karaf)	itests/README.md
UNOMI-946	Cluster health / replica checks in IT harness	Migration and dual-engine IT matrix

Searching JIRA for upgrade context

Useful JQL examples (Apache JIRA **Advanced** search):

```
project = UNOMI AND summary ~ "Karaf" ORDER BY updated DESC
project = UNOMI AND summary ~ "upgrade" AND status != Closed
project = UNOMI AND parent = UNOMI-875
project = UNOMI AND text ~ "elasticsearch-maven-plugin"
```

When a ticket and git history disagree, prefer **git commits** for exact file changes and **JIRA** for intent, blockers, and acceptance criteria.

21.2.3. Upgrading Apache Karaf

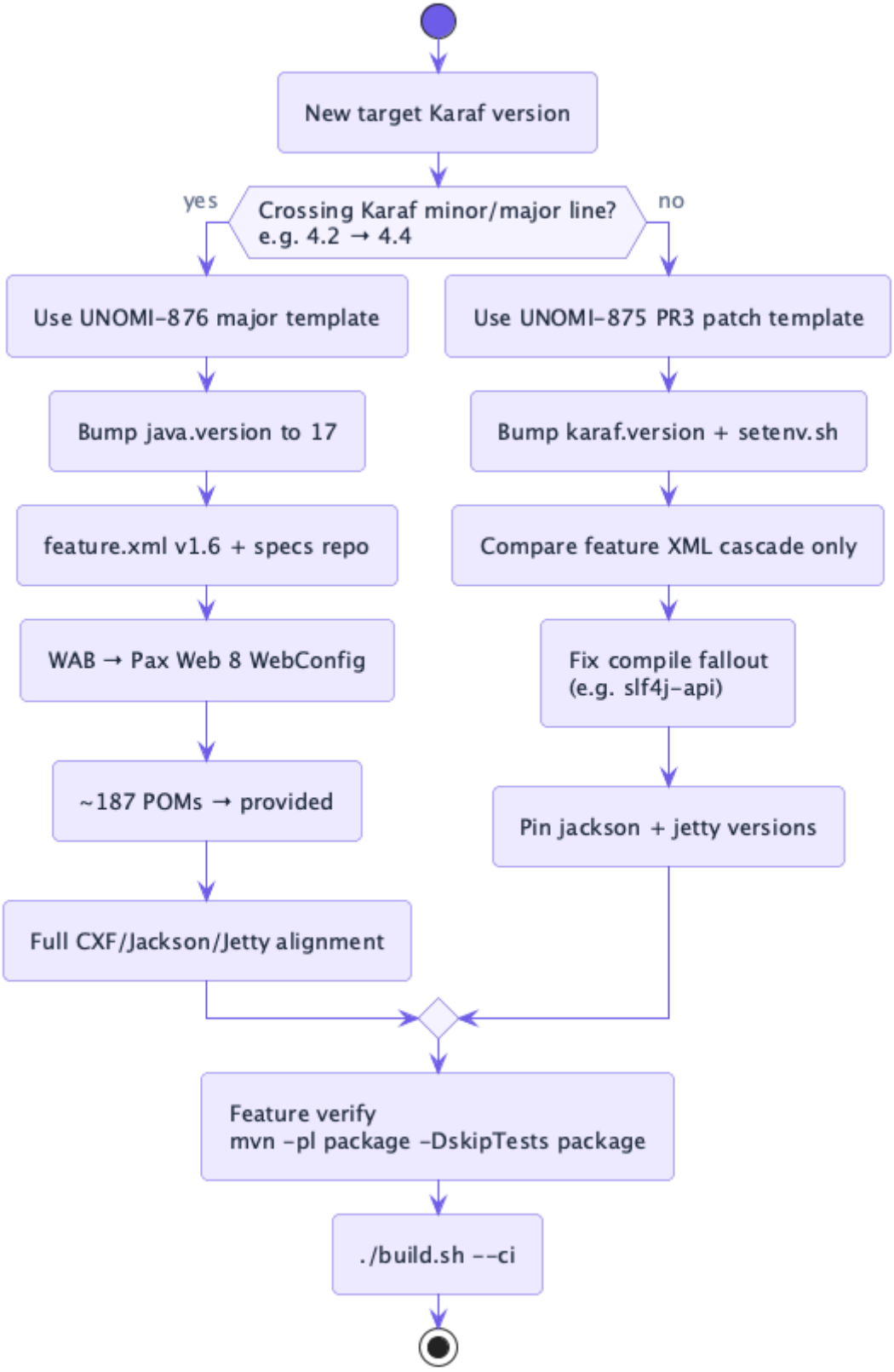
Karaf upgrades touch the **whole distribution**, not a single module. Scope depends on whether you are doing a **major**, **minor**, or **patch** Karaf bump.



JIRA context for prerequisites (Cellar removal, CXF 3.x ceiling, UNOMI-876 version matrix): JIRA reference.

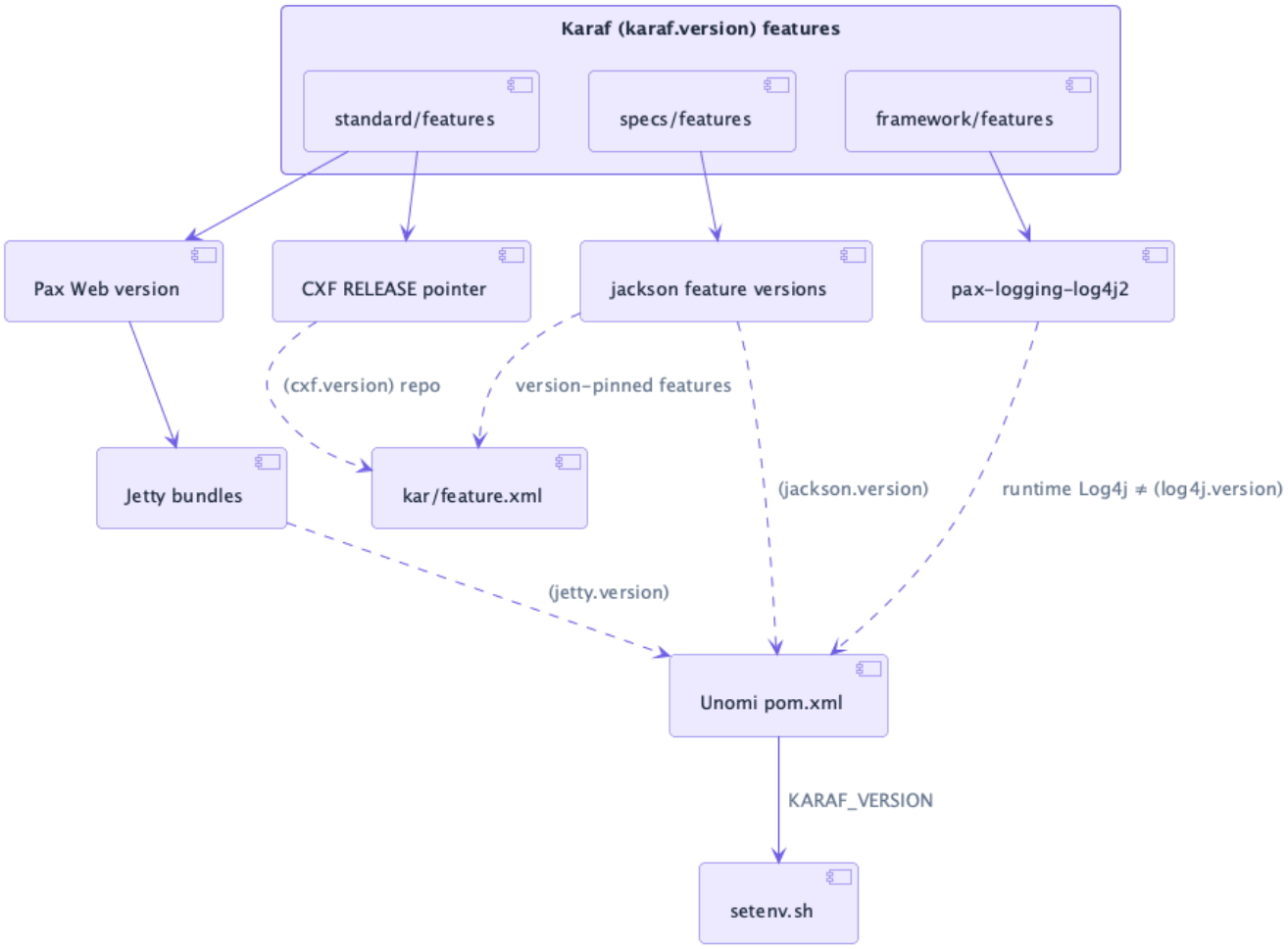
Bump type	Example	Typical Unomi effort
Major	4.2.15 → 4.4.8 (UNOMI-876)	JDK, feature namespace, WAB/Pax Web, ~187 POMs, CXF/Jackson/Jetty, full IT matrix
Minor / patch	4.4.8 → 4.4.11 (UNOMI-875 PR3)	karaf.version, cascade library pins, compile fallout (e.g. slf4j), feature verify, IT probe fixes
Config-only	UNOMI-919 startFeatures feature	Feature graph / cfg — not a Karaf version bump

Choosing major vs patch Karaf path



Framework cascade from Karaf feature XMLs

When ${karaf.version} changes, inspect upstream feature repositories before merging — patch releases move Pax Web, pax-logging, and Jackson specs independently of Unomi code.



Framework cascade (every Karaf bump)

Even patch releases can move platform libraries inside Karaf feature XMLs:

Component	4.4.8 example	4.4.11 example	Unomi action
Pax Web	8.0.33	8.0.35	Align ${jetty.version} — see CXF/Jackson/Jetty
pax-logging	2.3.0	2.3.3	Runtime Log4j version changes — see Log4j
Jackson specs	single 2.18.2 feature	2.18.3 **and** 2.21.2 features	Pin ${jackson.version} on <feature> — see CXF/Jackson/Jetty
CXF	RELEASE pointer	RELEASE pointer	Keep ${cxf.version} repo override (UNOMI-899)

Compare feature XMLs before merging:

```
KVER=4.4.11
for repo in framework standard specs enterprise; do
  echo "=== $repo ==="
  curl -sL "https://repo1.maven.org/maven2/org/apache/karaf/features/${repo}/${KVER}/${repo}-
${KVER}-features.xml" \
    | rg -i 'pax-web|pax-logging|jackson-core|jetty-server' | head -8
done
```

Version pins to update

Required (every Karaf bump):

File	Property / variable
Root pom.xml	<karaf.version>
setenv.sh	export KARAF_VERSION=...

Everything else should reference ${karaf.version}: bom/pom.xml, kar/pom.xml, package/pom.xml, distribution/pom.xml, itests/pom.xml, extension and GraphQL Karaf feature modules.

Search for hardcoded old versions:

```
rg '4\.4\.[0-9]+' --glob '!**/target/**' pom.xml setenv.sh package kar itests extensions graphql
```

Co-bump after comparing Karaf feature XMLs (patch/minor):

- ${jetty.version} — match Pax Web Jetty bundles
- ${jackson.version} — must exist as a Karaf specs feature version; pin in feature.xml
- ${cxf.version} — only if REST/Karaf release notes require it (major jumps usually do)

Co-bump on major Karaf migration only:

- java.version — see JDK upgrade
- Feature namespace (v1.6.0), specs repository, WAB servlet wiring — UNOMI-876

Prerequisites (JIRA)

Karaf bumps were blocked for years until clustering moved off Karaf Cellar:

- UNOMI-877 — replace Cellar/Hazelcast with PersistenceService-backed sync (required before 4.3+).
- KARAF-7526 — Cellar incompatibility cited in UNOMI-526 and UNOMI-829.
- CXF-8593 — blocked Karaf 4.3 / Jackson 2.12 alignment until the 4.4 + CXF 3.6 path.

Do not target CXF 4.x for Unomi on Karaf — UNOMI-876 notes OSGi support was removed in CXF 4.

Major Karaf migration (UNOMI-876 template)

Use this checklist when crossing Karaf minor/major lines (e.g. 4.2 → 4.4):

1. Read Karaf release notes and Java requirement.
2. Bump ${karaf.version} + java.version + setenv.sh.
3. Update kar/src/main/feature/feature.xml:
   - xmlns features v1.6.0
   - <repository>mvn:org.apache.karaf.features/specs/${karaf.version}/xml/features</repository>
   - CXF repo + versioned cxf-jaxrs
   - Version-pinned jackson / jackson-jaxrs
4. Migrate WAB servlets to Pax Web 8 programmatic registration (WebConfig.java).
5. Mass POM update: platform deps → scope=provided (~187 modules in UNOMI-876).
6. Update package/pom.xml boot/verify feature lists if graph changed (UNOMI-919: unomi-startup feature).
7. IT harness: Pax Exam instanceStartupTimeout, Jacoco, javase in verify config.
8. Full CXF/Jackson/Jetty alignment.
9. ./build.sh --ci

Minor / patch Karaf migration (UNOMI-875 PR3 template)

Use when staying on the same Karaf line (e.g. 4.4.8 → 4.4.11):

1. Bump ${karaf.version} and setenv.sh only as the first commit slice.
2. Run framework cascade comparison (Pax Web, pax-logging, Jackson specs).
3. Update ${jetty.version} and ${jackson.version} + pin Jackson features in feature.xml.
4. Compile all modules — fix missing provided deps (PR3: slf4j-api in lifecycle-watcher):

```
mvn -DskipTests compile 2>&1 | rg 'cannot find symbol|package .* does not exist|slf4j'
```

1. Feature verify (mandatory):

```
mvn -pl extensions/groovy-actions/karaf-kar -am -DskipTests package
mvn -pl package -DskipTests package
mvn -pl kar,distribution -DskipTests package
```

1. Check Integration tests for probe static-init / Docker image issues.
2. ./build.sh --ci

Do **not** skip cascade library alignment — patch Karaf releases have changed Pax Web, pax-logging, and Jackson specs without changing Unomi application code.

Lessons from past upgrades

UNOMI-216 (b873b35a2) — Karaf 4.1.7: ${env:...} placeholders in custom.system.properties need Karaf >= 4.1.6. Smoke-test property interpolation after any bump.

UNOMI-491 (dda24b66e) — Karaf 4.2.11: Moved Jackson YAML out of WAB; reactivated features in package verify. Do not embed platform Jackson in web modules.

UNOMI-876 (da9c57fd2) — Karaf 4.4.8 + JDK 17: Feature namespace v1.6; removed manual Jackson/Guava/SnakeYAML bundles; WAB WebConfig.java; ~187 POMs to provided; CXF 3.3 3.6.5.

UNOMI-899 (8cff171ef) — CXF 3.6.8 + Karaf Jackson features: Independent CXF bump; switched to Karaf jackson features (must be version-pinned on 4.4.11+).

UNOMI-919 (a0a08e4ad) — startFeatures: Refactored startup to a Karaf feature; adjust package/pom.xml verify lists when changing boot order.

PR3 / UNOMI-875 (545f6b8af) — 4.4.8 4.4.11: lifecycle-watcher lost transitive slf4j-api; Pax Web 8.0.35 Jetty 9.4.58; Jackson specs duplication pin ${jackson.version}; Pax Exam IT fix — remove static ObjectMapper init in BaseIT, use lazy instance getObjectMapper() CustomObjectMapper.getCustomInstance() (types from unomi-base, not Karaf boot) — see Pax Exam probe & static init.

Files to inspect

Area	Paths
Feature graphs	kar/src/main/feature/feature.xml, distribution/src/main/feature/feature.xml, extension Karaf features
Package verify	package/pom.xml — karaf-maven-plugin verify feature list
Runtime config	package/src/main/resources/etc/*.cfg, custom.system.properties
Logging	org.ops4j.pax.logging.cfg, extensions/log4j-extension/
IT container	itests/pom.xml, itests/.../BaseIT.java

Anti-patterns

Don't	Do instead	History
Embed platform libs in every bundle	scope=provided + Karaf features	UNOMI-876, PR2 log4j
Use unversioned <feature>jackson</feature> on 4.4.11+	<feature version="${jackson.version}">jackson</feature>	PR3 cascade

Don't	Do instead	History
Assume a Karaf patch is "just pom.xml"	Cascade pins + compile + feature verify	PR3 lifecycle-watcher, Jetty, Jackson
Skip ITs after Karaf bump	Run ./build.sh --ci	UNOMI-876

21.2.4. Upgrading the JDK

A JDK upgrade is almost always bundled with a **major** Karaf migration. Unomi 3.x baseline is **Java 17** (UNOMI-876, da9c57fd2).

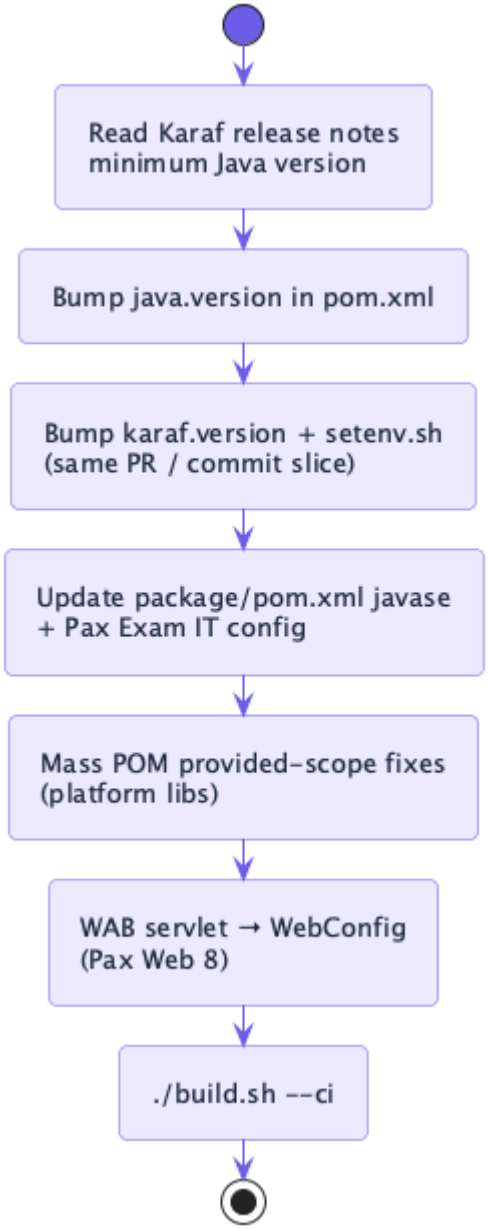
Version pin

Property	Value	Files
java.version	17	Root pom.xml (maven.compiler.source/target/release, karaf-maven-plugin javase in package/pom.xml)

Also update CI workflows, build.sh preflight checks, and contributor docs when raising the minimum JDK.

JDK upgrade coupled to Karaf (diagram)

Java version changes are not done in isolation on Unomi 3.x — they ride the major Karaf migration path (UNOMI-876).



UNOMI-876 checklist (Java 11 → 17)

Area	Action
Maven compiler	<java.version>17</java.version>; remove obsolete Java 11 --add-opens workarounds
Karaf distribution	Karaf 4.4.x requires Java 17 — bump ${karaf.version} in the same effort
Module POMs	~187 modules: platform libraries → scope=provided; fix cannot find symbol for removed JDK internals
WAB / Pax Web	Blueprint servlet registration → programmatic WebConfig.java (Pax Web 8)
IT harness	Pax Exam javase 17; Jacoco 0.8.13+; timeout property renames
Removed APIs	OGNL scripting removed from defaults; javax → jakarta where applicable

Procedure

1. Confirm target Karaf version's minimum Java from Karaf download page.
2. Bump java.version and ${karaf.version} together (see Karaf upgrade).
3. Full compile: mvn -DskipTests compile
4. Feature verify + ./build.sh --ci
5. Update GitHub Actions Java version matrix if present.

Anti-patterns

Don't	Do instead
Bump JDK without Karaf compatibility check	Read Karaf release notes; 4.4.x needs 17
Embed JDK-specific libraries in bundles	Use Karaf specs / provided scope
Skip ITs after JDK change	Run full ./build.sh --ci — bytecode and Pax Exam both shift

21.2.5. CXF, Jackson, and Jetty (Karaf cascade libraries)

Karaf upgrades change the **runtime** versions of several frameworks even when you only bump ${karaf.version}. Unomi pins some of these in root pom.xml and kar/src/main/feature/feature.xml so compile-time, feature verify, and OSGi resolution stay aligned.

This guide covers libraries that **cascade from Karaf** but are **owned by Unomi** for REST, WAB, GraphQL, and ITs.

Version pins (Unomi 4.0 baseline)

Property	Value	Primary consumers
cxf.version	3.6.8	kar/.../feature.xml, rest/, bom/pom.xml, OpenAPI/Swagger
jackson.version	2.18.3	bom/pom.xml, kar/.../feature.xml (pinned features + extra modules)
jetty.version	9.4.58.v20250814	graphql/karaf-feature/pom.xml, itests/pom.xml (compile/test only)

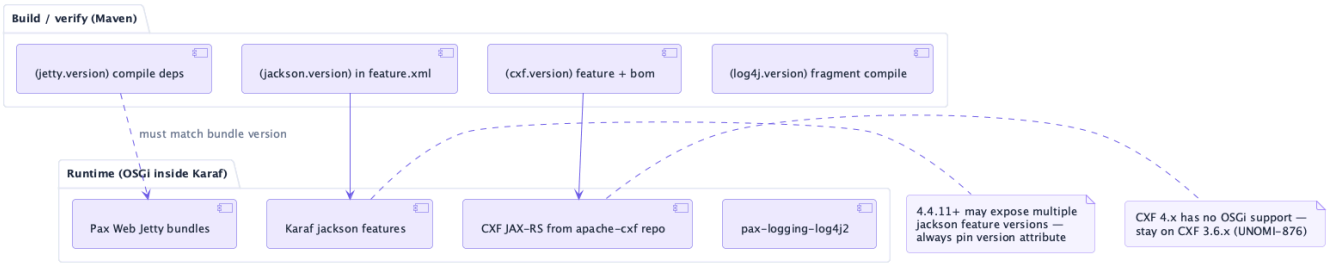


${jetty.version} must match the Jetty bundles shipped by the Pax Web version bundled in your ${karaf.version}. Inspect pax-web-features-*-features.xml on Maven Central — do not guess from release notes alone.

How Karaf supplies vs how Unomi overrides

Library	Karaf source	Unomi override
CXF	standard features reference apache-cxf/RELEASE	Explicit repo + versioned feature in feature.xml (UNOMI-876, UNOMI-899)
Jackson	specs features (jackson, jackson-jaxrs) — multiple versions on 4.4.11+	Pin <feature version="${jackson.version}"> — never unversioned (PR3 lesson)
Jetty	Pax Web bundles (runtime)	${jetty.version} for Maven compile/test deps only
Log4j (runtime)	pax-logging-log4j2 bundle	See Log4j upgrade — ${log4j.version} is compile-time for fragments

Karaf supply vs Unomi override (diagram)



CXF upgrade (UNOMI-899, 8cff171ef)

When: REST/OpenAPI breakage, Karaf major jump (4.2 → 4.4), or explicit CXF security release.

Steps:

1. Bump <cxf.version> in root pom.xml (managed via bom/pom.xml).
2. Confirm kar/src/main/feature/feature.xml still has:

```
<repository>mvn:org.apache.cxf.karaf/apache-cxf/${cxf.version}/xml/features</repository>
...
<feature version="${cxf.version}" dependency="true">cxf-jaxrs</feature>
```

1. Rebuild REST modules and run feature verify (mvn -pl package -DskipTests package).
2. Regenerate or smoke-test OpenAPI/Swagger UI (unomi-rest-ui feature) if JAX-RS annotations changed.

History: UNOMI-876 moved CXF 3.3.11 → 3.6.5 with Karaf 4.4.8. UNOMI-899 bumped 3.6.5 → 3.6.8 independently and switched Jackson delivery to Karaf features.

Jackson upgrade

Risk: Karaf 4.4.11+ ships **multiple** jackson feature versions (e.g. 2.18.3 and 2.21.2). An unversioned <feature>jackson</feature> may resolve to the newest, while Unomi still installs datatype/jaxb modules at ${jackson.version} — causing mixed Jackson on the OSGi classpath.

Required pattern in kar/src/main/feature/feature.xml:

```
<feature version="${jackson.version}">jackson</feature>
<feature version="${jackson.version}">jackson-jaxrs</feature>
<bundle>mvn:com.fasterxml.jackson.datatype/jackson-datatype-jsr310/${jackson.version}</bundle>
<bundle>mvn:com.fasterxml.jackson.module/jackson-module-jaxb-
annotations/${jackson.version}</bundle>
```

Procedure:

1. Check which jackson feature versions exist for your ${karaf.version}:

```
curl -sL
"https://repo1.maven.org/maven2/org/apache/karaf/features/specs/${KARAF_VERSION}/specs-
${KARAF_VERSION}-features.xml" \
  | rg 'feature name="jackson"'
```

1. Set ${jackson.version} to a version that exists in Karaf **specs** (not necessarily the highest).
2. Bump bom/pom.xml managed Jackson artifacts if needed.
3. Do **not** embed Jackson in WABs or persistence bundles — scope=provided (UNOMI-491, UNOMI-876).

Major Karaf migration note: UNOMI-876 initially listed every Jackson JAR in feature.xml; UNOMI-899 moved to Karaf features **with** version pins. Do not revert to unversioned features.

Jetty alignment after Karaf bump

Pax Web version is tied to ${karaf.version}. Example: Karaf 4.4.11 → Pax Web 8.0.35 → Jetty 9.4.58.v20250814.

1. Inspect Pax Web features for your Karaf version:

```
curl -sL
"https://repo1.maven.org/maven2/org/apache/karaf/features/standard/${KARAF_VERSION}/standard-
${KARAF_VERSION}-features.xml" \
  | rg 'pax-web-features'
# then open that pax-web version's features XML and rg 'jetty-server'
```

1. Update ${jetty.version} in root pom.xml to match Pax Web's Jetty bundle version.

2. Modules using ${jetty.version}: graphql/karaf-feature/pom.xml, itests/pom.xml.

Runtime Jetty comes from Pax Web OSGi bundles — no Jetty bundles in Unomi feature.xml.

Verification

```
rg 'cxf\.version|jackson\.version|jetty\.version' pom.xml kar/src/main/feature/feature.xml
mvn -pl kar,package -DskipTests package
./build.sh --ci
```

21.2.6. Upgrading the Elasticsearch client

Unomi does **not** embed the Elasticsearch server. It embeds the **Java API client** and REST transport inside the unomi-persistence-elasticsearch-core OSGi bundle, then exports query DSL types for the conditions bundle.

For **data migration** between Elasticsearch major versions, see also ES7 to ES9 migration (distinct from UNOMI-884 V3 data migration scripts).

Client migration is tracked in UNOMI-901 (supersedes UNOMI-851). JIRA notes an interim ES 8 client on Java 8 for 2.x branches only — Unomi 3 pairs ES 9 with Karaf 4.4 + JDK 17; see JIRA reference.

Version pins

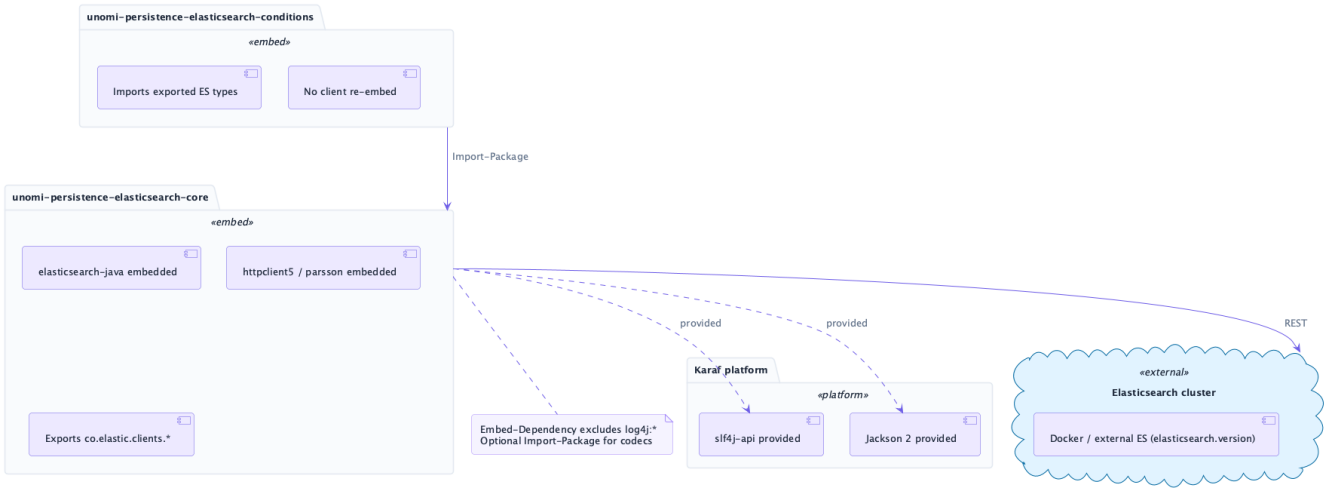
Property	Files
elasticsearch.version	Root pom.xml, bom/pom.xml
elasticsearch.test.version	Root pom.xml, IT Docker image in itests/pom.xml

Update in the same pull request: docker/README.md, docker/src/main/docker/docker-compose-*.yml, itests/docker-compose-snapshot-analysis.yml.

Image tag example: docker.elastic.co/elasticsearch/elasticsearch:9.4.3.

Bundle architecture

Module	Role
persistence-elasticsearch/core	Embeds ES client; exports co.elastic.clients.*; provides PersistenceService
persistence-elasticsearch/conditions	Query builders; imports exported ES types; does not re-embed client



OpenSearch follows the same pattern in persistence-opensearch/core — see OpenSearch upgrade.

Embed rules (core/pom.xml):

```
<Embed-Dependency>
   !groupId=org.apache.logging.log4j;artifactId=*,
   *;scope=compile|runtime
</Embed-Dependency>
<Embed-Transitive>true</Embed-Transitive>
```

Embed: elasticsearch-java, elasticsearch-rest5-client (9.4+), httpclient5, parsson, OpenTelemetry, etc.

Do not embed: log4j JARs (see Log4j); optional codecs (brotli, zstd, netty) — use optional Import-Package only.

Provided: slf4j-api, Jackson 2, commons, Unomi API modules.

Export from core: co.elastic.clients.elasticsearch, _types, _types.query_dsl, util.

Git history (OSGi embedding)

Commit	Issue	Fix
96a018ecb (UNOMI-70)	ES 5 monolithic embed	REST client only; optional codec imports
838fc1cb9 / a605e2731 (UNOMI-225)	ES 7 resolution	Rewrote Import-Package iteratively
2e6a6e8f1 (UNOMI-584)	Test dep pollution	Exclude lucene-core from lucene-test-framework
d3d2d16c4 (UNOMI-901)	HLRC Java API	Export co.elastic.clients.*; remove ChildFirstClassLoader
PR2 #795	log4j 2.24 embed	OSGi failure — exclude log4j from embed in PR3
PR3 #796	ES 9.4.3 / httpclient5 5.6	Routing API List<String>; optional brotli/zstd/netty imports

httpclient5 5.6+ optional codec imports

ES 9.4+ uses httpclient5 5.6.x. Add to Import-Package in core (before *):

```
com.aayushatharva.brotli4j.decoder;resolution:=optional,
com.aayushatharva.brotli4j.encoder;resolution:=optional,
com.github.luben.zstd;resolution:=optional,
io.netty.buffer;resolution:=optional,
io.netty.channel;resolution:=optional,
io.netty.util;resolution:=optional,
javax.xml.datatype;resolution:=optional,
javax.xml.namespace;resolution:=optional,
javax.xml.transform.dom;resolution:=optional,
org.newsclub.net.unix;resolution:=optional,
```

Do **not** add compile Maven dependencies for brotli4j / zstd-jni.

Java API checklist

1. mvn -pl persistence-elasticsearch/core,persistence-elasticsearch/conditions -am -DskipTests compile
2. Fix API breaks in ElasticSearchPersistenceServiceImpl.java (e.g. UpdateRequest.routing() → List<String> in 9.4).
3. ES 9.4 embeds Jackson 3 (tools.jackson.core) inside the bundle; Unomi app code uses Jackson 2 — do not add Jackson 3 as a root dependency.
4. Audit manifest — expect no embedded log4j or native codec JARs.

Verification

```
mvn -pl persistence-elasticsearch/core install -DskipTests
mvn -pl extensions/groovy-actions/karaf-kar -DskipTests package 2>&1 | rg -i 'Unable to
resolve|persistence-elasticsearch'
./build.sh --ci
```

21.2.7. Upgrading the OpenSearch client

OpenSearch persistence mirrors Elasticsearch: a fat OSGi bundle embeds the Java client and HTTP stack (UNOMI-828, 078ebceda).

For switching backends at runtime, see Elasticsearch to OpenSearch migration.

Version pins

Property	Files
opensearch.version	Root pom.xml, bom/pom.xml, Docker compose, IT image
opensearch.rest.client.version	Root pom.xml — keep in sync with opensearch.version



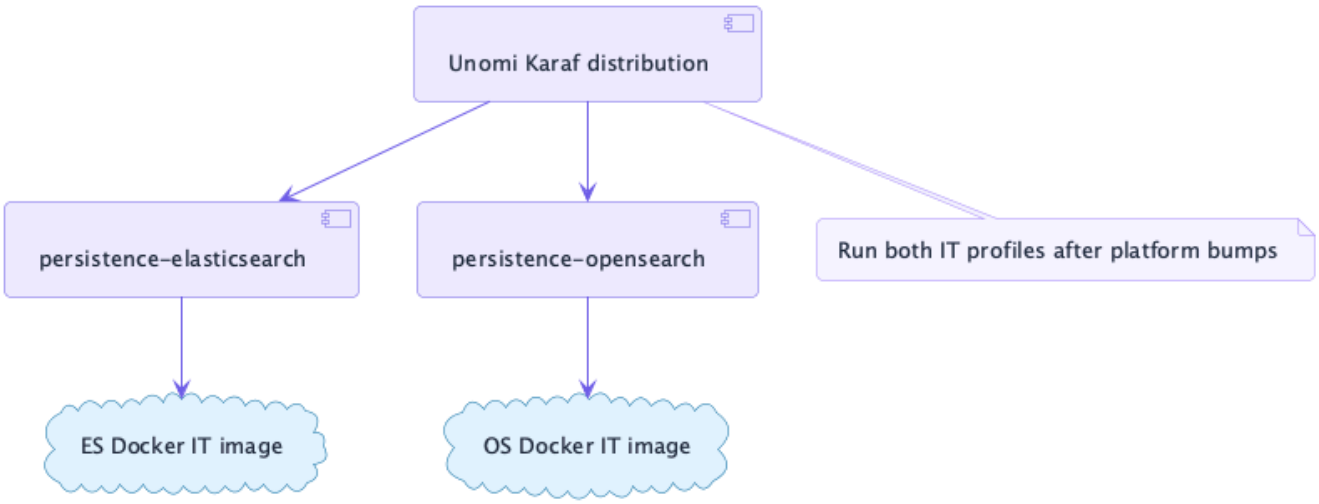
${opensearch.version} must match a published Docker Hub tag. Verify before bumping:

```
docker pull opensearchproject/opensearch:${opensearch.version}
```

Latest OpenSearch 3.x line on Docker Hub (as of Unomi 4.0 work): **3.7.0**. Do not assume every Maven Central opensearch-java version has a matching Docker image.

Dual-engine verification (diagram)

Unomi supports Elasticsearch and OpenSearch persistence modules; platform bumps must keep both IT profiles green.



Bundle architecture

Module: persistence-opensearch/core (+ conditions).

Same embed rules as Elasticsearch core — including log4j embed exclusion and optional codec imports after httpclient5 5.6.

Export-Package: org.opensearch., org.opensearch.index.query., org.apache.unomi.persistence.opensearch.

Step-by-step

1. Bump opensearch.version and opensearch.rest.client.version in root pom.xml.
2. Update Docker compose files and docker/README.md.
3. mvn -pl persistence-opensearch/core,persistence-opensearch/conditions -am -DskipTests compile
4. Fix Java API breaks in OpenSearchPersistenceServiceImpl.java if any.
5. Sync optional Import-Package with ES core when httpclient version jumps.

6. Confirm log4j embed exclusion on OS core.
7. Feature verify + ./build.sh --ci; run OpenSearch IT profile.

ES vs OS coupling

Concern	Keep in sync?
httpclient5 optional imports	**Yes**
log4j embed exclusion	**Yes**
Docker image tags	No — independent server versions
Java API code changes	No — separate impl classes

You may upgrade ES and OS in one pull request but use separate commits for review clarity.

21.2.8. Upgrading Log4j

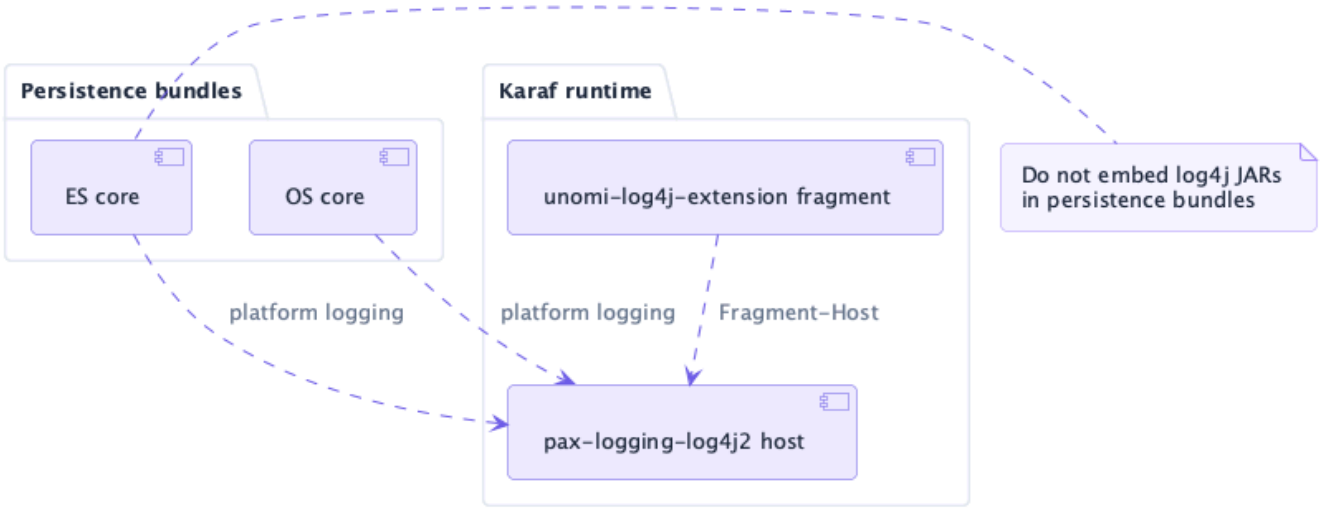
Unomi uses **pax-logging-log4j2** as the platform logging implementation. Custom appenders live in extensions/log4j-extension as an OSGi **fragment** on pax-logging. Persistence bundles must **never embed** log4j JARs and must not call Log4j APIs at runtime.

Version pin vs runtime version

Pin	Purpose	Notes
${log4j.version} in pom.xml	Compile-time for log4j-extension (provided)	Managed in bom/pom.xml
pax-logging-log4j2 bundle	**Runtime** Log4j inside Karaf	Version follows ${karaf.version} — inspect bundle manifest, do not assume ${log4j.version} matches

Example: Karaf 4.4.8 ships pax-logging 2.3.0 (Log4j 2.24.3 inside the bundle). Karaf 4.4.11 ships pax-logging 2.3.3 (Log4j 2.25.4 inside). Unomi ${log4j.version} 2.24.0 is still valid for fragment compilation if APIs are compatible — but always verify after a Karaf bump.

Architecture



Persistence ES/OS cores must **not** call Log4j APIs at runtime. REST client verbosity is configured in etc/org.ops4j.pax.logging.cfg (see REST client log levels).

REST client log levels

Elasticsearch and OpenSearch Java clients log through Apache Commons Logging SLF4J pax-logging. Do **not** set levels in persistence start() methods or persistence CM configs.

Deploy-time: etc/custom.system.properties (or env vars):

- org.apache.unomi.elasticsearch.logLevelRestClient / UNOMI_ELASTICSEARCH_LOG_LEVEL_REST_CLIENT (default ERROR)
- org.apache.unomi.opensearch.logLevelRestClient / UNOMI_OPENSEARCH_LOG_LEVEL_REST_CLIENT (default ERROR)
- org.apache.unomi.logs.restClientTracer.level / UNOMI_LOG_LEVEL_REST_CLIENT_TRACER — logger tracer (curl-style trace; default ERROR)

Runtime: Karaf shell log:set org.elasticsearch.client.RestClient DEBUG (or edit org.ops4j.pax.logging.cfg and reload logging).

etc/org.ops4j.pax.logging.cfg maps those system properties to log4j2.logger.* entries for org.elasticsearch.client.RestClient, org.opensearch.client.RestClient, and tracer.

PR3 embed exclusion removed embedded log4j-1.2-api; legacy code that called org.apache.log4j.Logger in ElasticSearchPersistenceServiceImpl.start() then failed with ClassNotFoundException — fixed by moving control to pax-logging (not by re-embedding log4j).

Optional Maven compile deps on log4j in the ES client POM are excluded from embed and unused at runtime. Felix must not see duplicate Bundle-Activator entries from embedded log4j-core.

Git history

- PR2 #795 (ef10dfe15): bumped to 2.24.0 — **reverted** after ES OSGi bundle failed feature verify (embedded log4j conflict).

- PR3 #796 (a5e6f35d2): re-applied 2.24.0 **with** persistence embed exclusion.
- UNOMI-936 (e9a330052): log format aligned with Karaf/pax-logging patterns — config in org.ops4j.pax.logging.cfg.

Embed exclusion (both persistence cores):

```
<Embed-Dependency>
  !groupId=org.apache.logging.log4j;artifactId=*,
  *;scope=compile|runtime
</Embed-Dependency>
```

Procedure

1. Never bump log4j alone without persistence OSGi verification.
2. After a Karaf bump, inspect pax-logging version change — runtime Log4j may jump even if ${log4j.version} is unchanged.
3. Update ${log4j.version} in root pom.xml.
4. Confirm ES and OS core embed exclusion (see Elasticsearch and OpenSearch).
5. Inspect manifests — no log4j-*.jar in Embedded-Artifacts.
6. mvn -pl package -DskipTests package then ./build.sh --ci.

log4j-extension fragment

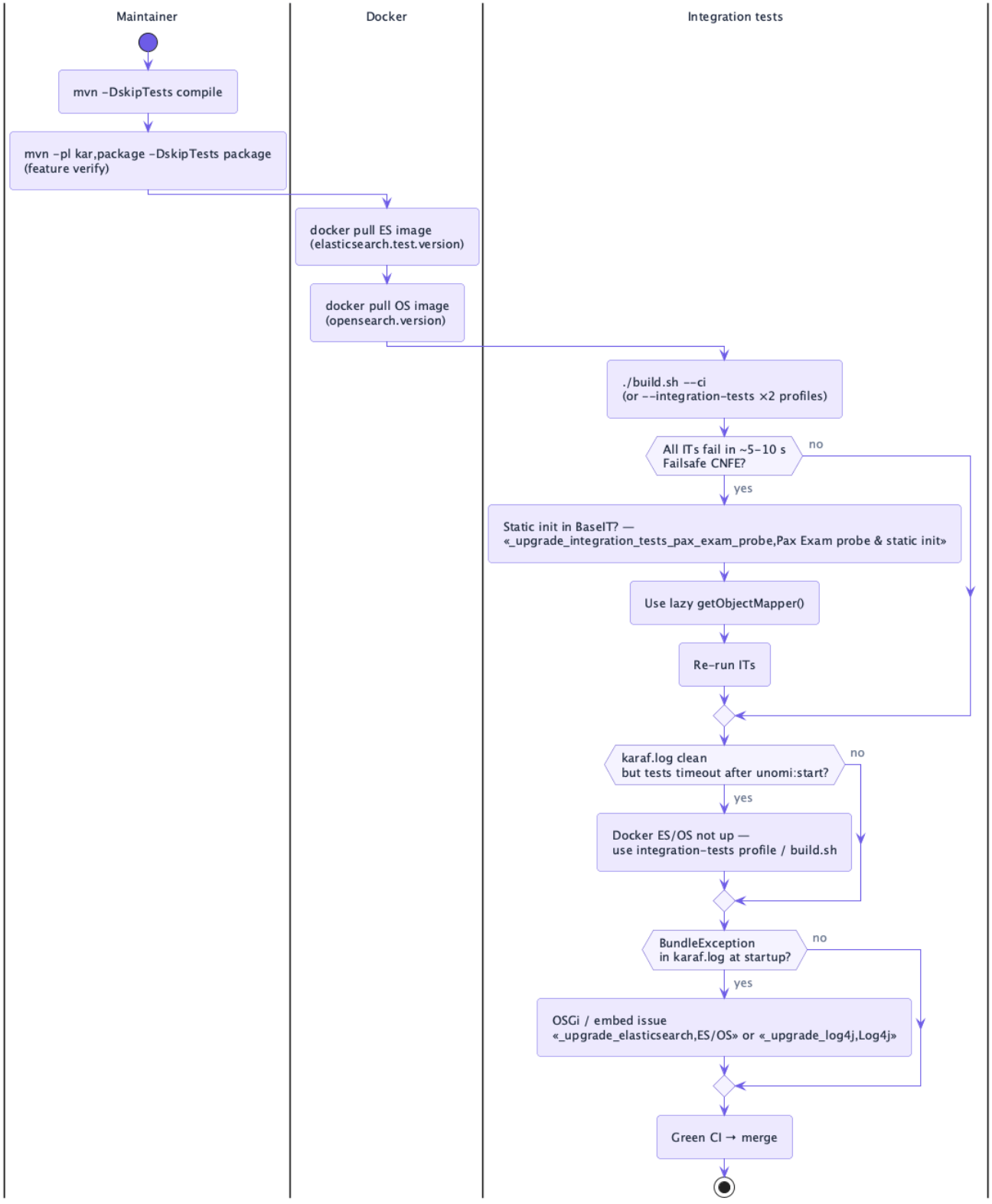
extensions/log4j-extension/pom.xml: Fragment-Host = org.ops4j.pax.logging.pax-logging-log4j2; log4j-core scope = provided.

21.2.9. Integration tests and Docker

Platform upgrades are validated by **Pax Exam** integration tests plus **Docker-backed** search engines. Failures often appear in Maven Failsafe output, not in karaf.log.

See also: itests/README.md and build.sh.

Validation pipeline after a platform bump



Architecture

Layer	Technology	Location
Test runner	Pax Exam + JUnit 4	itests/.../BaseIT.java
Karaf under test	Unomi distribution tar.gz	itests/pom.xml — ${karaf.version}
Search engine	docker-maven-plugin	ES port 9400 / OS port 9401

Layer	Technology	Location
JSON in IT helpers	BaseIT.getObjectMapper() → CustomObjectMapper.getCustomInstance() (lazy, on first use)	itests/.../BaseIT.java
Exam logs	Karaf instance	itests/target/exam/*/data/log/karaf.log
Reports	Failsafe	itests/target/failsafe-reports/

Docker IT (UNOMI-921)

UNOMI-921 motivation: stop downloading ES binaries in Maven, remove the default_template workaround required with elasticsearch-maven-plugin on ES 8/9, and match the OpenSearch Docker pattern.

Shipped: elasticsearch-maven-plugin replaced with io.fabric8:docker-maven-plugin (5a5aedba7, #767). Related stabilization: UNOMI-937, tooling UNOMI-944.

Engine	Image	Property
Elasticsearch	docker.elastic.co/elasticsearch/elasticsearch:${elasticsearch.test.version}	elasticsearch.test.version
OpenSearch	opensearchproject/opensearch:${opensearch.version}	opensearch.version

Always run docker pull for both images before bumping version properties.

Pax Exam probe and static initialization (Karaf 4.4.11 / Pax Exam 4.14)

After a Karaf bump, integration tests may fail **before Unomi starts**. karaf.log can look healthy or incomplete while Failsafe reports hundreds of errors in a few seconds — that pattern almost always means the **PAXEXAM-PROBE** bundle failed to load test classes, not a Karaf startup crash.

How the probe works

- Pax Exam 4.14 builds the probe with rawBuilder(): compiled test classes under target/test-classes plus DynamicImport-Package: * — not the full Maven test classpath.
- Maven <dependencies> on itests/pom.xml are for **compile-time** only. They are **not** automatically embedded in the probe.
- Types the probe needs at runtime must be exported by **already resolved** OSGi bundles in the Karaf container, unless you explicitly embed them on Bundle-ClassPath (avoid — see Do not embed).

Why static initializers break (UNOMI-875 PR3 lesson)

A static { ... } block (or static final field initialized with new ...()) runs when the JVM **loads** the probe class — typically when Pax Exam installs PAXEXAM-PROBE, **before** @Before, **before** unomi:start, and often **before** Unomi features finish installing.

That is too early for types that live in Unomi features, not in Karaf boot:

- com.fasterxml.jackson.module.jaxb.JaxbAnnotationModule — **not** in Karaf's jackson feature; shipped in Unomi's unomi-base (kar/src/main/feature/feature.xml).
- CustomObjectMapper and its Jackson modules — same: available only after unomi-base bundles are resolved.

On Karaf 4.4.8 this often **appeared** to work because feature boot timing usually left unomi-base ready before the probe loaded BaseIT. On 4.4.11 (heavier/async feature boot, dual Jackson stacks) the same static block fails reliably with:

```
ClassNotFoundException: com.fasterxml.jackson.module.jaxb.JaxbAnnotationModule
   not found by PAXEXAM-PROBE-...
```

Historical mistake: BaseIT had a static block since UNOMI-568 (May 2022) for a shared ObjectMapper; UNOMI-569 (June 2022) added JaxbAnnotationModule to mirror production JSON handling. That duplicated CustomObjectMapper and forced Jackson types to load at probe install time.

Do not do this

```
// BAD — runs when probe class loads, before Unomi features are up
protected static final ObjectMapper objectMapper;
static {
   objectMapper = new ObjectMapper();
   objectMapper.registerModule(new JaxbAnnotationModule());
}
```

Shipped fix (UNOMI-875 PR3, current)

Use **on-first-use** access after Unomi is running. Delegate to the same mapper as the server:

```
/** JSON mapper for IT HTTP/JSON helpers — lazy, after Unomi features are up. */
protected ObjectMapper getObjectMapper() {
   return CustomObjectMapper.getCustomInstance();
}
```

Call sites use getObjectMapper() in test methods (after @Before / waitForStartup()), not a static field. CustomObjectMapper.getCustomInstance() returns the holder singleton — the first call happens when HTTP/JSON helpers run, when unomi-base (including jackson-module-jaxb-annotations) is already active, so DynamicImport-Package: * can wire the types.

Do not embed (removed anti-patterns)

An earlier PR3 attempt embedded test JARs on the probe Bundle-ClassPath (copy-probe-lib, probe-bundle-classpath.properties, getOsgiService() JDK proxy). That was **more complex than necessary** and caused new failures (ClassCastException, Maven phase ordering bugs, Require-Bundle wiring races). **Do**

not reintroduce:

- maven-dependency-plugin copying Unomi JARs into target/test-classes/lib/
- @ProbeBuilder setting explicit Bundle-ClassPath for embedded libs
- Require-Bundle on the probe for Unomi bundles
- JDK dynamic proxy on getOsgiService() to paper over duplicate embedded API classes

If ITs fail with ClassNotFoundException on Jackson or Unomi types, fix **when** the probe touches those types (move off static init), not by embedding the whole test classpath into the probe.

Rules for probe-safe IT code

- **No static initializers** in BaseIT or IT base classes that reference Jackson modules, CustomObjectMapper, or other types from Unomi/Karaf features not in vanilla Karaf boot.
- **Lazy/on-first-use** for JSON mappers, parsers, and helpers that need Unomi or Jackson JAXB types — call from test methods after @Before has started Unomi (or after the relevant OSGi service is known to be up).
- **Do not** duplicate production mapper setup in the probe; use CustomObjectMapper.getCustomInstance() via BaseIT.getObjectMapper() (or OSGi services from the running container).
- **Do not** run mvn failsafe:integration-test on itests without the integration-tests profile unless Docker ES/OS is already listening on the IT port (9400 / 9401). A fixed probe can still hang in waitForStartup() if persistence is unreachable.

Symptom diagnosis

Failsafe symptom	Typical cause	Fix
ClassNotFoundException: …JaxbAnnotationModule (all ITs, ~5–10 s)	Static init or static field touching Jackson JAXB before unomi-base is up	Remove static block; use lazy instance getObjectMapper() CustomObjectMapper.getCustomInstance()
ClassNotFoundException: …PersistenceService (or other Unomi type) at class load	Static init or early reference before Unomi features resolve	Move work out of static blocks; ensure @Before starts Unomi first
All tests error in ~5–10 s; karaf.log short or still installing features	Probe class loading failed early (static init or missing dynamic import)	Failsafe + itests/target/it-run-operator-note.txt; check for static blocks
Tests reach unomi:start then timeout (~10–20 min)	Docker ES/OS not running or persistence blueprint failed	karaf.log (elasticSearchPersistenceService, port 9400); docker ps
EOFException / RMI Connection refused after long wait	Karaf container died during startup (often ES connectivity)	Tail of karaf.log in the exam folder

Reading logs

Artifact	Healthy signal	Failure signal
karaf.log	Features install, persistence blueprints start, Pax Web binds	BundleException, Unable to resolve, elasticSearchPersistenceService errors
Failsafe reports	Tests pass	ClassNotFound in probe (static init), timeouts
itests/target/it-run-operator-note.txt	Normal run duration, memory samples	~7 s run with 292 errors probe failure
docker-maven-plugin	Container starts, HTTP 200 on :9400 / :9401	Unable to pull, manifest not found, connection refused from Unomi



A clean or truncated karaf.log with all ITs failing in seconds usually indicates probe class loading (often a static block in BaseIT). A clean log with tests stuck after unomi:start usually indicates Docker / persistence — not the probe.

Running ITs locally

```
./build.sh --integration-tests
./build.sh --integration-tests --use-opensearch
./build.sh --ci --integration-tests --no-karaf
```

Use Maven profile ci-build-itests when building the itests module directly. The integration-tests profile starts the Docker search container; ad-hoc mvn -pl itests verify without that profile (or without ./build.sh --integration-tests) skips Docker and ITs will hang waiting for Elasticsearch/OpenSearch.

Karaf bump side effects (UNOMI-875 PR3)

Symptom	Cause	Fix
All ITs fail in ~5–10 s; ClassNotFoundException: …JaxbAnnotationModule	Static ObjectMapper init in BaseIT before unomi-base is up	Lazy instance getObjectMapper() CustomObjectMapper.getCustomInstance() — Pax Exam probe & static init
IT compile uses Jetty websocket client	${jetty.version} drift vs Pax Web runtime	Align Jetty pin after Karaf bump — CXF/Jackson/Jetty
Clean karaf.log, tests timeout after unomi:start	Docker ES/OS not started or wrong image tag	./build.sh --integration-tests; fix elasticsearch.test.version / opensearch.version

Build script (UNOMI-887)

Prefer ./build.sh --ci over ad-hoc Maven invocations — it runs the same gates as GitHub Actions (compile, feature verify path, IT profiles). See build.sh in the manual.

IT developer tooling (UNOMI-944)

For cross-run comparison and live Karaf inspection during upgrade debugging, see itests/README.md (run archival, log diff workflows).

21.2.10. Dependency hygiene (non-platform bumps)

Not every version bump is a Karaf or persistence migration. PR2-style hygiene (UNOMI-875 #795, ef10dfe15) covers **application** dependencies that must **not** break OSGi rules established in platform upgrades.

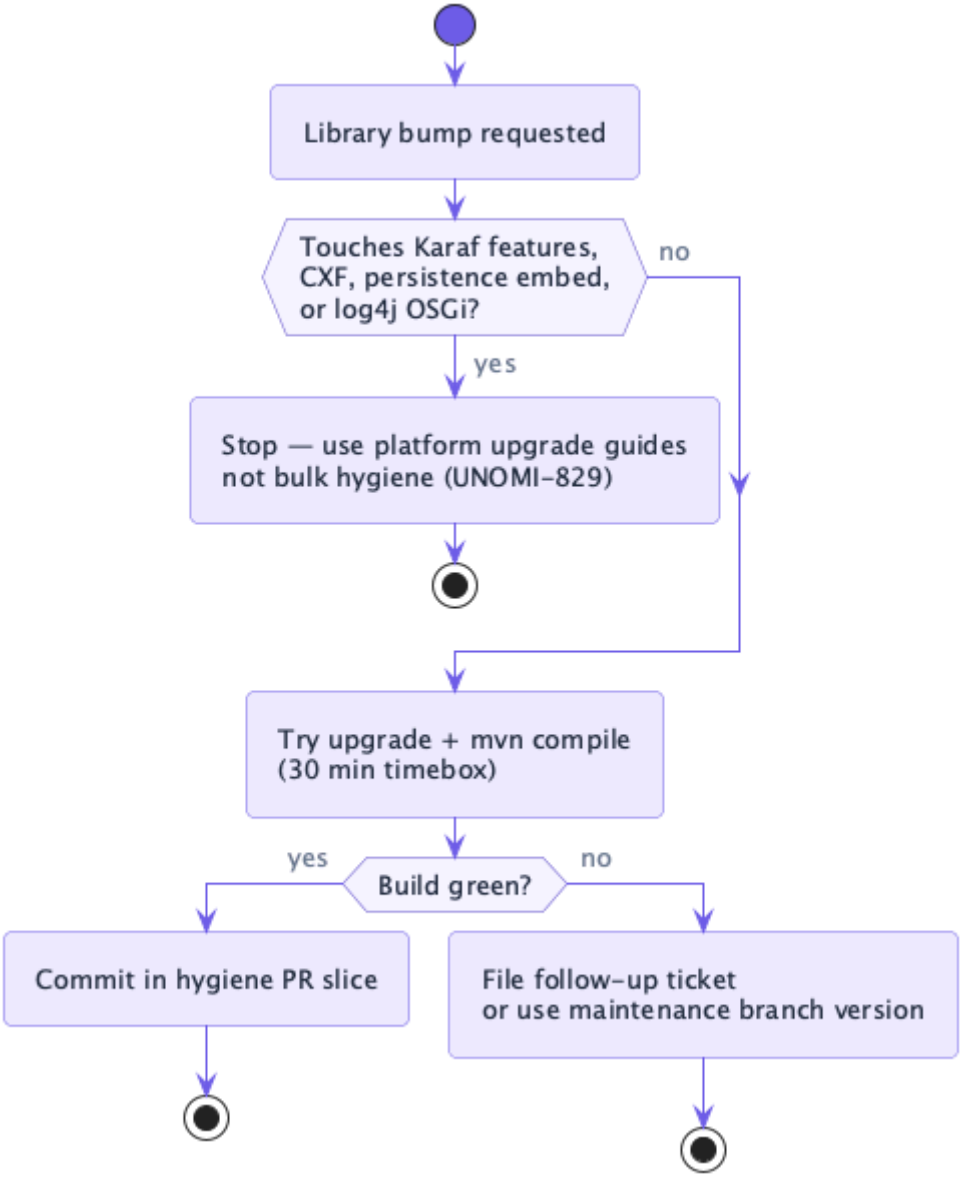
The bulk dependency pass UNOMI-829 defines the maintenance strategy: 30-minute timebox per library, maintenance-branch fallback, and separate tickets for Karaf/CXF/GraphQL — see JIRA reference.

Scope

Track	Examples	OSGi impact
Frontend / WAB npm	wab/, graphql-ui/ yarn lockfiles	Build-time only — verify WAB packaging still works
Messaging	kafka-clients in plugins/kafka-injector	Usually embed or declare imports explicitly in bundle
Commons / utilities	commons-lang3, commons-io, Guava	Prefer provided if Karaf feature supplies them
Logging	${log4j.version}	**High** — never without persistence embed exclusion

Recommended order (platform stack)

Hygiene vs platform upgrade scope



1. Karaf (+ JDK if major) — Karaf
2. CXF / Jackson / Jetty alignment — CXF/Jackson/Jetty
3. Elasticsearch + OpenSearch clients — Elasticsearch, OpenSearch
4. Log4j — Log4j (last among platform pins)
5. Application hygiene (npm, kafka, commons) — this section
6. ./build.sh --ci

npm / frontend (Track E1)

1. Bump lockfiles in wab/ and graphql-ui/ with yarn upgrade or targeted version edits.
2. Rebuild WAB modules: mvn -pl wab,graphql-ui -am -DskipTests package
3. Smoke-test tracker and GraphQL UI in a running Karaf if UI behavior changed.

Java libraries (Track E2)

1. Bump property in root pom.xml; confirm bom/pom.xml manages the artifact.
2. Check consuming bundle Import-Package / embed instructions — do not pull platform libs into compile scope accidentally.
3. Run unit tests for affected modules before full CI.

Log4j hygiene trap (PR2 lesson)

Bumping ${log4j.version} alone caused Elasticsearch core to embed log4j-core transitively Felix duplicate activator feature verify failure. Always pair log4j bumps with:

```
!groupId=org.apache.logging.log4j;artifactId=*,
```

in persistence ES/OS Embed-Dependency (see Log4j upgrade).

Verification

```
mvn -DskipTests compile
mvn -pl package -DskipTests package
./build.sh --ci
```

21.3. SSH Shell Commands

Apache Unomi provides its own Apache Karaf Shell commands to make it easy to control the application lifecycle or perform queries or modifications on the internal state of the system.

All Apache Unomi-specific commands are namespaced and use the unomi: namespace. You can use the Apache Karaf Shell's autocompletion to list all the commands available.

21.3.1. Using the shell

You can connect to the Apache Karaf SSH Shell using the following command:

```
ssh -p 8102 karaf@localhost
```

Authenticate with the karaf user and the password from UNOMI_ROOT_PASSWORD (Unomi does not ship a known default password). Set the password before start; see Getting started and client-facing hardening.

Once connected you can simply type in :

```
unomi:
```

And hit the <tab> key to see the list of all the available Apache Unomi commands. Note that some commands are only available when the application is started.

You can also use the help command on any command such as in the following example:

```
karaf@root()> help unomi:migrate
DESCRIPTION
        unomi:migrate

    This will migrate your data in Elasticsearch or OpenSearch to be compliant with the current
version.
    It's possible to configure the migration using OSGI configuration file:
org.apache.unomi.migration.cfg,
    if no configuration is provided then questions will be prompted during the migration process.

SYNTAX
        unomi:migrate [fromVersionWithoutSuffix] [skipConfirmation]

ARGUMENTS
        fromVersionWithoutSuffix
                Origin version without suffix/qualifier (e.g: 1.2.0)
                (defaults to 1.2.0)
        skipConfirmation
                Should the confirmation before starting the migration process be skipped ?
                (defaults to false)
```

21.3.2. Lifecycle commands

The commands control the lifecycle of the Apache Unomi server and are used to migrate, start or stop the server.

Table 7. Table Lifecycle commands

Command	Arguments	Description
setup	distribution-feature-name,--force	This command must be used only when the Apache Unomi application is NOT STARTED. It will perform initial setup of the application using the specified distribution feature name (unomi-distribution-elasticsearch for example).

Command	Arguments	Description
migrate	fromVersion	This command must be used only when the Apache Unomi application is NOT STARTED. It will perform migration of the data stored in search engine using the argument fromVersion as a starting point. Contributors writing new scripts should read Writing migration scripts.
stop	n/a	Shutsdown the Apache Unomi application
start	n/a	Starts the Apache Unomi application with the specified distribution (or by default unomi-distribution-elasticsearch). Note that this state will be remembered between Apache Karaf launches, so in general it is only needed after a first installation
version	n/a	Prints out the currently deployed version of the Apache Unomi application inside the Apache Karaf runtime.

21.3.3. Runtime commands

These commands are available once the application is running. If an argument is between brackets [] it means it is optional.

Most list/view/remove operations on profiles, events, rules, segments, sessions, actions and conditions now go through unomi:crud. Set the tenant first with tenant-set (see Tenant commands). Use unomi:crud help TYPE for the fields each type accepts.

Table 8. Table Runtime commands

Command	Arguments	Description
crud list	type [[--csv] [-n maxEntries]]	Lists items of the given type for the current tenant. Common types: profile, event, rule, segment, session, actiontype, conditiontype. Example: crud list profile or crud list rule --csv.

Command	Arguments	Description
crud read	type item-id	Dumps one item as JSON. Example: crud read profile PROFILE_ID or crud read rule RULE_ID.
crud delete	type item-id	Deletes one item. Warning: no confirmation is asked. Example: crud delete profile PROFILE_ID.
crud create / crud update	type JSON-or-file-URI [for update: item-id then JSON]	Create or update an item from a JSON string or a file:// path. Use crud help TYPE for required fields.
rule-reset-stats	n/a	Resets the rule statistics. This is notably useful when trying to understand rule performance and impact
rule-tail	n/a	Dumps any rule that is executed by the server. Only executed rules are logged here. If you want to have more detailed information about a particular rule's condition evaluation and if it's already been raised use the rule-watch command instead. This tail will continue until a CTRL+C key combination is pressed.

Command	Arguments	Description
rule-watch	rule-ids	Dumps detailed evaluation and execution information about the rules that are where specified in the rule-ids arguments (you can specify multiple rule identifiers separated by spaces). The Status column has the following values: EVALUATE - indicates that the rule's conditions are being evaluated (but they might not be satisfied), AR PROFILE - means the rule has already been raised for the profile and will therefore not execute again for this profile, AR SESSION - means the rule has already been executed for this session and will therefore only executed when another session for the profile is created, EXECUTE means the rule's actions are being executed.
event-tail	n/a	Dumps any incoming events to the Apache Unomi server to the console. Use CTRL+C to exit tail

Command	Arguments	Description
deploy-definition	[bundleId] [type] [fileName]	This command can be used to force redeployment of definitions from bundles. By default existing definitions will not be overriden unless they come from SNAPSHOT bundles. Using this command you can override this mechanism. Here are some examples of using this command: unomi:deploy-definition 175 rule * will redeploy all the rules provided by bundle with id 175. If you launch the command without any arguments you will get prompts for what you want to deploy from which bundle. If you want to deploy all the definitions of a bundle you can also use wildcards such as in the following example: deploy-definition 175 * *. It is also possible to give no argument to this command and it will then interactively request the definitions you want to deploy.
undeploy-definition	[bundleId] [type] [fileName]	This command does the opposite of the deploy-definition command and works exactly the same way in terms of arguments and interactive mode except that it undeploys definitions instead of deploying them. This command can be very useful when working on a plugin. For example to remove all the definitions deployed by a plugin you can simply use the following command: undeploy-definition BUNDLE_ID * * when BUNDLE_ID is the identifier of the bundle that contains your plugin.

21.3.4. Tenant commands

Multi-tenant shell operations (Unomi 4.0+). See Multi-tenancy for REST equivalents.

Table 9. Table Tenant commands

Command	Arguments	Description
tenant-get	n/a	Print the tenant ID stored for this Karaf session.
tenant-set	tenant-id	Set the current tenant for this shell session (system is allowed). Other CRUD commands use this context.
crud list tenant	[--csv]	List tenants (excludes the built-in system tenant from normal listings).
crud read tenant	tenant-id	Show one tenant as JSON.
crud create tenant	JSON or file:// path	Create a tenant. Use crud help tenant for the accepted properties.
crud update tenant	tenant-id JSON or file:// path	Update tenant metadata. The shell names the restricted event types property restrictedEventPermissions.
crud delete tenant	tenant-id	Delete a tenant.
crud list apikey	[--csv]	List API keys across all tenants (keys are masked in the table).
crud create apikey	JSON	Create an API key for a tenant (tenantId, keyType, optional validityPeriod).
cache	--tenant ID [--clear]	Inspect or clear per-tenant caches (segments, rules, and related types).

21.3.5. Scheduler commands

These commands manage background tasks created by the cluster-aware scheduler (see Task scheduler).

Table 10. Table Scheduler commands

Command	Arguments	Description
task-list	[-s status] [-t type] [--limit N]	List scheduled tasks. Filter by status (SCHEDULED, RUNNING, FAILED, ...) or task type.
task-show	task-id	Show full details for one task (parameters, checkpoint data, errors).

Command	Arguments	Description
task-cancel	task-id	Cancel a task (stops future executions).
task-retry	task-id [-r]	Retry a failed task. -r / --reset clears the failure count.
task-purge	[-d days] [-f]	Delete old COMPLETED tasks (default: older than 7 days). -f skips confirmation.
task-executor	boolean	Show whether this node runs tasks and its scheduler nodeId.

21.4. Writing plugins

Apache Unomi is extensible through OSGi plugins (Karaf bundles). Many use cases do **not** need a plugin:

- Groovy actions — custom rule actions stored at runtime (see Configuration)
- Complex condition trees — combine built-in condition types
- Rules, segments, and REST configuration — manage through APIs or JSON definitions in a bundle
- JSON Schemas — validate events without Java code

Build a plugin when you need compiled Java logic, for example:

- Custom ConditionEvaluator or search-engine *QueryBuilder implementations
- Custom ActionExecutor integrations with external systems
- Index mappings, Painless scripts, or migration patches shipped with your feature
- Background work through TaskExecutor and SchedulerService (see Task scheduler)

21.4.1. Prerequisites

- **Java 17** or later
- **Apache Karaf 4.4** (bundled with Unomi distributions)
- Maven packaging set to bundle (Felix maven-bundle-plugin)
- Depend on unomi-api and import unomi-bom for aligned artifact versions

See also:

- Data model overview
- Built-in condition types — registration patterns for Unomi 3.0+
- Built-in action types

- Condition validation — validation metadata on condition parameters

21.4.2. Types vs instances

Extension points follow a type/instance model (like classes and objects):

- **ConditionType** → **Condition** instances in segments, rules, and queries
- **ActionType** → **Action** instances executed by rules
- **PropertyType** → profile or session property metadata (UI hints; not server-side validation)

21.4.3. META-INF/cxs/ layout

A Unomi plugin is an OSGi bundle. JSON definitions and bundle resources live under src/main/resources/META-INF/cxs/ (and related paths below).

Entity	Directory under META-INF/cxs/	Auto-loaded at bundle start?
ActionType	actions/	Yes
ConditionType	conditions/	Yes
ValueType	values/	Yes
PropertyMergeStrategyType	mergers/	Yes
PropertyType	properties/profiles/… or properties/sessions/…	Yes
Rule	rules/	Yes
Segment	segments/	Yes
Scoring	scoring/	Yes
Persona	personas/	Yes
Patch	patches/	Yes (first run)
Index mapping	mappings/	Loaded by persistence layer
JSON Schema	schemas/	Loaded by schema service
Expression filter	expressions/	Loaded by scripting layer
Painless script	painless/	Loaded by persistence layer
Migration script	migration/	Used by shell migration tooling

Goals and campaigns can be deployed with unomi:deploy-definition but are not auto-loaded from bundles the same way as rules and segments.

Tags are not separate JSON files — use systemTags on metadata inside other definitions.

21.4.4. OSGi service registration (Declarative Services)

New plugins must use **OSGi Declarative Services** (@Component, @Reference) — not Blueprint XML. Unomi is migrating away from Blueprint; legacy bundles such as plugins/baseplugin and parts of persistence-* still ship Blueprint wiring, but that is not a template for new code.

Enable DS component generation in your bundle POM:

```
<dependency>
  <groupId>org.osgi</groupId>
  <artifactId>org.osgi.service.component.annotations</artifactId>
  <scope>provided</scope>
</dependency>
```

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
    <instructions>
      <_dsannotations>*</_dsannotations>
    </instructions>
  </configuration>
</plugin>
```

Follow plugins/advanced-conditions/pom.xml for a minimal DS-enabled plugin module.

Service property keys:

Extension	Service interface	OSGi property key
Condition evaluator	org.apache.unomi.persistence.spi.conditions.evaluator.ConditionEvaluator	conditionEvaluatorId
Elasticsearch query builder	org.apache.unomi.persistence.elasticsearch.ConditionESQueryBuilder	queryBuilderId
OpenSearch query builder	org.apache.unomi.persistence.opensearch.ConditionOSQueryBuilder	queryBuilderId
Action executor	org.apache.unomi.api.actions.ActionExecutor	actionExecutorId
Value type validator	org.apache.unomi.api.services.ValueTypeValidator	(service type only)

The JSON descriptor field (conditionEvaluator, queryBuilder, actionExecutor) must match the OSGi property value.

21.4.5. Creating a Maven plugin project

A minimal plugin is a Maven bundle project. Prefer importing unomi-bom rather than pinning versions manually.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-unomi-plugin</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>bundle</packaging>

  <properties>
    <unomi.version>4.0.0-SNAPSHOT</unomi.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.apache.unomi</groupId>
        <artifactId>unomi-bom</artifactId>
        <version>${unomi.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.apache.unomi</groupId>
      <artifactId>unomi-api</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.osgi</groupId>
      <artifactId>org.osgi.service.component.annotations</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>5.1.9</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Embed-Dependency>*;scope=compile|runtime</Embed-Dependency>
            <_dsannotations>*</_dsannotations>
            <Import-Package>*</Import-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

21.4.6. Deployment and updates

When a bundle starts, predefined JSON definitions are deployed **once** if they do not already exist. Redeploying the same bundle does not overwrite existing definitions.

To force redeploy:

- Karaf shell: unomi:deploy-definition <bundleId> <type> <fileName> — see the shell commands chapter in this manual
- Structural changes to existing definitions: Migration patches

Wildcards are supported, for example unomi:deploy-definition 175 rules *.

21.4.7. JSON-only extensions

Predefined segments

Place JSON under META-INF/cxs/segments/:

```
{
 "metadata": {
  "id": "leads",
  "name": "Leads",
  "scope": "systemscope",
  "description": "Profiles with leadAssignedTo set",
  "readOnly": true
 },
 "condition": {
  "type": "booleanCondition",
  "parameterValues": {
   "operator": "and",
   "subConditions": [
    {
     "type": "profilePropertyCondition",
     "parameterValues": {
      "propertyName": "properties.leadAssignedTo",
      "comparisonOperator": "exists"
     }
    }
   ]
  }
 }
}
```

Predefined rules

Place JSON under META-INF/cxs/rules/:

```
{
 "metadata": {
  "id": "evaluateProfileSegments",
  "name": "Evaluate segments",
  "description": "Evaluate segments when a profile is modified",
  "readOnly": true
 },
 "condition": {
  "type": "profileUpdatedEventCondition",
  "parameterValues": {}
 },
 "actions": [
  {
   "type": "evaluateProfileSegmentsAction",
   "parameterValues": {}
  }
 ]
}
```

Predefined property types

- Profiles: META-INF/cxs/properties/profiles/
- Sessions: META-INF/cxs/properties/sessions/

See Property types for the full model.

Predefined child conditions

Specialize an existing condition type with parentCondition under META-INF/cxs/conditions/:

```
{
 "metadata": {
  "id": "profileUpdatedEventCondition",
  "name": "profileUpdatedEventCondition",
  "systemTags": ["event", "eventCondition"],
  "readOnly": true
 },
 "parentCondition": {
  "type": "eventTypeCondition",
  "parameterValues": { "eventTypeId": "profileUpdated" }
 },
 "parameters": []
}
```

Inside parentCondition, a value written parameter::name takes the value the condition instance gives its name parameter. A parameter the instance leaves unset is optional, and the condition that reads it is left out:

- Under an AND, it states no constraint. formEventCondition with no pagePath matches a form on any page.

- Under an OR, it is not a match. hoverEventCondition compares targetId OR targetPath, and with only targetId set it matches hovers on that target only.
- An OR whose branches are all unset matches nothing, and an AND whose branches are all unset matches everything.
- A condition instance whose own parameter is unset is left out as a whole, constraints of its type included.

These rules apply in both rule evaluation and segment queries.

Predefined personas

Place JSON under META-INF/cxs/personas/. Each file can define a persona profile and optional sessions for testing personalization.

21.4.8. Custom action types

1. Add META-INF/cxs/actions/myAction.json with an actionExecutor ID.
2. Implement ActionExecutor and register it with @Component and matching actionExecutorId.

JSON descriptor (same shape as extensions/lists-extension/actions/.../addToLists.json):

```
{
 "metadata": {
  "id": "myNotifyAction",
  "name": "myNotifyAction",
  "readOnly": false
 },
 "actionExecutor": "myNotifyAction",
 "parameters": [
  { "id": "message", "type": "string", "multivalued": false }
 ]
}
```

Java executor (Declarative Services — write this pattern in your plugin; do not copy Blueprint from older bundles):

```
@Component(service = ActionExecutor.class, property = {"actionExecutorId=myNotifyAction"})
public class MyNotifyAction implements ActionExecutor {

    private ProfileService profileService;

    @Reference
    public void setProfileService(ProfileService profileService) {
        this.profileService = profileService;
    }

    @Override
    public int execute(Action action, Event event) {
        String message = (String) action.getParameter("message");
        // use profileService / event as needed
        return EventService.NO_CHANGE;
    }
}
```

For a maintained DS registration pattern in the tree, see graphql/cxs-impl/.../CDPUpdateListsAction.java. For action business logic (lists), see extensions/lists-extension/actions/.../AddToListsAction.java — migrate its Blueprint wiring to DS if you use it as a starting point.

Catalog of built-in actions: Built-in action types.

21.4.9. Custom condition types

Custom condition types combine:

- A JSON descriptor in META-INF/cxs/conditions/
- A ConditionEvaluator OSGi service (required for rule and event evaluation)
- ConditionESQueryBuilder / ConditionOSQueryBuilder services (required when the condition is used in segments or profile queries)

Unomi 3.0+ uses **engine-neutral** queryBuilder IDs in JSON (for example booleanConditionQueryBuilder, not *ESQueryBuilder).

Add validation blocks on parameters so invalid instances are rejected at save time — see Condition validation.

Evaluator registration (Declarative Services)

Reference implementation: plugins/advanced-conditions (SourceEventPropertyConditionEvaluator, PastEventConditionEvaluator).

```
@Component(service = ConditionEvaluator.class,
        property = {"conditionEvaluatorId=sourceEventPropertyConditionEvaluator"})
public class SourceEventPropertyConditionEvaluator implements ConditionEvaluator {

    private DefinitionsService definitionsService;

    @Reference
    public void setDefinitionsService(DefinitionsService definitionsService) {
        this.definitionsService = definitionsService;
    }

    @Override
    public boolean eval(Condition condition, Item item, Map<String, Object> context,
                        ConditionEvaluatorDispatcher dispatcher) {
        // ...
        return false;
    }
}
```

Registration examples for query builders: Plugin registration (Unomi 3.0+).

Query builders and the persistence split

In the Apache Unomi distribution, **core** query builders for built-in conditions live in:

- persistence-elasticsearch/core — unomi-elasticsearch-conditions feature
- persistence-opensearch/core — unomi-opensearch-conditions feature

The base plugin bundle registers evaluators only; it does not register query builders.

For **custom** conditions that participate in segment queries, ship search-engine-specific query builder bundles (see next section).

Query builder implementation (Elasticsearch Java API Client)

```
package com.example.plugin.elasticsearch;

import co.elastic.clients.elasticsearch._types.query_dsl.Query;
import org.apache.unomi.api.conditions.Condition;
import org.apache.unomi.persistence.elasticsearch.ConditionESQueryBuilder;
import org.apache.unomi.persistence.elasticsearch.ConditionESQueryBuilderDispatcher;

import java.util.Map;

public class MyCustomESQueryBuilder implements ConditionESQueryBuilder {

    @Override
    public Query buildQuery(Condition condition, Map<String, Object> context,
                            ConditionESQueryBuilderDispatcher dispatcher) {
        String fieldName = (String) condition.getParameter("fieldName");
        String fieldValue = (String) condition.getParameter("fieldValue");
        return Query.of(q -> q.term(t -> t.field(fieldName).value(v -> v.stringValue(fieldValue))));
    }
}
```

OpenSearch uses the same pattern with `org.apache.unomi.persistence.opensearch.ConditionOSQueryBuilder` and `org.opensearch.client.opensearch._types.query_dsl.Query`.

Register the query builder with Declarative Services in your Elasticsearch or OpenSearch bundle:

```
@Component(service = ConditionESQueryBuilder.class, property =
{"queryBuilderId=myCustomQueryBuilder"})
public class MyCustomESQueryBuilder implements ConditionESQueryBuilder {
    // buildQuery implementation ...
}
```

Core persistence modules still contain legacy Blueprint registrations for built-in query builders; new plugin code should use DS as above.

Shared condition type JSON:

```
{
 "metadata": {
  "id": "myCustomCondition",
  "name": "My Custom Condition",
  "systemTags": ["custom", "condition", "profileCondition"],
  "readOnly": false
 },
 "conditionEvaluator": "myCustomConditionEvaluator",
 "queryBuilder": "myCustomQueryBuilder",
 "parameters": [
  { "id": "fieldName", "type": "string", "validation": { "required": true } },
  { "id": "fieldValue", "type": "string", "validation": { "required": true } }
 ]
}
```

21.4.10. Multi-bundle layout for Elasticsearch and OpenSearch

When your condition needs segment/query support on **both** search engines, use three Maven modules:

```
my-plugin/
□□□ my-plugin-common/         # JSON descriptors, evaluators, shared code
□□□ my-plugin-elasticsearch/   # ConditionESQueryBuilder + ES mappings
□□□ my-plugin-opensearch/      # ConditionOSQueryBuilder + OS mappings
□□□ my-plugin-features/        # Karaf features.xml
```

Common bundle dependencies:

```
<dependency>
   <groupId>org.apache.unomi</groupId>
   <artifactId>unomi-api</artifactId>
   <scope>provided</scope>
</dependency>
<dependency>
   <groupId>org.apache.unomi</groupId>
   <artifactId>unomi-persistence-spi</artifactId>
   <scope>provided</scope>
</dependency>
```

Elasticsearch bundle adds:

```
<dependency>
   <groupId>org.apache.unomi</groupId>
   <artifactId>unomi-persistence-elasticsearch-core</artifactId>
   <scope>provided</scope>
</dependency>
```

OpenSearch bundle adds:

```
<dependency>
  <groupId>org.apache.unomi</groupId>
  <artifactId>unomi-persistence-opensearch-core</artifactId>
  <scope>provided</scope>
</dependency>
```

Karaf features

Define **separate** features per search engine. Use the same queryBuilderId in both implementations.

```
<features name="my-plugin" version="1.0.0">
  <feature name="my-plugin-elasticsearch" version="1.0.0">
    <bundle>mvn:com.example/my-plugin-common/1.0.0</bundle>
    <bundle>mvn:com.example/my-plugin-elasticsearch/1.0.0</bundle>
    <feature version="4.0.0">unomi-elasticsearch-conditions</feature>
  </feature>
  <feature name="my-plugin-opensearch" version="1.0.0">
    <bundle>mvn:com.example/my-plugin-common/1.0.0</bundle>
    <bundle>mvn:com.example/my-plugin-opensearch/1.0.0</bundle>
    <feature version="4.0.0">unomi-opensearch-conditions</feature>
  </feature>
</features>
```

Install at runtime:

- Elasticsearch: feature:install my-plugin-elasticsearch
- OpenSearch: feature:install my-plugin-opensearch

Custom distribution feature

For production, add your plugin feature to a custom distribution based on distribution/src/main/feature/feature.xml. Extend unomi-distribution-elasticsearch or unomi-distribution-opensearch and append your plugin feature **last**.

Elasticsearch example (abbreviated — compare with the real file in the repository):

```
<feature name="unomi-distribution-elasticsearch-custom" version="${project.version}">
  <feature dependency="true" version="${project.version}">unomi-distribution-
elasticsearch</feature>
  <feature dependency="true" version="${project.version}">my-plugin-elasticsearch</feature>
</feature>
```

OpenSearch deployments use unomi-distribution-opensearch and unomi-healthcheck-opensearch instead of the Elasticsearch healthcheck feature.

See also Building and deploying and Configuration.

21.4.11. Additional bundle resources

JSON Schemas for events

Place event schemas under META-INF/cxs/schemas/. See JSON Schemas.

Index mappings and Painless scripts

- META-INF/cxs/mappings/ — custom index mappings loaded by the persistence service
- META-INF/cxs/painless/ — .painless scripts used during persistence operations

Expression filters

META-INF/cxs/expressions/ — JSON descriptors for allowed scripting expressions (MVEL and related filters).

CXS migration patch files

META-INF/cxs/patches/ — one-time structural updates. See Migration patches.

21.4.12. Custom value types and validation

Register custom parameter types for condition validation by exposing a ValueTypeValidator OSGi service. DefinitionsService forwards validators to ConditionValidationService.

Use this when your condition type introduces a non-primitive type ID in parameter definitions.

21.4.13. Scheduling background work

Plugins can register TaskExecutor implementations and schedule tasks through SchedulerService. See Task scheduler for REST APIs, shell commands, tenant-aware executors, and checkpoint patterns.

21.4.14. GraphQL provider extensions

For GraphQL schema extensions, see graphql/cxs-impl and GraphQL API.

21.4.15. Reference plugins in the codebase

Use maintained modules in the main tree — not the older samples/ demos (those predate DS migration and Blueprint removal).

Module	What to study
plugins/advanced-conditions	DS ConditionEvaluator + pom.xml with _dsannotations (start here for condition plugins)
plugins/kafka-injector	DS lifecycle (@Activate, @Deactivate, @Reference)
extensions/lists-extension/actions	Action type JSON + AddToListsAction logic

Module	What to study
graphql/cxs-impl/.../actions	DS ActionExecutor registrations
persistence-elasticsearch/core	Query builder class implementations (study querybuilders/core/)
persistence-opensearch/core	OpenSearch query builder counterparts

The samples/ directory holds historical integration demos (Twitter button, login page, etc.). They illustrate end-to-end scenarios but are **not** maintained plugin templates — copy patterns from the modules above instead.

Scenario walkthroughs that use samples: Samples.

21.4.16. Best practices

- Prefer Groovy actions or JSON-only bundles before writing Java when possible.
- Register OSGi services with **Declarative Services** (@Component); do not add new Blueprint XML.
- Import unomi-bom; use provided scope for Unomi artifacts inside Karaf.
- Use the same queryBuilder ID in JSON and in both ES/OS service registrations.
- Put evaluators and JSON descriptors in the **common** bundle; query builders in engine-specific bundles.
- Add validation metadata on condition parameters (see Condition validation).
- Test segment conditions against both search engines if you ship both query builder bundles.
- Use explain=true on context requests to debug condition evaluation — see Debugging conditions.
- Do not copy legacy *ESQueryBuilder class names from Unomi 2.x; follow the 3.0+ naming in Built-in condition types.

Integration testing with Docker

```
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:9.4.3
  opensearch:
    image: opensearchproject/opensearch:3.7.0
```

21.4.17. Reference implementations in the core codebase

Study these modules when building persistence-related plugins:

Area	Location
Elasticsearch persistence	persistence-elasticsearch/core
OpenSearch persistence	persistence-opensearch/core

Area	Location
Built-in condition evaluators (legacy Blueprint)	plugins/baseplugin — study evaluators only, not wiring
Advanced evaluators (DS, preferred)	plugins/advanced-conditions
Distribution features	distribution/src/main/feature/feature.xml

These show query building patterns, mapping layout, connection handling, and how built-in condition types are split between evaluators (plugins) and query builders (persistence modules).

21.5. Migration patches

You may provide patches on any predefined items by simply adding a JSON file in :

```
src/main/resources/META-INF/cxs/patches
```

These patches will be applied when the module will be deployed the first time. They allow to modify an item, that would have been previously deployed on unomi by a previous version of the extension or by something else.

Each patch must have a unique id - unomi will use this id to remember that the patch has already been applied. It can also be used to reapply the patch when need by using the karaf command unomi:deploy-definition

A patch also need to reference the item to patch by setting patchedItemId and patchedItemType, and an operation that tells what the patch should do.

patchedItemType *can take one of the following value:*

- condition
- action
- goal
- campaign
- persona
- propertyType
- rule
- segment
- scoring

operation *can take one of the following value:*

- patch
- override

- remove

You can apply a patch in json-patch format in the data field, and by specifying operation patch like in this example :

```
{
 "itemId": "firstName-patch1",
 "patchedItemId": "firstName",
 "patchedItemType": "propertyType",
 "operation": "patch",
 "data": [
  {
   "op": "replace", "path": "/defaultValue", "value": "foo"
  }
 ]
}
```

If you need to completely redeploy a definition, you can use the override operation and put the definition in data

```
{
 "itemId": "gender-patch1",
 "patchedItemId": "gender",
 "patchedItemType": "propertyType",
 "operation": "override",
 "data": {
  "metadata": {
   "id": "gender",
   "name": "Gender",
   "systemTags": [
    "properties",
    "profileProperties"
   ]
  },
  "type": "string",
  "defaultValue": "foo",
  "automaticMappingsFrom": [ ],
  "rank": "105.0"
 }
}
```

It is also possible to simply remove an item by using the operation remove :

```
{
 "itemId": "firstName-patch2",
 "patchedItemId": "firstName",
 "patchedItemType": "propertyType",
 "operation": "remove"
}
```

Patches can also be deployed at runtime by using the REST endpoint /patch/apply .