APACHE UNOMI 1.X - DOCUMENTATION

TABLE OF CONTENTS

1. QUICK START

1.1. FIVE MINUTES QUICKSTART

1) Install JDK 8 (http://www.oracle.com/technetwork/java/javase/downloads/index.html) and make sure you set the JAVA_HOME variable https://docs.oracle.com/cd/E19182-01/820-7851/inst_cli_jdk_javahome_t/

2) Download ElasticSearch here : https://www.elastic.co/downloads/past-releases/elasticsearch-5-6-3 (please make sure you use the proper version : 5.6.3)

3) Uncompress it and change the config/elasticsearch.yml to include the following config : <code>cluster.name: contextElasticSearch</code>

4) Launch ElasticSearch using : bin/elasticsearch

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

6) Start it using : ./bin/karaf

7) Start the Apache Unomi packages using unomi:start in the Apache Karaf Shell

8) Wait for startup to complete

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

10) Request your first context by simply accessing : http://localhost:8181/context.js?sessionId=1234

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

2. CONCEPTS

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.

2.1. ITEMS AND TYPES

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.

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.

Base Item structure:

```
{
  "itemType": <type of the item>,
  "scope": <scope>,
  "itemId": <item identifier>,
  "properties": <optional properties>
}
```

Some types can be dynamically defined at runtime by calling to the REST API while other extensions are done via Unomi plugins. Part of extending Unomi, therefore, is a matter of defining new types and specifying which kind of Unomi entity (e.g. profiles) they can be affected to. For example, the following JSON document can be passed to Unomi to declare a new property type identified (and named) tweetNb, tagged with the social tag, targeting profiles and using the integer value type.

Example JSON type definition:

```
{
  "itemId": "tweetNb",
  "itemType": "propertyType",
  "metadata": {
    "id": "tweetNb",
    "name": "tweetNb",
    "systemTags": ["social"]
  },
  "target": "profiles",
  "type": "integer"
}
```

> Unomi defines a built-in scope (called systemscope) that clients can use to share data across scopes.

2.2. EVENTS

Users' actions are conveyed from clients to the context server using events. Of course, the required information depends on what is collected and users' interactions with the observed systems but events minimally provide a type, a scope and source and target items. Additionally, events are timestamped. Conceptually, an event can be seen as a sentence, the event's type being the verb, the source the subject and the target the object.

Event structure:

```
{
  "eventType": <type of the event>,
  "scope": <scope of the event>,
  "source": <Item>,
  "target": <Item>,
  "properties": <optional properties>
}
```

Source and target can be any Unomi item but are not limited to them. In particular, as long as they can be described using properties and Unomi's type mechanism and can be processed either natively or via extension plugins, source and target can represent just about anything. Events can also be triggered as part of Unomi's internal processes for example when a rule is triggered.

Events are sent to Unomi from client applications using the JSON format and a typical page view event from a web site could look something like the following:

Example page view event:

```
{
  "eventType": "view",
  "scope": "ACMESPACE",
  "source": {
    "itemType": "site",
    "scope": "ACMESPACE",
    "itemId": "c4761bbf-d85d-432b-8a94-37e866410375"
  },
  "target": {
    "itemType": "page",
    "scope": "ACMESPACE",
    "itemId": "b6acc7b3-6b9d-4a9f-af98-54800ec13a71",
    "properties": {
      "pageInfo": {
      "pageID": "b6acc7b3-6b9d-4a9f-af98-54800ec13a71",
      "pageName": "Home",
      "pagePath": "/sites/ACMESPACE/home",
      "destinationURL": "http://localhost:8080/sites/ACMESPACE/home.html",
      "referringURL": "http://localhost:8080/",
      "language": "en"
    },
    "category": {},
    "attributes": {}
   }
  }
}
```


2.3. PROFILES

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.

2.4. SESSIONS

A session represents a time-bounded interaction between a user (via their associated profile) and a Unomi-enabled application. A session represents the sequence of actions the user performed during its duration. For this reason, events are associated with the session during which they occurred. In the context of web applications, sessions are usually linked to HTTP sessions.

2.5. SEGMENTS

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.

Here is an example of a simple segment definition registered using the REST API:

```
curl -X POST http://localhost:8181/cxs/segments \
--user karaf:karaf \
-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.

2.6. CONDITIONS

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

A condition is composed of two basic elements:

- a condition type identifier
- a list of parameter values for the condition, that can be of any type, and in some cases may include sub-conditions

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. The parameter values are simply lists of objects 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. 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.

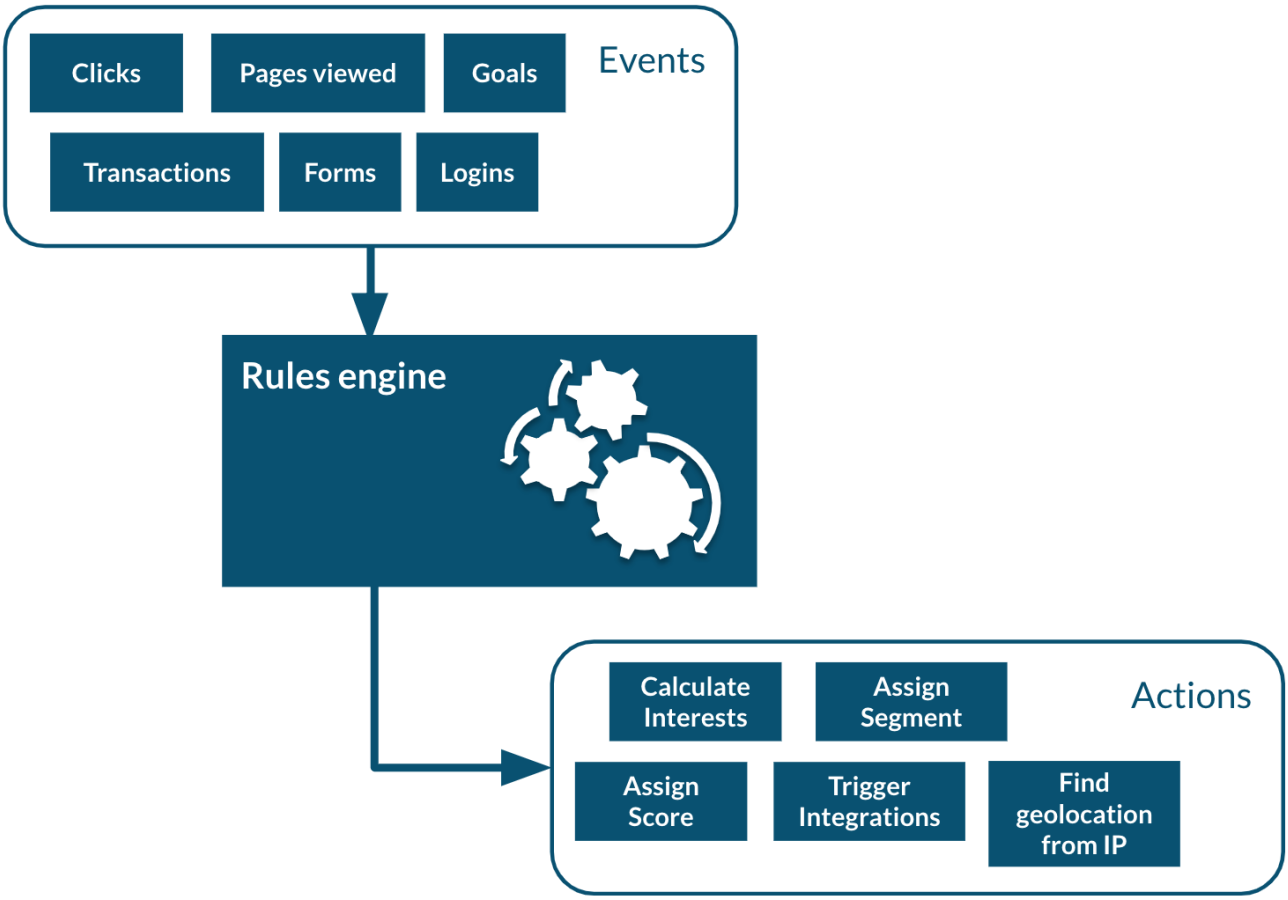
Here is an example of a complex condition:

```
{
 "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.

For a more complete list of available conditions, see the Built-in conditions reference section.

2.7. RULES



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 pluggeable 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 systesm, 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.

2.7.1. ACTIONS

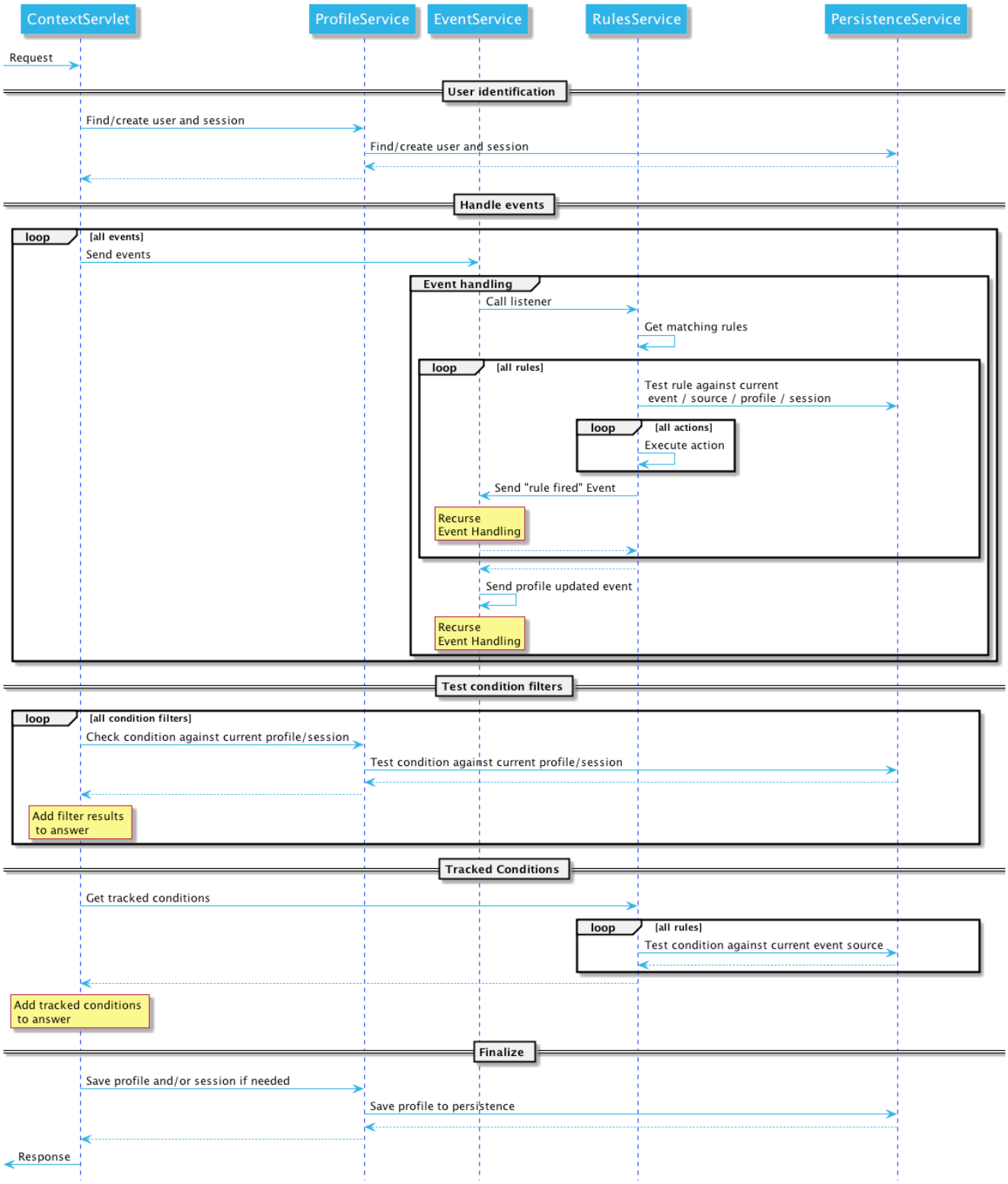
Actions are executed by rules in a sequence, and an action is only executed once the previous action has finished executing. If an action generates an exception, it will be logged and the execution sequence will continue unless in the case of a Runtime exception (such as a NullPointerException).

Actions 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 actions. You may find the list of built-in actions in the Built-in actions section.

2.8. REQUEST FLOW

Here is an overview of how Unomi processes incoming requests to the ContextServlet.



3. FIRST STEPS WITH APACHE UNOMI

3.1. 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.

3.1.1. PREREQUISITES

This document assumes that you are already familiar with Unomi's concepts. On the technical side, we also assume working knowledge of git to be able to retrieve the code for Unomi and the example. Additionally, you will require a working Java 7 or above install. Refer to http://www.oracle.com/technetwork/java/javase/ for details on how to download and install Java SE 7 or greater.

3.1.2. RUNNING UNOMI

START UNOMI

Start Unomi according to the 5 minute quick start 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. You can then open a browser and go to http://localhost:8181/cxs to see the list of available RESTful services or retrieve an initial context at http://localhost:8181/context.json (which isn't very useful at this point).

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.

3.2. RECIPES

3.2.1. INTRODUCTION

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

3.2.2. HOW TO READ A PROFILE

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

Here is an example that will retrieve all the session and profile properties.

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

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 /context.json endpoint for consistency and security.

3.2.3. 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 and you're ready to update profiles using events. This is also the safest way to update a profile because if you design your events to be as specific as possible to your needs, only the data that you specified will be copied to the profile, making sure that even in the case an attacker tries to send more data using your custom event it will simply be ignored.
- Use the protected built-in "updateProperties" event. This event is designed to be used for administrative purposes only. Again, prefer the custom events solution because as this is a protected event it will require sending the Unomi key as a request header, 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.

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 karaf:karaf \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "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"
      }
    }
  ]
}
EOF
```

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'.

You could then simply send the contactInfoSubmitted event using a request similar to this one:

```
curl -X POST http://localhost:8181/eventcollector \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
   "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"
         }
      }
   ]
}
EOF
```

3.2.4. 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 karaf:karaf \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{ "offset" : 0,
  "limit" : 20,
  "condition" : {
    "type": "eventPropertyCondition",
    "parameterValues" : {
      "propertyName" : "profileId",
      "comparisonOperator" : "equals",
      "propertyValue" : "PROFILE_ID"
    }
  }
}
EOF
```

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

3.2.5. 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 karaf:karaf \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "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"
    }
  ]
}
EOF
```

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.

3.2.6. 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 karaf:karaf \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
  "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"
          }
        }
      ]
    }
  }
}
EOF
```

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.

3.2.7. GETTING / UPDATING CONSENTS

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

3.2.8. 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. You can find it 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 !

3.3. REQUEST EXAMPLES

3.3.1. RETRIEVING YOUR FIRST CONTEXT

You can retrieve a context using curl like this :

```
curl http://localhost:8181/context.js?sessionId=1234
```

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).

3.3.2. 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/context.json?sessionId=1234
```

3.3.3. 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.

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

3.3.4. 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/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
   "source":{
      "itemId":"homepage",
      "itemType":"page",
      "scope":"example"
   },
   "events":[
      {
         "eventType":"view",
         "scope": "example",
         "source":{
            "itemType": "site",
            "scope":"example",
            "itemId": "mysite"
         },
         "target":{
            "itemType":"page",
            "scope":"example",
            "itemId":"homepage",
            "properties":{
               "pageInfo":{
                  "referringURL":""
               }
            }
         }
      }
   ]
}
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).

3.3.5. 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/eventcollector \
-H "Content-Type: application/json" \
-d @- <<'EOF'
{
   "sessionId" : "1234",
   "events":[
      {
         "eventType":"view",
         "scope": "example",
         "source":{
            "itemType": "site",
            "scope":"example",
            "itemId": "mysite"
         },
         "target":{
            "itemType":"page",
            "scope":"example",
            "itemId":"homepage",
            "properties":{
               "pageInfo":{
                  "referringURL":""
               }
            }
         }
      }
   ]
}
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.

3.3.6. 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.
- You may want to know integrate the provided web tracker into your web site.
- Read the Twitter sample documentation that contains a detailed example of how to integrate with Apache Unomi.

3.4. WEB TRACKER

This extension is providing the web tracker to start collecting visitors data on your website. The tracker is implemented as an integration of analytics.js for Unomi.

3.4.1. GETTING STARTED

Extension can be tested at : http://localhost:8181/tracker/index.html

In your page include unomiOptions and include code snippet from snippet.min.js :

```
<script type="text/javascript">
      var unomiOption = {
         scope: 'realEstateManager',
         url: 'http://localhost:8181'
      };
      window.unomiTracker||(window.unomiTracker={}),function(){function
e(e){for(unomiTracker.initialize({"Apache Unomi":unomiOption});n.length>0;){var
r=n.shift(),t=r.shift();unomiTracker[t]&&unomiTracker[t].apply(unomiTracker,r)}}for(var
n=[],r=["trackSubmit","trackClick","trackLink","trackForm","initialize","pageview","identify","reset","
group","track","ready","alias","debug","page","once","off","on","personalize"],t=0;t<r.length;t++){var
i=r[t];window.unomiTracker[i]=function(e){return function(){var
r=Array.prototype.slice.call(arguments);return
r.unshift(e),n.push(r),window.unomiTracker}}(i)}unomiTracker.load=function(){var
n=document.createElement("script");n.type="text/javascript",n.async=!0,n.src=unomiOption.url+"/tr
acker/unomi-
tracker.min.js",n.addEventListener?n.addEventListener("load",function(n){"function"==typeof
e&&e(n)},!1):n.onreadystatechange=function(){"complete"!==this.readyState&&"loaded"!==this.ready
State||e(window.event)};var
r=document.getElementsByTagName("script")[0];r.parentNode.insertBefore(n,r)},document.addEve
ntListener("DOMContentLoaded",unomiTracker.load),unomiTracker.page()}();
</script>
```

window.unomiTracker can be used to send additional events when needed.

Check analytics.js API here. All methods can be used on unomiTracker object, although not all event types are supported by Unomi intergation.

3.4.2. HOW TO CONTRIBUTE

The source code is in the folder javascript with a package.json, the file to update is analytics.js-integration-apache-unomi.js apply your modification in this file then use the command yarn build to compile a new JS file. Then you can use the test page to try your changes http://localhost:8181/tracker/index.html.

3.4.3. TRACKING PAGE VIEWS

By default the script will track page views, but maybe you want to take control over this mechanism of add page views to a single page application. In order to generate a page view programmatically from Javascript you can use code similar to this :

```
    <script type="text/javascript">
        // This is an example of how to provide more details page properties to the view event. This can
be useful
        // in the case of an SPA that wants to provide information about a view that has metadata such
as categories,
        // tags or interests.
        path = location.pathname + location.hash;
        properties = {
            path: path,
            pageInfo: {
                destinationURL: location.href,
                tags : [ "tag1", "tag2", "tag3"],
                categories : ["category1", "category2", "category3"],
            },
            interests : {
                "interest1" : 1,
                "interest2" : 2,
                "interest3" : 3
            }
        };
        console.log(properties);
        // this will trigger a second page view for the same page (the first page view is in the tracker
snippet).
        window.unomiTracker.page(properties);
    </script>
```

Here is a more detail view of what you may include in the pageInfo object :

Table 1. PageInfo Properties

Name	Description
pageID	A unique identifier in string format for the page. Default value : page path
pageName	A user-displayed name for the page. Default value : page title
pagePath	The path of the page, stored by Unomi. This value should be the same as the one passed in the page property of the object passed to the unomiTracker call. Default value : page path
destinationURL	The full URL for the page view. This doesn't have to be a real existing URL it could be an internal SPA route. Default value : page URL
referringURL	The referringURL also known as the previous URL of the page/screen viewed. Default value : page referrer URL
tags	A String array of tag identifiers. For example ['tag1', 'tag2', 'tag3']

Name	Description
categories	A String array of category identifiers. For example ['category1', 'category2', 'category3']

The interests object is basically list of interests with "weights" attached to them. These interests will be accumulated in Apache Unomi on profiles to indicate growing interest over time for specific topics. These are freely defined and will be accepted by Apache Unomi without needing to declare them previously anywhere (the same is true for tags and categories).

3.4.4. TRACKING FORM SUBMISSIONS

Using the web tracker you can also track form submissions. In order to do this a few steps are required to get a form's submission to be tracked and then its form values to be sent as events to Apache Unomi. Finally setting up a rule to react to the incoming event will help use the form values to perform any action that is desired.

Let's look at a concrete example. Before we get started you should know that this example is already available to directly test in Apache Unomi at the following URL :

```
http://localhost:8181/tracker
```

Simply modify the form values and click submit and it will perform all the steps we are describing below.

So here is the form we want to track :

```
<form id="testFormTracking" action="#" name="testFormTracking">
  <label for="firstName">First name</label>
  <input type="text" id="firstName" name="firstName" value="John"/>

  <label for="lastName">Last name</label>
  <input type="text" id="lastName" name="lastName" value="Doe"/>

  <label for="email">Email</label>
  <input type="email" id="email" name="email" value="johndoe@acme.com"/>

  <input type="submit" name="submitButton" value="Submit"/>
</form>
```

As you can see it's composed of three fields - firstName, lastName and email - as well as a submit button. In order to track it we can add directly under the following snippet :

```
<script type="text/javascript">
   window.addEventListener("load", function () {
      var form = document.getElementById('testFormTracking');
      unomiTracker.trackForm(form, 'formSubmitted', {formName: form.name});
   });
</script>
```

What this snippet does is retrieve the form using its element ID and then uses the unomiTracker to track form submissions. Be careful to always use in the form event name a string that starts with form in order for the event to be sent back to Unomi. Also the form name is also a mandatory parameter that will be passed to Unomi inside a event of type form under the target.itemId property name.

Here is an example of the event that gets sent back to Apache Unomi:

```
{
  "itemId" : "cd627012-963e-4bb5-97f0-480990b41254",
  "itemType" : "event",
  "scope" : "realEstateManager",
  "version" : 1,
  "eventType" : "form",
  "sessionId" : "aaad09aa-88c2-67bd-b106-5a47ded43ead",
  "profileId" : "48563fd0-6319-4260-8dba-ae421beba26f",
  "timeStamp" : "2018-11-23T16:32:26Z",
  "properties" : {
    "firstName" : "John",
    "lastName" : "Doe",
    "email" : "johndoe@acme.com",
    "submitButton" : "Submit"
  },
  "source" : {
    "itemId" : "/tracker/",
    "itemType" : "page",
    "scope" : "realEstateManager",
    "version" : null,
    "properties" : {
      "pageInfo" : {
        "destinationURL" :
"http://localhost:8181/tracker/?firstName=Bill&lastName=Gates&email=bgates%40microsoft.com",
        "pageID" : "/tracker/",
        "pagePath" : "/tracker/",
        "pageName" : "Apache Unomi Web Tracker Test Page",
        "referringURL" :
"http://localhost:8181/tracker/?firstName=John&lastName=Doe&email=johndoe%40acme.com"
      },
      "attributes" : [ ],
      "consentTypes" : [ ],
      "interests" : { }
    }
  },
  "target" : {
    "itemId" : "testFormTracking",
    "itemType" : "form",
    "scope" : "realEstateManager",
    "version" : null,
    "properties" : { }
  },
  "persistent" : true
}
```


You can see in this event that the form values are sent as properties of the event itself, while the form name is sent as the target.itemId

While setting up form tracking, it can be very useful to use the Apache Unomi Karaf SSH shell commands : event-tail and event-view to check if you are properly receiving the form submission events and that they contain the expected data. If not, check your tracking code for any errors.

Now that the data is properly sent using an event to Apache Unomi, we must still use it to perform some kind of actions. Using rules, we could do anything from updating the profile to sending the data to a

third-party server (using a custom- developped action of course). In this example we will illustrate how to update the profile.

In order to do so we will deploy a rule that will copy data coming from the event into a profile. But we will need to map the form field names to profile names, and this can be done using the setPropertyAction that's available out of the box in the Apache Unomi server.

There are two ways to register rules : either by building a custom OSGi bundle plugin or using the REST API to directly send a JSON representation of the rule to be saved. We will in this example use the CURL shell command to make a call to the REST API.

```
curl -X POST -k -u karaf:karaf https://localhost:9443/cxs/rules \
  --header "Content-Type: application/json" \
-d @- << EOF
{
  "itemId": "form-mapping-example",
  "itemType": "rule",
  "linkedItems": null,
  "raiseEventOnlyOnceForProfile": false,
  "raiseEventOnlyOnceForSession": false,
  "priority": -1,
  "metadata": {
    "id": "form-mapping-example",
    "name": "Example Form Mapping",
    "description": "An example of how to map event properties to profile properties",
    "scope": "realEstateManager",
    "tags": [],
    "enabled": true,
    "missingPlugins": false,
    "hidden": false,
    "readOnly": false
  },
  "condition": {
    "type": "formEventCondition",
    "parameterValues": {
      "formId": "testFormTracking",
      "pagePath" : "/tracker/"
    }
  },
  "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"
      }
    }
  ]
}
EOF
```

As you can see in this request, we have a few parameters that need explaining:

- -k is used to accept any certificate as we are in this example using a default Apache Unomi server configuration that comes with its predefined HTTPS certificates
- -u karaf:karaf is the default username/password for authenticating to the REST API. To change this value you should edit the `etc/users.properties`file and it is required to modify this login before going to production.

Finally the rule itself should be pretty self-explanatory but there are a few important things to note :

- the itemId and metadata.id values should be the same
- the scope should be the same as the scope that was setup in the tracker initialization
- the formId parameter must have the form name value
- the pagePath should be the pagePath passed through the event (if you're not sure of its value, you could either using network debugging in the browser or use the event-tail and event-view commands in the Apache Unomi Karaf SSH shell).
- the setPropertyAction may be repeated as many times as desired to copy the values from the event to the profile. Note that the setPropertyName will define the property to set on the profile and the setPropertyValue will define where the value is coming from. In this example the name and the value are the same but that is no way a requirement. It could even be possible to using multiple setPropertyAction instances to copy the same event property into different profile properties.

To check if your rule is properly deployed you can use the following SSH shell command :

unomi:rule-view form-mapping-example

The parameter is the itemId of the rule. If you want to see all the rules deployed in the system you can use the command :

unomi:rule-list 1000

The 1000 parameter is the limit of number of objects to retrieve. As the number of rules can grow quickly in an Apache Unomi instance, it is recommended to put this value a bit high to make sure you get the full list of rules.

Once the rule is in place, try submitting the form with some values and check that the profile is properly updated. One recommend way of doing this is to use the event-tail command that will output something like this :

```
ID                                   |Type        |Session                               |Profile                               |Timestamp
                    |Scope        |Persi|
------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------
cef09b89-6b99-4e4f-a99c-a4159a66b42b|form        |aaad09aa-88c2-67bd-b106-
5a47ded43ead|48563fd0-6319-4260-8dba-ae421beba26f|Fri Nov 23 17:52:33 CET 2018
|realEstateManag|true |
```

You can directly see the profile that is being used, so you can then simply use the

unomi:profile-view 48563fd0-6319-4260-8dba-ae421beba26f

command to see a JSON dump of the profile and check that the form values have been properly positioned.

3.5. CONFIGURATION

3.5.1. 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.

3.5.2. 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.

3.5.3. 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 an ElasticSearch cluster name, or a host and port that are different than the default, it is recommended to do this BEFORE you start the server for the first time, or you will loose all the data you have stored previously.

You can use the following properties for the ElasticSearch configuration

```
org.apache.unomi.elasticsearch.cluster.name=contextElasticSearch
# The elasticsearch.adresses may be a comma seperated 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
```

3.5.4. SECURED EVENTS CONFIGURATION

Apache Unomi secures some events by default. It comes out of the box with a default configuration that you can adjust by using 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_IPADDRESSES:-127.0.0.1,::1}
org.apache.unomi.thirdparty.provider1.allowedEvents=${env:UNOMI_THIRDPARTY_PROVIDER1_ALLOWEDEVENTS:-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-key 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_IPADDRESSES:-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_IPADDRESSES:-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_IPADDRESSES:-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
```

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=secret-key
thirdparty.provider1.ipAddresses=127.0.0.1,::1
thirdparty.provider1.allowedEvents=login,updateProperties
```

3.5.5. 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.

3.5.6. 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.

3.5.7. REST API SECURITY

The Apache Unomi Context Server REST API is protected using JAAS authentication and using Basic or Digest HTTP auth. By default, the login/password for the REST API full administrative access is "karaf/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 default Karaf password can be done by modifying the org.apache.unomi.security.root.password in the $MY_KARAF_HOME/etc/unomi.custom.system.properties file

3.5.8. 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.

Upon merge, the old profiles are marked with a "mergedWith" property that will be used on next profile access to delete the original profile and replace it with the merged profile (aka "master" profile). Once this is done, all cookie tracking will use the merged profile.

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.

3.5.9. 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/context.js
- http://localhost:8181/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 :

- http://karaf.apache.org/manual/latest/users-guide/security.html

The default username/password is

```
karaf/karaf
```

You should really change this default username/password as soon as possible. Changing the default Karaf password can be done by modifying the org.apache.unomi.security.root.password in the $MY_KARAF_HOME/etc/unomi.custom.system.properties file

Or if you want to also change the user name you could modify the following file :

```
$MY_KARAF_HOME/etc/users.properties
```

But you will also need to change the following property in the $MY_KARAF_HOME/etc/unomi.custom.system.properties :

```
karaf.local.user = karaf
```

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.

3.5.10. 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/
```

3.5.11. 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.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.

3.5.12. 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.

3.5.13. ELASTICSEARCH X-PACK SUPPORT

It is now possible to use X-Pack to connect to ElasticSearch. However, for licensing reasons this is not provided out of the box. Here is the procedure to install X-Pack with Apache Unomi:

IMPORTANT !

Do not start Unomi directly with unomi:start, perform the following steps below first !

INSTALLATION STEPS

1. Create a directory for all the JARs that you will download, we will call it XPACK_JARS_DIRECTORY
2. Download https://artifacts.elastic.co/maven/org/elasticsearch/client/x-pack-transport/5.6.3/x-pack-transport-5.6.3.jar to XPACK_JARS_DIRECTORY
3. Download https://artifacts.elastic.co/maven/org/elasticsearch/plugin/x-pack-api/5.6.3/x-pack-api-5.6.3.jar to XPACK_JARS_DIRECTORY
4. Download http://central.maven.org/maven2/com/unboundid/unboundid-ldapsdk/3.2.0/unboundid-ldapsdk-3.2.0.jar to XPACK_JARS_DIRECTORY
5. Download http://central.maven.org/maven2/org/bouncycastle/bcpkix-jdk15on/1.55/bcpkix-jdk15on-1.55.jar to XPACK_JARS_DIRECTORY

6. Download http://central.maven.org/maven2/org/bouncycastle/bcprov-jdk15on/1.55/bcprov-jdk15on-1.55.jar to XPACK_JARS_DIRECTORY
7. Download http://central.maven.org/maven2/com/sun/mail/javax.mail/1.5.3/javax.mail-1.5.3.jar to XPACK_JARS_DIRECTORY .

Edit etc/org.apache.unomi.persistence.elasticsearch.cfg to add the following settings:

```
transportClientClassName=org.elasticsearch.xpack.client.PreBuiltXPackTransportClient
transportClientJarDirectory=XPACK_JARS_DIRECTORY
transportClientProperties=xpack.security.user=elastic:changeme
```

You can setup more properties (for example for SSL/TLS support) by seperating the properties with commas, as in the following example:

```
transportClientProperties=xpack.security.user=elastic:changeme,xpack.ssl.key=/home/user/elasticsearch-5.6.3/config/x-pack/localhost/localhost.key,xpack.ssl.certificate=/home/user/elasticsearch-5.6.3/config/x-pack/localhost/localhost.crt,xpack.ssl.certificate_authorities=/home/user/elasticsearch-5.6.3/config/x-pack/ca/ca.crt,xpack.security.transport.ssl.enabled=true
```

.

Launch Karaf and launch unomi using the command from the shell :

```
unomi:start
```

Alternatively you could edit the configuration directly from the Karaf shell using the following commands:

```
config:edit org.apache.unomi.persistence.elasticsearch
config:property-set transportClientClassName
org.elasticsearch.xpack.client.PreBuiltXPackTransportClient
config:property-set transportClientJarDirectory XPACK_JARS_DIRECTORY
config:property-set transportClientProperties xpack.security.user=elastic:changeme
config:update
unomi:start
```

You can setup more properties (for example for SSL/TLS support) by seperating the properties with commas, as in the following example:

```
config:property-set transportClientProperties
xpack.security.user=elastic:changeme,xpack.ssl.key=/home/user/elasticsearch-5.6.3/config/x-pack/localhost/localhost.key,xpack.ssl.certificate=/home/user/elasticsearch-5.6.3/config/x-pack/localhost/localhost.crt,xpack.ssl.certificate_authorities=/home/user/elasticsearch-5.6.3/config/x-pack/ca/ca.crt,xpack.security.transport.ssl.enabled=true
```

3.6. 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 2. 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
/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.

Path	Method	Description
/cxs/rules/statistics	GET	Get all rule execution statistics
/cxs/rules/statistics	DELETE	Reset all rule execution statistics to 0

3.7. HOW PROFILE TRACKING WORKS

In this section you will learn how Apache Unomi keeps track of visitors.

3.7.1. STEPS

1. A visitor comes to a website
2. The web server resolves a previous request session ID if it exists, or if it doesn't it create a new sessionID
3. A request to Apache Unomi's /context.json servlet is made passing the web server session ID as a query parameter
4. Unomi uses the sessionID and tries to load an existing session, if none is found a new session is created with the ID passed by the web server
5. If a session was found, the profile ID is extracted from the session and if it not found, Unomi looks for a cookie called context-profile-id to read the profileID. If no profileID is found or if the session didn't exist, a new profile ID is created by Apache Unomi
6. If the profile ID existed, the corresponding profile is loaded by Apache Unomi, otherwise a new profile is created
7. If events were passed along with the request to the context.json endpoint, they are processed against the profile
8. The updated profile is sent back as a response to the context.json request. Along with the response

It is important to note that the profileID is always server-generated. Injecting a custom cookie with a non-valid profile ID will result in failure to load the profile. Profile ID are UUIDs, which make them (pretty) safe from brute- forcing.

4. 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.

In this section we will show examples of requests that may be built using this API.

4.1. 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 ElasticSearch/Lucene optimizations 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 karaf:karaf \
-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
```

4.2. METRICS

Metric queries make it possible to apply functions to the resulting property. The supported metrics are:

- sum
- avg
- min

- max

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 karaf:karaf \
-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
}
```

4.3. 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)

4.3.1. 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. For more information about the format, it is directly inherited from ElasticSearch and you may find it here: https://www.elastic.co/guide/en/elasticsearch/reference/5.6/search-aggregations-bucket-datehistogram-aggregation.html

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 karaf:karaf \
-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.

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 karaf:karaf \
-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
}
```


You can find more information about the date range formats here: https://www.elastic.co/guide/en/elasticsearch/reference/5.6/search-aggregations-bucket-daterange-aggregation.html

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 karaf:karaf \
-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
}
```


5. 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.

5.1. 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.

5.1.1. 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
  ...
}
```

5.2. 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.

5.2.1. 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
   ...
}
```

5.3. 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
```

Final 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
```

6. CONSENT MANAGEMENT

6.1. 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.

6.1.1. 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/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-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.

6.1.2. 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.

6.1.3. 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/context.json?sessionId=1234 \
-H "Content-Type: application/json" \
-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
```

6.1.4. 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.

7. 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.

7.1. 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

7.2. 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.

Here's an example of a request to anonymize a profile:

```
curl -X POST http://localhost:8181/cxs/profiles/{profileID}/anonymize?scope=ASCOPE
```

where {profileID} must be replaced by the actual identifier of a profile and ASCOPE must be replaced by a scope identifier.

7.3. 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/client/myprofile.[json,csv,yaml,text] \
--cookie "context-profile-id=PROFILE-ID"
```

where PROFILE-ID is the profile identifier for which to download the profile.

7.4. 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/profiles/{profileID}?withData=false --user karaf:karaf
```

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

7.5. RELATED

You might also be interested in the Consent API section that describe how to manage profile consents.

8. CLUSTER SETUP

8.1. CLUSTER SETUP

Apache Karaf relies on Apache Karaf Cellar, which in turn uses Hazelcast to discover and configure its cluster.

You can control most of the important clustering settings through the centralized configuration file at

```
etc/unomi.custom.system.properties
```

And notably using the following properties:

```
org.apache.unomi.hazelcast.group.name=${env:UNOMI_HAZELCAST_GROUP_NAME:-cellar}
org.apache.unomi.hazelcast.group.password=${env:UNOMI_HAZELCAST_GROUP_PASSWORD:-pass}
# This list can be comma separated and use ranges such as 192.168.1.0-7,192.168.1.21
org.apache.unomi.hazelcast.tcp-ip.members=${env:UNOMI_HAZELCAST_TCPIP_MEMBERS:-
127.0.0.1}
org.apache.unomi.hazelcast.tcp-ip.interface=${env:UNOMI_HAZELCAST_TCPIP_INTERFACE:-
127.0.0.1}
org.apache.unomi.hazelcast.network.port=${env:UNOMI_HAZELCAST_NETWORK_PORT:-5701}
```

If you need more fine-grained control over the Hazelcast configuration you could also edit the following file:

```
etc/hazelcast.xml
```

Note that it would be best to keep all configuration in the centralized custom configuration, for example by adding placeholders in the hazelcast.xml file if need be and adding the properties to the centralized configuration file.

9. REFERENCE

9.1. BUILT-IN CONDITIONS

Apache Unomi comes with an extensive collection of built-in conditions. Instead of detailing them one by one you will find here an overview of what a JSON condition descriptor looks like:

```
{
  "metadata": {
    "id": "booleanCondition",
    "name": "booleanCondition",
    "description": "",
    "systemTags": [
      "profileTags",
      "logical",
      "condition",
      "profileCondition",
      "eventCondition",
      "sessionCondition",
      "sourceEventCondition"
    ],
    "readOnly": true
  },
  "conditionEvaluator": "booleanConditionEvaluator",
  "queryBuilder": "booleanConditionESQueryBuilder",
  "parameters": [
    {
      "id": "operator",
      "type": "String",
      "multivalued": false,
      "defaultValue": "and"
    },
    {
      "id": "subConditions",
      "type": "Condition",
      "multivalued": true
    }
  ]
}
```

Note that conditions have two important identifiers:

- conditionEvaluator
- queryBuilder

This is because conditions can either be used to build queries or to evaluate a condition in real time. When implementing a new condition type, both implementations much be provided. Here's an example an OSGi Blueprint registration for the above condition descriptor:

From https://github.com/apache/unomi/blob/master/plugins/baseplugin/src/main/resources/OSGI-INF/blueprint/blueprint.xml

```
...
   <service

interface="org.apache.unomi.persistence.elasticsearch.conditions.ConditionESQueryBuilder">
      <service-properties>
         <entry key="queryBuilderId" value="booleanConditionESQueryBuilder"/>
      </service-properties>
      <bean
class="org.apache.unomi.plugins.baseplugin.conditions.BooleanConditionESQueryBuilder"/>
   </service>
...
   <!-- Condition evaluators -->
   <service interface="org.apache.unomi.persistence.elasticsearch.conditions.ConditionEvaluator">
      <service-properties>
         <entry key="conditionEvaluatorId" value="booleanConditionEvaluator"/>
      </service-properties>
      <bean class="org.apache.unomi.plugins.baseplugin.conditions.BooleanConditionEvaluator"/>
   </service>
...
```

As you can see two Java classes are used to build a single condition. You don't need to understand all these details in order to use conditions, but this might be interesting to know if you're interested in building your own condition implementations. For more details on building your own custom plugins/extensions, please refer to the corresponding sections.

9.1.1. EXISTING CONDITION DESCRIPTORS

Here is a non-exhaustive list of conditions built into Apache Unomi. Feel free to browse the source code if you want to discover more. But the list below should get you started with the most useful conditions:

- https://github.com/apache/unomi/tree/master/plugins/baseplugin/src/main/resources/META-INF/cxs/conditions

Of course it is also possible to build your own custom conditions by developing custom Unomi plugins/extensions.

You will also note that some conditions can re-use a parentCondition. This is a way to inherit from another condition type to make them more specific.

9.2. BUILT-IN ACTIONS

Unomi comes with quite a lot of built-in actions. Instead of detailing them one by one you will find here an overview of what an action descriptor looks like:

```
{
 "metadata": {
  "id": "UNIQUE_IDENTIFIER_STRING",
  "name": "DISPLAYABLE_ACTION_NAME",
  "description": "DISPLAYABLE_ACTION_DESCRIPTION",
  "systemTags": [
   "profileTags",
   "event",
   "availableToEndUser",
   "allowMultipleInstances"
  ],
  "readOnly": true
 },
 "actionExecutor": "ACTION_EXECUTOR_ID",
 "parameters": [
   ... parameters specific to each action ...
 ]
}
```

The ACTION_EXECUTOR_ID points to a OSGi Blueprint parameter that is defined when implementing the action in a plugin. Here's an example of such a registration:

From https://github.com/apache/unomi/blob/master/plugins/mail/src/main/resources/OSGI-INF/blueprint/blueprint.xml

```
  <bean id="sendMailActionImpl" class="org.apache.unomi.plugins.mail.actions.SendMailAction">
    <!-- ... bean properties ... -->
  </bean>
  <service id="sendMailAction" ref="sendMailActionImpl"
interface="org.apache.unomi.api.actions.ActionExecutor">
    <service-properties>
      <entry key="actionExecutorId" value="sendMail"/>
    </service-properties>
  </service>
```

In the above example the ACTION_EXECUTOR_ID is sendMail

9.2.1. EXISTING ACTIONS DESCRIPTORS

Here is a non-exhaustive list of actions built into Apache Unomi. Feel free to browse the source code if you want to discover more. But the list below should get you started with the most useful actions:

- https://github.com/apache/unomi/tree/master/plugins/baseplugin/src/main/resources/META-INF/cxs/actions
- https://github.com/apache/unomi/tree/master/plugins/request/src/main/resources/META-INF/cxs/actions
- https://github.com/apache/unomi/tree/master/plugins/mail/src/main/resources/META-INF/cxs/actions

Of course it is also possible to build your own custom actions by developing custom Unomi plugins/extensions.

10. INTEGRATION SAMPLES

10.1. SAMPLES

Apache Unomi provides the following samples:

- Twitter integration
- Login integration

10.2. LOGIN SAMPLE

This samples is an example of what is involved in integrated a login with Apache Unomi.

10.2.1. WARNING !

The example code uses client-side Javascript code to send the login event. This is only done this way for the sake of samples simplicity but if should NEVER BE DONE THIS WAY in real cases.

The login event should always be sent from the server performing the actual login since it must only be sent if the user has authenticated properly, and only the authentication server can validate this.

10.2.2. INSTALLING THE SAMPLES

Login into the Unomi Karaf SSH shell using something like this :

```
ssh -p 8102 karaf@localhost (default password is karaf)
```

Install the login samples using the following command:

```
bundle:install mvn:org.apache.unomi/login-integration-samples/${project.version}
```

when the bundle is successfully install you will get an bundle ID back we will call it BUNDLE_ID.

You can then do:

```
bundle:start BUNDLE_ID
```

If all went well you can access the login samples HTML page here :

```
http://localhost:8181/login/index.html
```

You can fill in the form to test it. Note that the hardcoded password is:

```
test1234
```

10.3. TWITTER SAMPLE

10.3.1. 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-1.5.0-SNAPSHOT.jar ../../package/target/unomi-1.5.0-SNAPSHOT/deploy
```

TESTING THE SAMPLES

You can now go to http://localhost:8181/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.

10.3.2. 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.

10.3.3. 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>/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/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 personalization filters>,
  "trackedConditions":<tracked conditions in the source page, if any>
}
```

We will look at the details of the context request and response later.

10.4. EXAMPLE

10.4.1. 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.

10.4.2. 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/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.
- 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 <code>ContextServlet</code> using the default install URI, specifying the session identifier: <code>http://localhost:8181/context.json?sessionId=&#39; + sessionId</code>. This URI requests context from Unomi, resulting in an updated <code>cxs</code> 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 <code>context.json</code>. However, if the client requests <code>context.js</code> then useful functions to interact with Unomi are added to the <code>cxs</code> 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:

```
{
  source: <Item source of the context request>,
  events: <optional array of triggered events>,
  requiredProfileProperties: <optional array of property identifiers>,
  requiredSessionProperties: <optional array of property identifiers>,
  filters: <optional array of filters 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.

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:

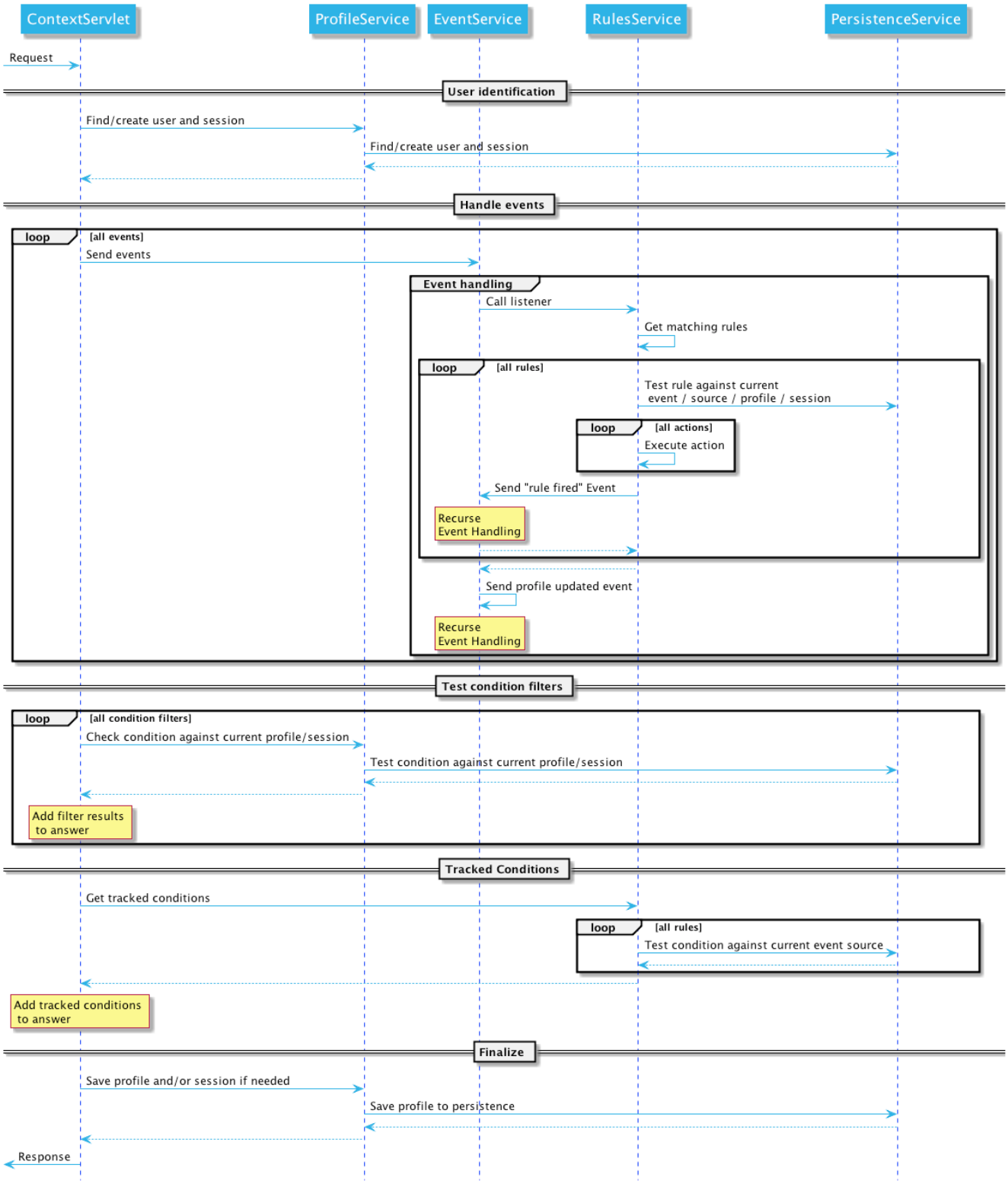
```
private String extractSourceURL(Event event) {
    final Item sourceAsItem = event.getSource();
    if (sourceAsItem instanceof CustomItem) {
        CustomItem source = (CustomItem) sourceAsItem;
        final String url = (String) source.getProperties().get("url");
        if (url != null) {
            return url;
        }
    }

    return null;
}
```

10.5. 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.

10.6. ANNEX

Here is an overview of how Unomi processes incoming requests to the ContextServlet.



10.7. WEATHER UPDATE SAMPLE

11. CONNECTORS

11.1. CONNECTORS

Apache Unomi provides the following connectors:

- Salesforce CRM connector
- Mailchimp connector

11.1.1. 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.

11.2. 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.

11.2.1. 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 (default password is karaf)
```

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. You can find this user

and password information in the etc/users.properties file.

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 Concepts and the Getting Started pages.

11.2.2. 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}
```

11.2.3. 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 (password by default is karaf)
```

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 a password it's the same karaf/karaf default)

11.2.4. 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
```

11.2.5. 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
```

11.2.6. 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)

11.3. MAILCHIMP CONNECTOR

This extension has 3 actions:

- Add a visitor into a defined Mailchimp list.
- Remove a visitor from a defined Mailchimp list.
- Unsubscribe a visitor from a defined Mailchimp list.

11.3.1. GETTING STARTED

1. Create a new MailChimp account: https://login.mailchimp.com/signup/
2. Generate a new API Key, or get the default: https://usX.admin.mailchimp.com/account/api/
3. Configure the MailChimp Connector Basic in the etc/unomi.custom.system.properties file and add/change the following settings:

```
org.apache.unomi.mailchimp.apiKey=${env:UNOMI_MAILCHIMP_APIKEY:-yourApiKey}
org.apache.unomi.mailchimp.url.subDomain=${env:UNOMI_MAILCHIMP_URL_SUBDOMAIN:-us16}
```

4. Before starting configure the mapping between Apache Unomi profile properties and MailChimp member properties.
   The mapping can't be use with multitued properties. You need to setup your MailChimp properties first in the MailChimp administration.

```
Go to: lists/
Select the triggered list
Settings
```

   Then in the cfg file org.apache.unomi.mailchimp.list.merge-fields.activate={Boolean} if you like to activate the mapping feature.

   This is the property to configure for the mapping, the format is as shown. org.apache.unomi.mailchimp.list.merge-fields.mapping={Apache Unomi property ID}<□{MailChimp Tag name}

   there is a particular format for the address {Apache Unomi property ID}<□{MailChimp Tag name}<□{MailChimp tag sub entry}

   MailChimp supported type are:

   - Date The format is (DD/MM/YYYY) or (MM/DD/YYYY)
   - Birthday The format is (DD/MM) or (MM/DD)
   - Website or Text They are text
   - Number The number will be parse into a Integer
   - Phone The North American format is not supported, use international
   - Address

   Street, City, Country and Zip are mandatory properties, otherwise the address property will be skipped.

```
address<=>ADDRESS<=>addr1,
city<=>ADDRESS<=>city,
zipCode<=>ADDRESS<=>zip,
countryName<=>ADDRESS<=>country
```

5. Deploy into Apache Unomi using the following commands from the Apache Karaf shell:

```
feature:repo-add mvn:org.apache.unomi/unomi-mailchimp-connector-karaf-kar/${project.version}/xml/features
feature:install unomi-mailchimp-connector-karaf-kar
```

12. DEVELOPERS

12.1. BUILDING

12.1.1. INITIAL SETUP

1. Install J2SE 8.0 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 3.0.3 (or later), which can be downloaded from http://maven.apache.org/download.html. Make sure that your PATH includes the MVN_HOME/bin directory.

12.1.2. BUILDING

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,\!performance-tests clean install
   ```

   This will compile Apache Unomi without running the tests and takes less time to build.

4. The distributions will be available under "package/target" directory.

12.1.3. INSTALLING AN ELASTICSEARCH SERVER

Starting with version 1.2, Apache Unomi no longer embeds an ElasticSearch server as this is no longer supported by the developers of ElasticSearch. Therefore you will need to install a standalone ElasticSearch using the following steps:

Download an ElasticSearch version. Here's the version you will need depending on your version of

Apache Unomi.

Apache Unomi <= 1.2 : https://www.elastic.co/downloads/past-releases/elasticsearch-5-1-2 Apache Unomi >= 1.3 : https://www.elastic.co/downloads/past-releases/elasticsearch-5-6-3

Uncompress the downloaded package into a directory

In the config/elasticsearch.yml file, uncomment and modify the following line :

```
cluster.name: contextElasticSearch
```

Launch the server using

```
bin/elasticsearch (Mac, Linux)
bin\elasticsearch.bat (Windows)
```

Check that the ElasticSearch is up and running by accessing the following URL :

http://localhost:9200

12.1.4. 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.

You can then start the server simply by using the command on UNIX/Linux/MacOS X :

```
./bin/karaf
```

or on Windows shell :

```
bin\karaf.bat
```

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
```

12.1.5. 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 3.x, http://karaf.apache.org

Before deploying, make sure that you have Apache Karaf properly installed. You will also have to increase the default maximum memory size and perm gen 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 JAVA_MAX_MEM=3G
export JAVA_MAX_PERM_MEM=384M
```

Install the WAR support, CXF and Karaf Cellar into Karaf by doing the following in the Karaf command line:

```
feature:repo-add cxf 3.0.2
feature:repo-add cellar 3.0.3
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 with karaf / karaf and see basic server information. If not something went wrong during the install.

12.1.6. 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 (7):
    1.7.0_51, x86_64:   "Java SE 7"   /Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home
    1.7.0_45, x86_64:   "Java SE 7"   /Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/Home
    1.7.0_25, x86_64:   "Java SE 7"   /Library/Java/JavaVirtualMachines/jdk1.7.0_25.jdk/Contents/Home
    1.6.0_65-b14-462, x86_64:   "Java SE 6"   /Library/Java/JavaVirtualMachines/1.6.0_65-b14-
462.jdk/Contents/Home
    1.6.0_65-b14-462, i386: "Java SE 6" /Library/Java/JavaVirtualMachines/1.6.0_65-b14-
462.jdk/Contents/Home
    1.6.0_65-b14-462, x86_64:   "Java SE 6"
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
    1.6.0_65-b14-462, i386: "Java SE 6"
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
```

You can then select the one you want using :

```
export JAVA_HOME=`/usr/libexec/java_home -v 1.7.0_51`
```

and then check that it was correctly referenced using:

```
java -version
```

which should give you a result such as this:

```
java version "1.7.0_51"
Java(TM) SE Runtime Environment (build 1.7.0_51-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)
```

12.1.7. 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 7 or more recent !

To run the tests simply activate the following profile :

```
mvn -P integration-tests clean install
```

12.1.8. RUNNING THE PERFORMANCE TESTS

Performance tests are based on Gatling. You need to have a running context server or cluster of servers before executing the tests.

Test parameteres are editable in the performance-tests/src/test/scala/unomi/Parameters.scala file. baseUrls should contains the URLs of all your cluster nodes

Run the test by using the gatling.conf file in performance-tests/src/test/resources :

```
export GATLING_CONF=<path>/performance-tests/src/test/resources
gatling.sh
```

Reports are generated in performance-tests/target/results.

12.1.9. TESTING WITH AN EXAMPLE PAGE

A default test page is provided at the following URL:

```
http://localhost:8181/index.html
```

This test page will trigger the loading of the /context.js script, which will try to retrieving the user context or create a new one if it doesn't exist yet. It also contains an experimental integration with Facebook Login, but it doesn't yet save the context back to the context server.

12.2. 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.

12.2.1. USING THE SHELL

You can connect to the Apache Karaf SSH Shell using the following command:

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

The default username/password is karaf/karaf. You should change this as soon as possible by editing the etc/users.properties file.

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 date in ES to be compliant with current version

SYNTAX
        unomi:migrate [fromVersionWithoutSuffix]

ARGUMENTS
        fromVersionWithoutSuffix
                Origin version without suffix/qualifier (e.g: 1.2.0)
                (defaults to 1.2.0)
```

12.2.2. LIFECYCLE COMMANDS

The commands control the lifecycle of the Apache Unomi server and are used to migrate, start or stop the server.

Table 3. Table Lifecycle commands

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 ElasticSearch using the argument fromVersion as a starting point.
stop	n/a	Shutsdown the Apache Unomi application
start	n/a	Starts the Apache Unomi application. Note that this state will be remembered between Apache Karaf launches, so in general it is only needed after a first installation or after a migrate command
version	n/a	Prints out the currently deployed version of the Apache Unomi application inside the Apache Karaf runtime.

12.2.3. RUNTIME COMMANDS

These commands are available once the application is running. If an argument is between brackets [] it means it is optional.

Table 4. Table Runtime commands

Command	Arguments	Description
rule-list	[maxEntries] [--csv]	Lists all the rules registered in the Apache Unomi server. The maxEntries (defaults to 100) will allow you to specify how many entries need to be retrieved. If the value is inferior to the total value, a message will display the total value of rules registered in the server. If you add the "--csv" option the list will be output as a CSV formatted table
rule-view	rule-id	Dumps a single rule in JSON. The rule-id argument can be retrieved from the rule-list command output.
rule-remove	rule-id	Removes a single rule from Apache Unomi. The rule-id argument can be retrieved from the rule-list command output. Warning: no confirmation is asked, be careful with this command.
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
event-view	event-id	Dumps a single event in JSON. The event-id can be retrieved from the event-tail command output.
event-list	[max-entries] [--csv]	List the last events processed by Apache Unomi. The max-entries parameter can be used to control how many events are displayed (default is 100). The --csv argument is used to output the list as a CSV list instead of an ASCII table.
event-search	profile-id [event-type] [max-entries]	This command makes it possible to search for the last events by profile-id and by event-type. A max-entries parameter (with a default value of 100) is also accepted to control the number of results returned by the search.

Command	Arguments	Description
action-list	[--csv]	Lists all the rule actions registered in the Apache Unomi server. This command is useful when developing plugins to check that everything is properly registered. If you add the "--csv" option the list will be output as a CSV formatted table
action-view	action-id	Dumps a single action in JSON. The action-id argument can be retrieved from the action-list command output.
condition-list	[csv]	List all the conditions registered in the server. If you add the "--csv" option the list will be output as a CSV formatted table
condition-view	condition-id	Dumps a single condition in JSON. The condition-id can be retrieved from the condition-list command output.
profile-list	[--csv]	List the last 10 modified profiles. If you add the "--csv" option the list will be output as a CSV formatted table
profile-view	profile-id	Dumps a single profile in JSON. The profile-id argument can be retrieved from the profile-list command output.
profile-remove	profile-id	Removes a profile identified by profile-id argument. Warning: no confirmation is asked so be careful with this command!
segment-list	[--csv]	Lists all the segments registered in the Apache Unomi server. If you add the "--csv" option the list will be output as a CSV formatted table
segment-view	segment-id	Dumps a single segment in JSON. The segment-id argument can be retrieved from the segment-list command output.

Command	Arguments	Description
segment-remove	segment-id	Removes a single segment identified by the segment-id argument. Warning: no confirmation is asked so be careful with this command!
session-list	[--csv]	Lists the last 10 sessions by last event date. If you add the "--csv" option the list will be output as a CSV formatted table
session-view	session-id	Dumps a single session in JSON. The session-id argument can be retrieved from the session-list, profile-list or event-tail command output.
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.

Command	Arguments	Description
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.

Unomi is architected so that users can provided extensions in the form of plugins.

12.3. TYPES VS. INSTANCES

Several extension points in Unomi rely on the concept of type: the extension defines a prototype for what the actual items will be once parameterized with values known only at runtime. This is similar to the concept of classes in object-oriented programming: types define classes, providing the expected structure and which fields are expected to be provided at runtime, that are then instantiated when needed with actual values.

12.4. PLUGIN STRUCTURE

Being built on top of Apache Karaf, Unomi leverages OSGi to support plugins. A Unomi plugin is, thus, an OSGi bundle specifying some specific metadata to tell Unomi the kind of entities it provides. A plugin can provide the following entities to extend Unomi, each with its associated definition (as a JSON file), located in a specific spot within the META-INF/cxs/ directory of the bundle JAR file:

Entity	Location in cxs directory
ActionType	actions
ConditionType	conditions
Persona	personas
PropertyMergeStrategyType	mergers
PropertyType	properties then profiles or sessions subdirectory then <category name> directory
Rule	rules

Entity	Location in cxs directory
Scoring	scorings
Segment	segments
ValueType	values

Blueprint is used to declare what the plugin provides and inject any required dependency. The Blueprint file is located, as usual, at OSGI-INF/blueprint/blueprint.xml in the bundle JAR file.

The plugin otherwise follows a regular maven project layout and should depend on the Unomi API maven artifact:

```
<dependency>
   <groupId>org.apache.unomi</groupId>
   <artifactId>unomi-api</artifactId>
   <version>...</version>
</dependency>
```

Some plugins consists only of JSON definitions that are used to instantiate the appropriate structures at runtime while some more involved plugins provide code that extends Unomi in deeper ways.

In both cases, plugins can provide more that one type of extension. For example, a plugin could provide both `ActionType`s and `ConditionType`s.

12.5. EXTENSION POINTS

12.5.1. ACTIONTYPE

ActionType`s define new actions that can be used as consequences of Rules being triggered. When a rule triggers, it creates new actions based on the event data and the rule internal processes, providing values for parameters defined in the associated `ActionType. Example actions include: “Set user property x to value y” or “Send a message to service x”.

12.5.2. CONDITIONTYPE

`ConditionType`s define new conditions that can be applied to items (for example to decide whether a rule needs to be triggered or if a profile is considered as taking part in a campaign) or to perform queries against the stored Unomi data. They may be implemented in Java when attempting to define a particularly complex test or one that can better be optimized by coding it. They may also be defined as combination of other conditions. A simple condition could be: “User is male”, while a more generic condition with parameters may test whether a given property has a specific value: “User property x has value y”.

12.5.3. PERSONA

A persona is a "virtual" profile used to represent categories of profiles, and may also be used to test how

a personalized experience would look like using this virtual profile. A persona can define predefined properties and sessions. Persona definition make it possible to “emulate” a certain type of profile, e.g : US visitor, non-US visitor, etc.

12.5.4. PROPERTYMERGESTRATEGYTYPE

A strategy to resolve how to merge properties when merging profile together.

12.5.5. PROPERTYTYPE

Definition for a profile or session property, specifying how possible values are constrained, if the value is multi-valued (a vector of values as opposed to a scalar value). `PropertyType`s can also be categorized using systemTags or file system structure, using sub-directories to organize definition files.

12.5.6. RULE

`Rule`s are conditional sets of actions to be executed in response to incoming events. Triggering of rules is guarded by a condition: the rule is only triggered if the associated condition is satisfied. That condition can test the event itself, but also the profile or the session. Once a rule triggers, a list of actions can be performed as consequences. Also, when rules trigger, a specific event is raised so that other parts of Unomi can react accordingly.

12.5.7. SCORING

`Scoring`s are set of conditions associated with a value to assign to profiles when matching so that the associated users can be scored along that dimension. Each scoring element is evaluated and matching profiles' scores are incremented with the associated value.

12.5.8. SEGMENTS

`Segment`s represent dynamically evaluated groups of similar profiles in order to categorize the associated users. To be considered part of a given segment, users must satisfies the segment’s condition. If they match, users are automatically added to the segment. Similarly, if at any given point during, they cease to satisfy the segment’s condition, they are automatically removed from it.

12.5.9. TAG

`Tag`s are simple labels that are used to classify all other objects inside Unomi.

12.5.10. VALUETYPE

Definition for values that can be assigned to properties ("primitive" types).

12.6. OTHER UNOMI ENTITIES

12.6.1. USERLIST

User list are simple static lists of users. The associated profile stores the lists it belongs to in a specific property.

12.6.2. GOAL

Goals represent tracked activities / actions that can be accomplished by site (or more precisely scope) visitors. These are tracked in general because they relate to specific business objectives or are relevant to measure site/scope performance.

Goals can be defined at the scope level or in the context of a particular Campaign. Either types of goals behave exactly the same way with the exception of two notable differences: - duration: scope-level goals are considered until removed while campaign-level goals are only considered for the campaign duration - audience filtering: any visitor is considered for scope-level goals while campaign-level goals only consider visitors who match the campaign's conditions

12.6.3. CAMPAIGN

A goal-oriented, time-limited marketing operation that needs to be evaluated for return on investment performance by tracking the ratio of visits to conversions.

12.7. CUSTOM EXTENSIONS

Apache Unomi is a pluggeable server that may be extended in many ways. This document assumes you are familiar with the Apache Unomi concepts . This document is mostly a reference document on the different things that may be used inside an extension. If you are looking for complete samples, please see the samples page.

12.7.1. CREATING AN EXTENSION

An extension is simply a Maven project, with a Maven pom that looks like this:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <parent>
    <groupId>org.apache.unomi</groupId>
    <artifactId>unomi-extensions</artifactId>
    <version>${project.version}</version>
  </parent>

  <modelVersion>4.0.0</modelVersion>

  <artifactId>unomi-extension-example</artifactId>
  <name>Apache Unomi :: Extensions :: Example</name>
  <description>Service implementation for the Apache Unomi Context Server extension that
integrates with the Geonames database</description>
  <version>${project.version}</version>
  <packaging>bundle</packaging>

  <dependencies>
    <!-- This dependency is not required but generally used in extensions -->
    <dependency>
      <groupId>org.apache.unomi</groupId>
      <artifactId>unomi-api</artifactId>
      <version>${project.version}</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Embed-Dependency>*;scope=compile|runtime</Embed-Dependency>
            <Import-Package>
              sun.misc;resolution:=optional,
              *
            </Import-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

An extension may contain many different kinds of Apache Unomi objects, as well as custom OSGi services or anything that is needed to build your application.

12.7.2. DEPLOYMENT AND CUSTOM DEFINITION

When you deploy a custom bundle with a custom definition (see "Predefined xxx" chapters under) for the first time, the definition will automatically be deployed at your bundle start event **if it does not exist**. After that if you redeploy the same bundle, the definition will not be redeployed, but you can redeploy it manually using the command unomi:deploy-definition <bundleId> <fileName> If you need to modify an existing definition when deploying the module, see Migration patches.

12.7.3. PREDEFINED SEGMENTS

You may provide pre-defined segments by simply adding a JSON file in the src/main/resources/META-INF/cxs/segments directory of your Maven project. Here is an example of a pre-defined segment:

```
{
 "metadata": {
  "id": "leads",
  "name": "Leads",
  "scope": "systemscope",
  "description": "You can customize the list below by editing the leads segment.",
  "readOnly":true
 },
 "condition": {
  "parameterValues": {
   "subConditions": [
    {
     "parameterValues": {
      "propertyName": "properties.leadAssignedTo",
      "comparisonOperator": "exists"
     },
     "type": "profilePropertyCondition"
    }
   ],
   "operator" : "and"
  },
  "type": "booleanCondition"
 }
}
```

Basically this segment uses a condition to test if the profile has a property leadAssignedTo that exists. All profiles that match this condition will be part of the pre-defined segment.

12.7.4. PREDEFINED RULES

You may provide pre-defined rules by simply adding a JSON file in the src/main/resources/META-INF/cxs/rules directory of your Maven project. Here is an example of a pre-defined rule:

```
{
  "metadata" : {
    "id": "evaluateProfileSegments",
    "name": "Evaluate segments",
    "description" : "Evaluate segments when a profile is modified",
    "readOnly":true
  },

  "condition" : {
    "type": "profileUpdatedEventCondition",
    "parameterValues": {
    }
  },

  "actions" : [
    {
      "type": "evaluateProfileSegmentsAction",
      "parameterValues": {
      }
    }
  ]

}
```

In this example we provide a rule that will execute when a predefined composed condition of type "profileUpdatedEventCondition" is received. See below to see how predefined composed conditions are declared. Once the condition is matched, the actions will be executed in sequence. In this example there is only a single action of type "evaluateProfileSegmentsAction" that is defined so it will be executed by Apache Unomi's rule engine. You can also see below how custom actions may be defined.

12.7.5. PREDEFINED PROPERTIES

By default Apache Unomi comes with a set of pre-defined properties, but in many cases it is useful to add additional predefined property definitions. You can create property definitions for session or profile properties by creating them in different directories.

For session properties you must create a JSON file in the following directory in your Maven project:

```
src/main/resources/META-INF/cxs/properties/sessions
```

For profile properties you must create the JSON file inside the directory in your Maven project:

```
src/main/resources/META-INF/cxs/properties/profiles
```

Here is an example of a property definition JSON file

```
{
   "metadata": {
      "id": "city",
      "name": "City",
      "systemTags": ["properties", "profileProperties", "contactProfileProperties"]
   },
   "type": "string",
   "defaultValue": "",
   "automaticMappingsFrom": [ ],
   "rank": "304.0"
}
```

12.7.6. PREDEFINED CHILD CONDITIONS

You can define new predefined conditions that are actually conditions inheriting from a parent condition and setting pre-defined parameter values. You can do this by creating a JSON file in:

```
src/main/resources/META-INF/cxs/conditions
```

Here is an example of a JSON file that defines a profileUpdateEventCondition that inherits from a parent condition of type eventTypeCondition.

```
{
 "metadata": {
  "id": "profileUpdatedEventCondition",
  "name": "profileUpdatedEventCondition",
  "description": "",
  "systemTags": [
   "event",
   "eventCondition"
  ],
  "readOnly": true
 },
 "parentCondition": {
  "type": "eventTypeCondition",
  "parameterValues": {
   "eventTypeId": "profileUpdated"
  }
 },

 "parameters": [
 ]
}
```

12.7.7. PREDEFINED PERSONAS

Personas may also be pre-defined by creating JSON files in the following directory:

```
src/main/resources/META-INF/cxs/personas
```

Here is an example of a persona definition JSON file:

```
{
  "persona": {
    "itemId": "usVisitor",
    "properties": {
      "description": "Represents a visitor browsing from inside the continental US",
      "firstName": "U.S.",
      "lastName": "Visitor"
    },
    "segments": []
  },
  "sessions": [
    {
      "itemId": "aa3b04bd-8f4d-4a07-8e96-d33ffa04d3d9",
      "profileId": "usVisitor",
      "properties": {
        "operatingSystemName": "OS X 10.9 Mavericks",
        "sessionCountryName": "United States",
        "location": {
          "lat":37.422,
          "lon":-122.084058
        },
        "userAgentVersion": "37.0.2062.120",
        "sessionCountryCode": "US",
        "deviceCategory": "Personal computer",
        "operatingSystemFamily": "OS X",
        "userAgentName": "Chrome",
        "sessionCity": "Mountain View"
      },
      "timeStamp": "2014-09-18T11:40:54Z",
      "lastEventDate": "2014-09-18T11:40:59Z",
      "duration": 4790
    }
  ]
}
```

You can see that it's also possible to define sessions for personas.

12.7.8. CUSTOM ACTIONS

Custom actions are a powerful way to integrate with external systems by being able to define custom logic that will be executed by an Apache Unomi rule. An action is defined by a JSON file created in the following directory:

```
src/main/resources/META-INF/cxs/actions
```

Here is an example of a JSON action definition:

```
{
 "metadata": {
  "id": "addToListsAction",
  "name": "addToListsAction",
  "description": "",
  "systemTags": [
   "demographic",
   "availableToEndUser"
  ],
  "readOnly": true
 },
 "actionExecutor": "addToLists",
 "parameters": [
  {
   "id": "listIdentifiers",
   "type": "string",
   "multivalued": true
  }
 ]
}
```

The actionExecutor identifier refers to a service property that is defined in the OSGi Blueprint service registration. Note that any OSGi service registration may be used, but in these examples we use OSGi Blueprint. The definition for the above JSON file will be found in a file called src/main/resources/OSGI-INF/blueprint/blueprint.xml with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<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"/>
   <reference id="eventService" interface="org.apache.unomi.api.services.EventService"/>

   <!-- Action executors -->

   <service interface="org.apache.unomi.api.actions.ActionExecutor">
      <service-properties>
         <entry key="actionExecutorId" value="addToLists"/>
      </service-properties>
      <bean class="org.apache.unomi.lists.actions.AddToListsAction">
         <property name="profileService" ref="profileService"/>
         <property name="eventService" ref="eventService"/>
      </bean>
   </service>

</blueprint>
```

You can note here the actionExecutorId that corresponds to the actionExecutor in the JSON file.

The implementation of the action is available here : org.apache.unomi.lists.actions.AddToListsAction

12.7.9. CUSTOM CONDITIONS

Custom conditions are different from predefined child conditions because they implement their logic using Java classes. They are also declared by adding a JSON file into the conditions directory:

```
src/main/resources/META-INF/cxs/conditions
```

Here is an example of JSON custom condition definition:

```
{
  "metadata": {
    "id": "matchAllCondition",
    "name": "matchAllCondition",
    "description": "",
    "systemTags": [
      "logical",
      "profileCondition",
      "eventCondition",
      "sessionCondition",
      "sourceEventCondition"
    ],
    "readOnly": true
  },
  "conditionEvaluator": "matchAllConditionEvaluator",
  "queryBuilder": "matchAllConditionESQueryBuilder",

  "parameters": [
  ]
}
```

Note the conditionEvaluator and the queryBuilder values. These reference OSGi service properties that are declared in an OSGi Blueprint configuration file (other service definitions may also be used such as Declarative Services or even Java registered services). Here is an example of an OSGi Blueprint definition corresponding to the above JSON condition definition file.

```
src/main/resources/OSGI-INF/blueprint/blueprint.xml

<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">

   <service

interface="org.apache.unomi.persistence.elasticsearch.conditions.ConditionESQueryBuilder">
      <service-properties>
         <entry key="queryBuilderId" value="matchAllConditionESQueryBuilder"/>
      </service-properties>
      <bean
class="org.apache.unomi.plugins.baseplugin.conditions.MatchAllConditionESQueryBuilder"/>
   </service>

   <service interface="org.apache.unomi.persistence.elasticsearch.conditions.ConditionEvaluator">
      <service-properties>
         <entry key="conditionEvaluatorId" value="matchAllConditionEvaluator"/>
      </service-properties>
      <bean class="org.apache.unomi.plugins.baseplugin.conditions.MatchAllConditionEvaluator"/>
   </service>

</blueprint>
```

You can find the implementation of the two classes here :

- org.apache.unomi.plugins.baseplugin.conditions.MatchAllConditionESQueryBuilder
- org.apache.unomi.plugins.baseplugin.conditions.MatchAllConditionEvaluator

12.8. 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 .