APACHE UNOMI 1.1.X - DOCUMENTATION

Apache Software Foundation

TABLE OF CONTENTS

1. CONCEPTS

1.1. 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.

1.1.1. 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",
    "tags": ["social"]
  },
  "target": "profiles",
  "type": "integer"
}
```

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

1.1.2. 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": {}
   }
  }
}
```


1.1.3. 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.

1.1.4. 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.

1.2. EXTENDING UNOMI VIA PLUGINS

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

1.2.1. 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.

1.2.2. 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
Scoring	scorings
Segment	segments
Tag	tags
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.

1.2.3. EXTENSION POINTS

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”.

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”.

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.

PROPERTYMERGESTRATEGYTYPE

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

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 tags or file system structure, using sub-directories to organize definition files.

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.

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.

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.

TAG

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

VALUETYPE

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

1.2.4. OTHER UNOMI ENTITIES

USERLIST

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

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

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.

2. QUICK START

2.1. BUILDING

Simply type at the root of the project:

```
mvn clean install -P generate-package
```

The Maven build process will generate both a standalone package you can use directly to start the context server (see "Deploying the generated package") or a KAR file that you can then deploy using a manual deployment process into an already installed Apache Karaf server (see "Deploying into an existing Karaf server")

If you want to build and run the integration tests, you should instead use :

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

2.1.1. DEPLOYING THE GENERATED 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
```

2.1.2. 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 3.0.2+, http://karaf.apache.org - Local copy of the Elasticsearch ZIP package, available here : http://www.elasticsearch.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 KARAF_OPTS="-Djava.library.path=$MY_KARAF_HOME/lib/sigar"
export JAVA_MAX_MEM=3G
export JAVA_MAX_PERM_MEM=384M
```

.

You will also need to have the Hyperic Sigar native libraries in your Karaf installation, so in order to this go to the Elasticsearch website (http://www.elasticsearch.org) and download the ZIP package. Decompress it somewhere on your disk and copy all the files from the lib/sigar directory into Karaf's lib/sigar directory (must be created first) EXCEPT THE SIGAR.JAR file.

.

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

```
feature:install -v war
feature:repo-add cxf 2.7.11
feature:install -v cxf/2.7.11
```

.

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
```

.

Copy the following KAR to the Karaf deploy directory, as in this example line:

```
cp kar/target/unomi-kar-1.0.0-SNAPSHOT.kar ~/java/deployments/unomi/apache-karaf-3.0.1/deploy/
```

.

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.

2.1.3. 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)
```

2.1.4. 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
```

2.1.5. 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.

2.1.6. 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.

2.1.7. INTEGRATING ONTO A PAGE

Simply reference the context script in your HTML as in the following example:

```
<script type="text/javascript">
   (function(){ var u=(("https:" == document.location.protocol) ? "https://localhost:8181/" :
"http://localhost:8181/");
   var d=document, g=d.createElement('script'), s=d.getElementsByTagName('script')[0];
g.type='text/javascript'; g.defer=true; g.async=true; g.src=u+'context.js';
   s.parentNode.insertBefore(g,s); })();
</script>
```

2.2. 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.

2.2.1. 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. Additionnally, 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.

2.2.2. RUNNING UNOMI

BUILDING UNOMI

1. Get the code: git clone https://git-wip-us.apache.org/repos/asf/incubator-unomi.git
2. Build and install according to the instructions and install Unomi.

START UNOMI

Start Unomi according to the 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).

BUILDING THE TWEET BUTTON SAMPLE

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 SAMPLE

In standard Karaf fashion, you will need to copy the sample 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-1.0.0-incubating-SNAPSHOT.jar ../../package/target/unomi-1.0.0-incubating-SNAPSHOT/deploy
```

TESTING THE SAMPLE

You can now go to http://localhost:8181/index.html to test the sample 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.

2.2.3. 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.

2.2.4. 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.

2.3. EXAMPLE

2.3.1. 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 want to react to Twitter events by associating information to their profile. We will record the number of times the user tweeted (as a tweetNb profile integer property) as well as the URLs they tweeted from (as a tweetedFrom multi-valued string profile property). We will accomplish this using a simple HTML page on which we position a standard "Tweet" button. 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.

2.3.2. HTML PAGE

The code for the HTML page with our Tweet button can be found at https://github.com/apache/incubator-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.

2.3.3. 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, segments or scores 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-sample), 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-sample';
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",
  "tags": [
    "event"
  ],
  "parameters": []
}
```

We specify the identifier for the action type, a list of tags 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" auto-export="interfaces">
      <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;
}
```

2.4. 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.

2.5. ANNEX

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

2.6. CONFIGURATION

2.6.1. CHANGING THE DEFAULT CONFIGURATION

If you want to change the default configuration, you can perform any modification you want in the $MY_KARAF_HOME/etc directory.

The context server configuration is kept in the $MY_KARAF_HOME/etc/org.apache.unomi.web.cfg . It defines the addresses and port where it can be found :

```
contextserver.address=localhost
contextserver.port=8181
contextserver.secureAddress=localhost
contextserver.securePort=9443
contextserver.domain=apache.org
```

If you need to specify an Elasticsearch cluster name that is 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.

To change the cluster name, first create a file called

```
$MY_KARAF_HOME/etc/org.apache.unomi.persistence.elasticsearch.cfg
```

with the following contents:

```
cluster.name=contextElasticSearch
index.name=context
elasticSearchConfig=file:${karaf.etc}/elasticsearch.yml
```

And replace the cluster.name parameter here by your cluster name.

You can also put an elasticsearch configuration file in $MY_KARAF_HOME/etc/elasticsearch.yml , and put any standard Elasticsearch configuration options in this last file.

If you want your context server to be a client only on a cluster of elasticsearch nodes, just set the node.data property to false.

2.6.2. INSTALLING THE MAXMIND GEOIPLITE2 IP LOOKUP DATABASE

The Context Server 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.

2.6.3. INSTALLING GEONAMES DATABASE

Context server 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/org.apache.unomi.geonames.cfg and set request.geonamesDatabase.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.

2.6.4. REST API SECURITY

The 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/custom.properties file :

```
org.osgi.service.http.secure.enabled = true
org.ops4j.pax.web.ssl.keystore=${karaf.etc}/keystore
org.ops4j.pax.web.ssl.password=changeme
org.ops4j.pax.web.ssl.keypassword=changeme
org.osgi.service.http.port.secure=9443
```

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

2.6.5. AUTOMATIC PROFILE MERGING

The context server 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.

2.6.6. 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 Context Server 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 context server 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. To do so, simply modify the following file :

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

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

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

Step 2 : Adjust the Context Server IP filtering

By default the Context Server limits to connections to port 9200 and 9300 to the following IP ranges

```
- localhost
- 127.0.0.1
- ::1
- the current subnet (i.e., 192.168.1.0-192.168.1.255)
```

(this is done using a custom plugin for Elasticsearch, that you may find here : https://git-wip-us.apache.org/repos/asf/incubator-unomi/context-server/persistence-elasticsearch/plugins/security)

You can adjust this setting by using the following setting in the $MY_KARAF_HOME/etc/elasticsearch.yml file :

```
security.ipranges: localhost,127.0.0.1,::1,10.0.1.0-10.0.1.255
```

Step 3 : 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.

2.6.7. 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 /etc/org.apache.unomi.web.cfg for unomi.apache.org

contextserver.address=unomi.apache.org contextserver.port=80
contextserver.secureAddress=unomi.apache.org contextserver.securePort=443
contextserver.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. CLUSTER SETUP

3.1. CLUSTER SETUP

Context server relies on Elasticsearch to discover and configure its cluster. You just need to install multiple context servers on the same network, and enable the discovery protocol in $MY_KARAF_HOME/etc/org.apache.unomi.persistence.elasticsearch.cfg file :

```
discovery.zen.ping.multicast.enabled=true
```

All nodes on the same network, sharing the same cluster name will be part of the same cluster.

3.1.1. RECOMMENDED CONFIGURATIONS

It is recommended to have one node dedicated to the context server, where the other nodes take care of the Elasticsearch persistence. The node dedicated to the context server will have node.data set to false.

2 NODES CONFIGURATION

One node dedicated to context server, 1 node for elasticsearch storage.

Node A :

```
node.data=true
numberOfReplicas=0
monthlyIndex.numberOfReplicas=0
```

Node B :

```
node.data=false
numberOfReplicas=0
monthlyIndex.numberOfReplicas=0
```

3 NODES CONFIGURATION

One node dedicated to context server, 2 nodes for elasticsearch storage with fault-tolerance

Node A :

```
node.data=false
numberOfReplicas=1
monthlyIndex.numberOfReplicas=1
```

Node B :

```
node.data=true
numberOfReplicas=1
monthlyIndex.numberOfReplicas=1
```

Node C :

```
node.data=true
numberOfReplicas=1
monthlyIndex.numberOfReplicas=1
```

3.1.2. SPECIFIC CONFIGURATION

If multicast is not allowed on your network, you'll need to switch to unicast protocol and manually configure the server IPs. This can be done by disabling the elasticsearch automatic discovery in $MY_KARAF_HOME/etc/org.apache.unomi.persistence.elasticsearch.cfg :

```
discovery.zen.ping.multicast.enabled=false
```

And then set the property discovery.zen.ping.unicast.hosts in $MY_KARAF_HOME/etc/elasticsearch.yml files :

```
discovery.zen.ping.unicast.hosts: ['192.168.0.1:9300', '192.168.0.2:9300']
```

More information and configuration options can be found at : https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-discovery.html