Mayam Tasks v3

Programming Guide

Table of Contents

Document Information

Revision History

Revision	Date	Author	Comment
1.0	Sep. 2018	André Cruz	
2.0	Mar. 2023	Ganesh Srinivasson	

Contact Information

Mayam AB,
Vikingavägen 2,
182 63 Djursholm
Sweden
Phone: +46 (0)730-808012
E-mail: info@mayam.com[1]

[1] mailto:info@mayam.com

1. Abstract

The Mayam task management system, Mayam Tasks, extends a workflow and/or MAM environment with advanced task management capabilities and APIs specifically designed for use with BPM systems.

1.1. What to expect from this document

This document introduces the product by providing an overview of the tasklist GUI, the BPM interface and other support applications such as the reporing engine. A description of the system architecture is also provided.

1.2. What You Should Know Before Reading This Manual

This manual describes how to manage the Mayam Tasks environment and the associated MAM and BPM systems. Hence, relatively deep understanding of the principal functionality of these systems is required to make adequate use of the functions provided through the `SDK` and `API` covered in this manual. For Mayam Tasks, please refer to the document Mayam Tasks Technical Reference (found at: http://your-hostname-here:8084/tasksdoc/tasks_technical_reference.pdf) for information regarding functionality, system architecture, data model and system configuration (however, the parts regarding installation and maintenance can be skipped). Refer to MAM system product documentation to familiarize yourself with the MAM system concepts, data model, security model. A moderate understanding of the Java language and an appropriate IDE (such as Eclipse) is required for the `SDK` portion of the tutorial. A good understanding of `HTTP` and `JSON` is required for the `REST` portion.

2. Introduction

The Mayam Tasks application extends the Media Asset Management (MAM) environment with advanced task management, business process and integration capabilities. Through the Mayam approach to workflow implementation, workflows in a MAM environment are provided through the use of a dedicated task and process modelling and execution subsystem, relying on Mayam Tasks for GUI functions and MAM-independent APIs.

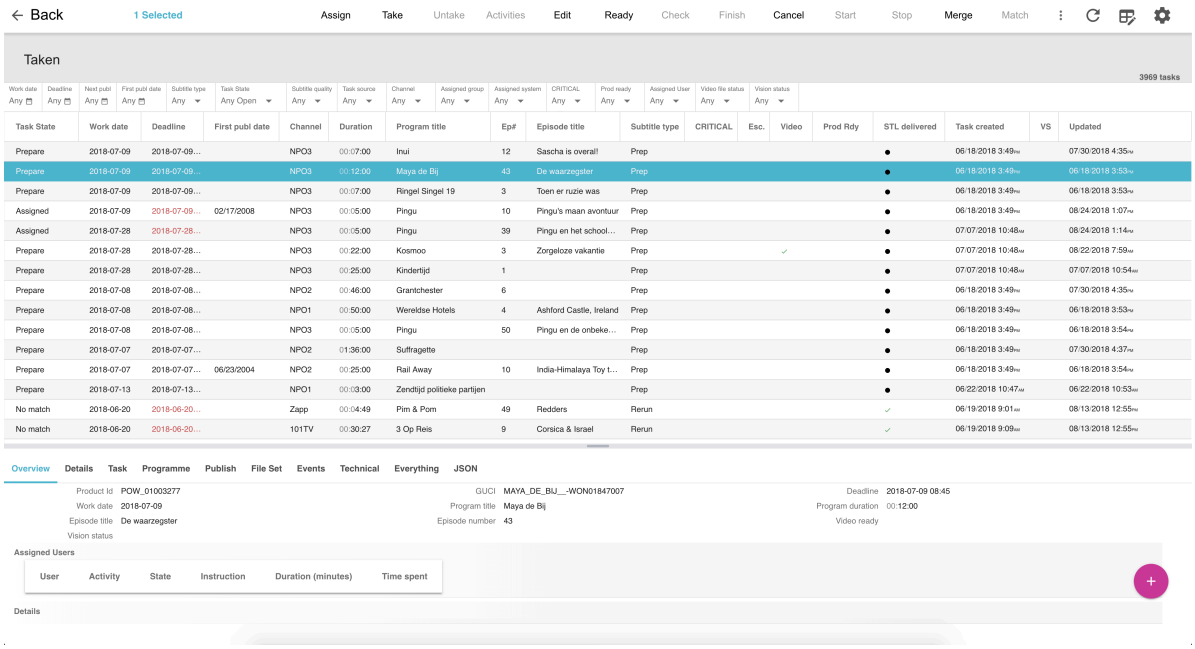

Figure 2.1. Mayam Tasks GUI.

This manual aims to provide the reader with an understanding of the parts of the Mayam Tasks system that is exposed to third party developers. Basic concepts and terminology will be described, and then followed by coding tutorials. However, the full SDK and API reference documentation is not listed in this manual. References to online documentation are listed in Reference Documentation.

2.1. Interfaces

There are two interfaces covered by this manual. The recommended choice is our Java SDK, which consists of client classes and domain classes and provides type safe and convenient programmable access to our system. The second approach is to directly access the `REST API` that the SDK sits on top of.

3. Programming Data Model

The primary purpose with the Mayam Tasks SDK (and API) is to provide unified high-level access to workflow data including MAM assets. While the SDK can be used to manage workflow tasks and MAM assets with a minimum of operations, understanding of the high-level programming data level is needed to utilize advanced functions. There are three primary high-level entities:

- Task
- Asset
- BPM process instance

These entities are described below. In addition, other entities such as system configurations can also be handled. But for now, let's focus on the primary entities listed below.

3.1. Top Level Entities and Operations

Mayam Tasks chiefly manages Mayam tasks (referred to as Task entities), MAM assets, and BPM process instances and this is also reflected in our interfaces.

3.1.1. Task

The workflow tasks are understandably the main focus, given the name of our product. A task is a unit of work to be performed by human and/or machine. Unique numeric identifier and state are the two required attributes, but the full set is available for use by tasks. Simply put, the data shown when clicking on a row in the Mayam Tasks GUI is represented by the Task entity. In addition, a Task entity refers to zero or more Tasks and Assets via key ID attributes. The SDK provides Task management functions ranging from basic CRUD functions to advanced search and filter lookup and sub-field level data manipulation (like adding a comment to a comment log). A number of additional operations like notifications can also be performed. It is also possible to configure the system to send data change events to the SDK in case for example key data changes should be propagated to a third-party system.

3.1.2. Asset

The term Asset refers to an Asset managed by Media Asset Management (MAM) system. Using a type qualifier, most MAM entities such as Series, Programmes, Versions/Items, Packages and Folders can be represented using an Asset entity. In addition, files not managed by a MAM can also be described. To assure uniqueness, both asset type and asset identifier is required. A key property of the Asset and Task processing is that Assets can be connected to tasks, causing select data to be synchronized between the two. For example, if a Task title attribute is mapped to the corresponding MAM field, bi-directional data synchronization will occur. The effect of this mapping scheme is that the SDK user can focus on the high

level data management and leave underlying data synchronization between Tasks and Assets to the underlying system. Beyond Asset CRUD operations, the Mayam SDK also exposes

a number of MAM media functions such as media file transfers, transcoding, segment list management etc.

3.1.3. BPM Process Instance

Mayam Tasks interacts with a BPM platform as follows:

- Workflow tasks and assets can be managed from the BPM process
- BPM process instances can be managed (create, update/signal and delete) from the Mayam Tasks GUI and the SDK
- Task and Asset events can be routed to the BPM platform. For example, when a media file is imported into a MAM asset, Mayam Tasks forwards the **`has media`** event. Process instances use the asset ID as the correlation key and will receive this event without the need for any explicit coded signalling. From the point of view of Mayam Tasks, a BPM process instance is a (potentially) long running job, much like a scheduled file transfer. Through the SDK (and API), BPM process instances can be created, updated and terminated.

3.2. Static Constructs

A driving design principle of Mayam Tasks is to reduce run-time errors by strict data typing and a predefined set of variables (referred to as Attributes below). Another key principle is that differences between MAM systems should be kept to a minimum. Both these principles are supported by representing all top level entities using a fixed set of types attributes.

3.2.1. Attribute

In order to robustly pass both technical data and metadata from one system to another, while also allow for meaningful presentation within our web application, we have compiled a list of mappable attributes. Each attribute has a defined purpose (as annotated with `@Purpose`), a value class, and a set of constraints. The subjects of create, read, and update operations are in most cases represented as maps from attribute to value class associated with the respective attribute. While this is not easily enforced at compile time, the `AttributeMap` class comes with an `AttributeValidator` companion which does this checking at runtime well before any network traffic is involved..

3.2.2. Value Data Types

For many of our attributes, a set of acceptable values have been collected. In the Java SDK, these are listed in enums. Only a subset of each is expected to be used per installation, but the values themselves are documented and generic logic may apply - display rendering or otherwise.

Some attributes require complex data structures, and classes for these are also provided with the SDK. Annotated with `@Complex`, these value classes are limited in their use. Searches, for example, are rarely supported for complex classes.

3.2.3. Unmanaged Metadata

From version 2.4, there is a special Unmanaged Metadata facility. This facility provides access to the underlying MAM field metadata in a key/value map without the need for configured mappings to attributes. The whole collection of Unmanaged Metadata fields can also be stored in the attribute UNMANAGED_METADATA.

3.3. Configurable Entities

Building upon the foundation of the static constructs, Mayam Tasks is highly configurable. The dynamic parts are mainly concerned with the presentation of data within our web application, but a number of the configuration options also have an effect visible for third party developers.

3.3.1. Task List Configuration

Each customer installation comes with a factory default that is maintained by Mayam using a Java DSL. The Task Administration web application can then be used by customer and integrators to make adjustments. The full configuration is available through the SDK, with write support exposed through the `REST` interface; the latter should really only be used to transfer configuration between systems (such as staging to production).

3.3.2. Fields

As stated above, attributes have statically been assigned purpose, type, and constraints. Further constraints, along with rendering hints, can be configured using fields. One common example is the association of a **`CVL`** which is either entered directly into the configuration, or as a reference to a MAM `dictionary`. It should be noted that these constraints do not have any effect on operations performed using the SDK or via `REST` directly. More than one field may be associated with a particular attribute. This is typically done to allow mandatory access or use alternative labels in some situations but not others. External use of fields has historically been limited to label and type use within reporting plugins.

3.3.3. Attribute Mapping

Attributes can be mapped to metadata fields or, for some MAMs, directly to database columns. This causes changes made to assets to be propagated through message queues and daemons to any tasks associated with said asset. Changes to mapped attributes in tasks that are associated with assets will conversely cause those changes to be propagated to the MAM. For a mapping to be valid, constraints must align. Notably, data types need to be compatible.

3.4. Further Reading

While the configuration of task lists and of mappings between task and MAM data can seem straightforward at first, the complexity in real world implementation typically lies in the scope and extent of the configuration. It is not uncommon with 10+ task lists, 100+ fields and 50+ forms plus mappings to MAM data at 4 levels plus the shared use MAM dictionaries for configurable drop downs. To be fully effective working with these configurable entities, a thorough understanding is of the Tasks data model and the Field / Attribute constructs is required. Beyond reading the Mayam Tasks Technical Reference document, a good idea is to explore the Task Administration application for some hands-on experience working with Fields and Attributes.

4. Java SDK Tutorial

In this chapter, a tutorial on the Java SDK is provided. The material focuses on a hands-on experience using a typical development environment to code and run working programs performing typical task and asset management functions.

4.1. Development Environment Setup

The SDK and all of its runtime dependencies are provided in a single, easy to manage `jar` file. This can be downloaded from any server running our web services app. Typical URL is http://mayam:8084/tasks-ws/. This is also where reference documentation and examples can be found.

For users of Maven, the dependencies can also be made available as separate artifacts.

Recommended development environment is Eclipse. Configure the build path of your project to include the external jar downloaded. You are now ready to start coding.

4.1.1. My First Task

The SDK supports automatic dependency injection using **Guice**, a framework developed by Google. In our first examples, however, this feature will not be used. Factory methods for the most commonly needed classes are provided as part of the client.

```
package com.mayam.wf.ws.client.example.tutorial;
import java.net.URL;
import com.mayam.wf.attributes.shared.Attribute;
import com.mayam.wf.attributes.shared.AttributeMap;
import com.mayam.wf.attributes.shared.type.TaskState;
import com.mayam.wf.ws.client.TasksClient;

public class MyFirstTask {
  public static void main( String [] args ) throws Exception {
    final URL baseUrl = new URL( "http://localhost:8084/tasks-ws" );
    final String apiToken = "test:test";
    final TasksClient client = TasksClient. createTaskClient()

    setup( baseUrl, apiToken, "MyFirstTask", "1.0" );
    final AttributeMap task = client. createAttributeMap();
    final String suffix = Long.toString(
      System.currentTimeMillis() / 1000, 26 );
      task.setAttribute( Attribute.TASK_LIST_ID, "example" );
      task.setAttribute( Attribute.TASK_STATE, TaskState.OPEN );
      task.setAttribute( Attribute.ASSET_TITLE, "MyFirstTask " + suffix );
      final AttributeMap result = client.taskApi().createTask( task );
      final Long taskId = result.getAttribute( Attribute TASK_ID );
      System.out.println( "Task created with id " + taskId );
  }
}
```

For brevity, no error handling is performed. The above code is by no means meant to teach best practices. Following is a breakdown of the code, top to bottom.

There are two variables that may need changing for the examples to run. The **`baseUrl`** (line 10) should point to wherever the tasks-ws.war is deployed. Typically, you should only need to replace localhost with the mayam server name.

The **`apiToken`** (line 11) is a pair of account name and secret. Please note that "test" is not considered secret in a production environment. For the example to work, the token must be listed in the **`api.accounts`** of **`site-config.properties`** which can be found under `/mayam/jetty/default/resources/`. Mayam Tasks must be restarted for any changes to take effect.

The client itself is created (line 12) by means of the **`createTaskClient()`** factory method, after which it is configured and enabled using **`setup()`**. The last two arguments associate the client with the name and version of our application, allowing the information to be logged with the server upon connection.

To describe our new task, an **`AttributeMap`** is created (line 15) using factory method **`createAttributeMap()`**. It is possible, however highly discouraged to simply create the map using its default constructor instead. This will disable input validation and may cause other logic to fail.

When generating tasks or assets, it is always a good name to provide them with unique titles to keep track of them. As a tip, `Base-26` is a nice and simple way to trim down the size of large numbers like the current epoch time.

For a new task, task list identifier and initial task state are required input. Even though we are not currently involving an asset, the asset title is often used as the primary title for tasks and is provided here as an example of optional data. Note that when using **`setAttribute()`**, it is important for the second parameter to exactly match the value class of the attribute.

If you find yourself wanting to convert a non-`@Complex` value class between its true form and a `String`, you can make use of the **`setAttributeFromString()`** and **`getAttributeAsString()`** methods. Many other convenient methods are available; the Javadoc is recommended reading.

The client is divided into a number of different API parts. Creating a task, for example, is done by accessing the **`taskApi()`** and from there the actual **`createTask()`** method (line 22). The resulting map will have the identifier of the newly created task set (line 23). More information will also have been added, such as timestamps.

4.2. Managing Tasks

Building upon the first example, let's create a set of ten tasks to play with. We keep the suffix variable to group them up. We will be discarding the return values of the creates, leaving us with no access to the identifiers of the tasks for now.

```
for( int i = 0 ; i < 10 ; i ++ ) {
  final AttributeMap task = client. createAttributeMap();
  task.setAttribute( Attribute.TASK_LIST_ID, "example" );
  task.setAttribute( Attribute.TASK_STATE, TaskState.OPEN );
  task.setAttribute( Attribute.OP_SRC, suffix );
  task.setAttribute( Attribute.ASSET_TITLE, "ManagingTasks " + i + " " + suffix );
```

```
  client.taskApi().createTask( task );
}
```

The SDK comes with an **`AttributeMapRandomizer`** class which can be used generate random attribute values. Useful for conveniently creating test tasks.

```
final FilterCriteria crit = client.taskApi().createFilterCriteria();
crit.getFilterEqualities().setAttribute( Attribute.TASK_LIST_ID, "example" );
crit.getFilterEqualities().setAttribute( Attribute.OP_SRC, suffix );
final FilterResult result = client.taskApi().getTasks( crit, 10, 0 );
for( final AttributeMap task : result.getMatches()) {
  final Long taskId = task.getAttribute( Attribute.TASK_ID );
  System.out.println( "Task with id " + taskId + " is about to be deleted" );
  client.taskApi().deleteTask( taskId );
}
```

The type safe way of doing searches is by using **`FilterCriteria`**. There is also a more powerful/error prone way which entails expressions that are a subset of ecmascript. Starting off with a factory call to create the `FilterCriteria` (line 1), we then access an `AttributeMap` that describes equality checks. We set the task list identifier (line 2), which means no tasks from other tasks lists will be returned.

In order to find all of the tasks we created, we will make use (line 3) of the fact that stored the suffix in the **`OP_SRC`** attribute, which can be searched.

The **`getTasks()`** call takes the criteria, the number of requested tasks, and the offset into the full result set. Together, the two latter parameters allow for pagination. In this case, we happen to request exactly the amount of tasks that were previously created. Looping over all of the results (line 5), we then extract the task identifier for use with the **`deleteTask()`** call (line 8) that completely remove each task.

Note that task deletion will not be used extensively in production code. It is much more useful to close a task by setting its state to one of the closed states. This set of states is exposed as the constant **`TaskState.CLOSED_STATES`**. Examples include `FINISHED` for when the task was completed as expected and `REJECTED` for a task that was likely terminated early on by a manager (i.e. request denied). Leaving these closed tasks around rather than deleting them allows for reporting and troubleshooting.

```
task.setAttribute( Attribute.TASK_STATE, TaskState.FINISHED );
client.taskApi().updateTask( task );
```

Conveniently, the `AttributeMap` class has support for automatically flagging values as dirty. This means that any modifications done between a **`getTasks()`** or **`getTask()`** call and a call to **`updateTask()`** will become part of the update request, whereas attributes that remained untouched will not.

4.3. Managing Assets

Modifying the initial example again, we remove the task code and focus on assets. Depending on the MAM, the set of supported asset types varies. For testing, we recommend you start with what we call an item - which usually maps to the same term on the MAM. The item typically describes a piece of media which may or may not already be ingested and available. Creating an item is done very much like creating a task.

```
AttributeMap item = client.createAttributeMap();
item.setAttribute( Attribute.ASSET_TYPE, AssetType.ITEM );
item.setAttribute( Attribute.METADATA_FORM, "example" );
item.setAttribute( Attribute.ASSET_TITLE, "ManagingAssets " + suffix );
item = client.assetApi().createAsset( item );
final String assetId = item.getAttribute( Attribute.ASSET_ID );
System.out.println( "Item created with id " + assetId );
```

The required attributes here are the asset type, which we set to item (line 2) and the name of the metadata form (line 3). The latter may be optional for some combinations of MAM and asset type. A few attributes, like **`METADATA_FORM`**, are mapped directly within each MAM specific implementation of our generic interface `Mambrella`. Others need to by manually mapped in configuration, or passing them will have no effect. The map returned upon creation (line 6) is likely to contain more information than the map that was posted. Changes to posted attributes may also have been made.

```
item.setAttribute( Attribute.CONT_FMT, "XDCAM-HD422" );
item = client.assetApi().updateAsset( item );
```

Updates are, again, very similar. Dirty checking of attribute values applies.

```
client.assetApi().deleteAsset( AssetType ITEM, assetId );
```

Since both type and identifier are required for unique identification, both are needed to request a deletion. Typically, the deletion request is queued, in which case default grace times and priorities apply.

4.3.1. Unmanaged Metadata (from version 2.4)

MAM level metadata can be read and written using the **`Unmanaged Metadata`** construct. An example is listed below.

```
UnmanagedMetadata md = client.assetApi().getUnmanagedMetadata( AssetType.ITEM, assetId );
client.assetApi().updateUnmanagedMetadata( AssetType.ITEM, assetId,
  UnmanagedMetadata.builder()
  .set( "asset.title", "EXAMPLE " + md.get( "example.identity" ))
    .subs( "asset.contributor", UnmanagedMetadata.builder()
      .set( "person.role", "role1.1" )
      .set( "person.firstName", "Example" )
      .set( "person.lastName", "von Beispiel" ))
    .build()) ;
```

4.4. Managing BPM Process Instances

Creating a process instance is very similar to creating a task or an asset. The trick is figuring out what to put in the `AttributeMap` and then call the create method. In the case of BPM integration, we are generally able to push our attributes directly rather than map them to predefined fields within the target system.

```
final String id = client.bpmsApi().createProcessInstance( "import",
  client.createAttributeMap()
    .setAttribute( Attribute.ASSET_TYPE, AssetType.ITEM )
    .setAttribute( Attribute.ASSET_ID, "urn:example:placeholder1" )
    .setAttribute( Attribute.FILE_NAME, "sports-ball.mxf" ));
System.out.println( "Process instance created with id " + id );
```

The above example creates an instance of a fictional import process which needs only a file name for the actual decision making and then causes media to be added to a placeholder item. The instance identifier is typically stored in the **`BPMS_INST`** attribute of the task that initiated it (if any). The current values of any instance data matching an attribute can be read by calling **`getProcessInstanceData()`**.

Next up, we are going to perform a search. Some BPMS specific limitations may apply.

```
for( final AttributeMap inst : client.bpmsApi().listProcessInstances(
  "import", client.createAttributeMap()
              .setAttribute( Attribute.FILE_NAME, "sports-ball.mxf" ))) {
                System.out.println( "Active, matching process instance: "
                + inst.getAttributeAsString( Attribute.BPMS_INST ));
              }
```

Instances are always filtered by their process name, but further filtering is also possible. The map passed is very similar to that of the `FilterCritiera`.**`getFilterEqualities()`**. Any value set becomes an equality check vs the data stored in active process instances.

```
client.bpmsApi().updateProcessInstance(
  id, BpmsSignal.PASS, client.createAttributeMap()
    .setAttribute( Attribute.QC_RESULT, "success" ));
```

Updating a process instance requires not only the identifier and the changes to be made, but also a signal. The signal is used to trigger the process to continue at one or more nodes where it is currently waiting for an update.

If you feel that the best course of action is to delete a process instance, rather than letting it run its course and retire on its own, there is a **`deleteProcessInstance()`** method available.

4.5. Receiving Events from a Message Queue

Moving to message queue events, things start to get a bit more involved. While most of the complexities of JMS have been hidden, there are too many dependencies in play for us to maintain factory methods like those found in previous examples. If you are not proficient in the use of Guice, you may want to read up now before proceeding. Below is a new full example.

```
package com.mayam.wf.ws.client.example.tutorial;

import javax.inject.Inject;

import com.google.inject.Guice;
import com.google.inject.Injector;
import com.mayam.wf.attributes.shared.Attribute;
import com.mayam.wf.mq.Mq;
import com.mayam.wf.mq.Mq.ListenIntensity;
import com.mayam.wf.mq.MqDestination;
import com.mayam.wf.mq.MqMessage;
import com.mayam.wf.mq.MqModule;
import com.mayam.wf.mq.common.ContentTypes;

public class ReceivingEvents {
  public static void main( String [] args ) throws Exception {
    final Injector injector = Guice.createInjector(
      new MqModule( "ReceivingEvents" ));
    injector.getInstance( ReceivingEvents.class ).loop();
  }

  @Inject Mq mq;

  public void loop() throws Exception {
    mq.attachListener( MqDestination.of( "queue://mayam.task.test" ) ,
      new Mq.Listener() {
        @Override public void onMessage( MqMessage message ) throws Throwable {
          if( message.getType().equals( ContentTypes.ATTRIBUTES )) {
            System.out.println( "Got a message related to " + "task/asset with title " +
            message.getSubject().getAttribute( Attribute .ASSET_TITLE));
          }
        }
      });
  while( System.in.available() == 0 )
    mq.listen( ListenIntensity.RELAXED );

  mq.shutdownConsumers();
  }
}
```

A Guice module which binds message queue related dependencies is provided, and used in the example to create an `injector` (line 17). The module takes the name of your application as parameter (line 18) for reasons related to message replay.

An instance of the class is created (line 19), whereupon the mq object will be injected (line 22). Within our `loop()` method that is called, a message listener is created and registered (line 25). Listeners can be attached to topics or queues.

It is recommended that you create a queue for use with your application and configure **Camel**, or whatever routing framework is connected to the message queue, to copy relevant messages

into your queue. Our Topics class contains the topics to which we post information and, in most cases, route them to ours on queues, available in the Queues class.

The messages themselves are almost always `AttributeMap` based when they originate from our software, in which case we are able to recreate the map (line 32) and extract the information we need.

In order for the attached listener to trigger, we must actually allow it to listen (line 39). The **`ListenIntensity`** is an abstraction over `timeout` settings. Generally, go with the relaxed approach, machine don't necessarily handle stress better than their human counterparts.

For the purpose of this example, the code will loop until enter is pressed on the console (line 38). Our own code is run almost exclusively in servlets and the loops are set to terminate when container shutdown is initiated.

Since both consumers and producers of messages have their own threads, a clean shutdown requires that these too be terminated. The example does not send messages, so only the consumers need be shut down (line 41).

4.6. Calling Site Hooks

A new function in 2.6 is the ability to call a **site hook**. A site hook is a locally registered site-specific function identified by its **`SITE_ACTIVITY_ID`**. In addition, an extra call parameter can be passed in the attribute map as **`SITE_ACTIVITY_PARAM`**.

See the code snippet below for an example of how to call the site hook using the **Activity API**.

```
result.setAttribute(Attribute.SITE_ACTIVITY_ID, "id_of_site_activity");
result.setAttribute(Attribute.SITE_ACTIVITY_PARAM, "parameter passed to site activity");
// Call the site hook
client.activityApi().performSiteActivity(result);
```

5. Rest API Tutorial

The online API reference is available for browsing at the same location as the SDK downloads and documentation, the `tasks-ws/` of your Mayam server. This section aims to give you a few pointers to get started.

5.1. Non-Programmatic Access to the API

If you are opting for this path rather than using our SDK, then your language and framework of choice is unknown to us, which means we are not able to provide you with code examples. The de facto standard for providing guidance, that we will adhere to, is command-line calls using the curl client which is readily available for free online.

5.1.1. My First Task

The request and reply bodies are in `JSON` format, and kept as simple as possible. In the case of the `AttributeMap`, some of the values are as complex and not covered in this tutorial but are described in the online reference. For simple values, their nearest `JSON` equivalent is used. Dates take the form of the extended combination of date and time of day described in chapter 5.4 of **ISO 8601**; more commonly referred to as `xs:dateTime` since it became popular in `XML` documents. Our timestamps will always be sent with a **UTC timezone** and thus ending with a `Z`.

```
{
  "complete_by_date" : "2015-02-14T10:18:54.467Z",
  "task_list_id" : "ingest",
  "task_state" : "PENDING"
}
```

Note above that the attribute names are used in lowercase when used as keys for an `AttributeMap` in `JSON`. Any Java enum, including the attributes, will remain uppercase when used as values (`PENDING`, line 3).

```
curl -H 'X-ApiToken: test:test' \
  -X POST -d @ new_task.json \
  -H 'Content-type: application/vnd.mayam.attributemap-v1+json' \
  -H 'Accept: application/vnd.mayam.attributemap-v1+json' \
  'http://localhost:8084/tasks-ws/rest/tasks/'
```

The API token is passed using the **`X-ApiToken`** custom header (line 1), alternatively using the **`TasksToken`** cookie. After setting the method to **`POST`** and piping in our `JSON` file (line 2), we specify that we are sending (line 3) and receiving (line 4) `AttributeMap` representations. All of it is sent off to the tasks resource.

```
{
  "task_updated_by" : null,
  "task_state" : "PENDING",
  "task_created_by" : null,
  "task_list_id" : "ingest",
  "task_updated_by_system" : "rest:test",
  "complete_by_date" : "2015-02-14T10:18:54.467Z",
  "task_id" : 68039
}
```

Looking at the `JSON` sent back, we see that the task was created not by a user (line 2) but by a `REST` call (line 6) made by the test account (*secret not logged*). The identifier is provided of course (line 8).

5.2. Managing Tasks

Since the reader is now introduced to the `JSON` representation of attributes, it makes sense to make use of that knowledge when discussing task filtering. Using the SDK, we made use of the type safety of **`FilterCriteria`**, whereas we are now moving on to the flexibility of the **Ecmascript** expressions.

```
{
  "expression" : "s.TASK_LIST_ID == 'import' && \
      s.COMPLETE_BY_DATE <= now() && s.ASSIGNED_USER == p.USER",
  "expressionObjects" : {
    "p" : {
      "USER" : "sarah"
    }
  }
}
```

Expression and supporting objects: consider the expression as a question posed in turn to every task, loaded into the variable `s`. If the boolean expression evaluates to true, the task is included in the result set. Obviously, the underlying database access is much more efficient than that.

The example expression above has three parts, all of which need to be true. The first one will likely always be present; a task list needs to be specified in all but the most generic reporting cases.

Although the Ecmascript subset used does not support side effects, basic function support is available, exemplified here by the **`now()`** function that returns the current timestamp (database time if supported, otherwise server time).

The third one is tricky because it involves a variable named `p`, which defined in the expression objects. Instead of writing `p.USER` we could have written `'sarah'`. The point is of course to have the expression be hard coded into your software and separately pass along anything dynamic.

```
curl -H 'X-ApiToken: test:test' \
  -X POST -d @ filter.json \
  -H 'Content-type: application/vnd.mayam.filterexpression-v1+json' \
  -H 'Accept: application/vnd.mayam.attributemap-collection-v1+json' \
```

```
  'http://localhost:8084/tasks-ws/rest/tasks/filtered'
```

The result is a JSON array of the same kind of maps returned by creating a task.

5.3. Managing Assets

The similarities between creating a task and an asset are equally noticeable in the `REST API`. To make the following example a bit more interesting, a complex value type was added.

```
{
"asset_type" : "ITEM" ,
"metadata_form" : "example" ,
"asset_title" : "One great looking item" ,
"asset_site_id" : "12345-one-great-looking-item" ,
"asset_access" : {
  "standard" : [
    {
      "entityType" : "GROUP" ,
      "entity" : "cataloguers" , "read" : true ,
      "write" : true ,
      "admin" : false
    }]
  }
}
```

Asset type and metadata form are still required, and in case the system is not configured to produce them itself you may need to provide a site identifier. Titles are generally a good idea to set.

The complex value chosen for our example is an access control list, ACL, which determines who gets to do what with the asset about to be created. Somewhat MAM specific, the gist of this particular ACL is that cataloguers get to read and update catalogue information but not change access rights or perform media related operations.

```
curl -H 'X-ApiToken: test:test' \
  -X POST -d @ new_asset.json \
  -H 'Content-type: application/vnd.mayam.attributemap-v1+json' \
  -H 'Accept: application/vnd.mayam.attributemap-v1+json' \
  'http://localhost:8084/tasks-ws/rest/assets/'
```

The matching curl command line. As with tasks, the result is an AttributeMap representation.

```
curl -H 'X-ApiToken: test:test' \
  -X DELETE \
  -H 'Accept: application/vnd.mayam.attributemap-v1+json' \
  'http://localhost:8084/tasks-ws/rest/assets/asset-ITEM-123456789'
```

When deleting, or otherwise referencing an existing asset, both the type and the identifier need to be specified. In the above example, those are `ITEM` and `123456789` respectively. Deleting an asset will first return a copy of it, which is why we specify an **`Accept`** header.

When referencing an aspect of an asset, the URL gets more specific. Even more resources and operations can be found in the reference manual, but the last asset example for this tutorial will be an uningest - a request to delete the media of an item.

```
curl -H 'X-ApiToken: test:test' \
  -X DELETE \
  'http://localhost:8084/tasks-ws/rest/assets/asset-ITEM-123456789/media'
```

5.4. Managing BPM Process Instances

Recreating the process instance creation of the SDK example gives us the following `JSON` representation of the attributes.

```
{
  "asset_type" : "ITEM",
  "asset_id" : "urn:example:placeholder1",
  "file_name" : "sports-ball.mxf"
}
```

The map is then posted to the server. Note the process-prefix before the process name.

```
curl -H 'X-ApiToken: test:test' \
  -X POST -d @ new_pi.json \
  -H 'Content-type: application/vnd.mayam.attributemap-v1+json' \
  -H 'Accept: application/vnd.mayam.job-v1+json' \
  'http://localhost:8084/tasks-ws/rest/bpms/process-ingest/'
```

From the perspective of Mayam Tasks, a process instance is like any other long running job which is why the instance identifier is passed as a job identifier. In order to retrieve (**`GET`**) or update (**`UPDATE`**) data for an instance, or to delete an instance (**`DELETE`**), the resource is `/bpms/instance-12345` for instance having the identifier `12345`. Note that for updates, the signal is not part of the query but instead posted as value for the **`BPMS_SIGNAL`** attribute.

5.5. Receiving Events from a Message Queue

Event reception from the message queue is not supported in the `REST API`. To receive task and asset events, a native interface implementation is required. In most cases, this translates to implementing a native `JMS` reader that typically reads messages from Apache ActiveMQ. Several internet resources exist that provides implementation guidance for the relevant Message Queue, protocol (`JMS/Stomp`) and programming language. If you plan to use Java, the SDK client provides necessary functions for sending and receiving Tasks messages.

6. Reference Documentation

For general technical information on Mayam Tasks, refer to the technical reference manual. This manual is available at: http://your-hostname-here:8084/tasksdoc/tasks_technical_reference.pdf

6.1. SDK and API Online Reference Documentation

The reference documentation for the Mayam Tasks Java SDK and REST API is available directly from the service endpoint: http://your-hostname-here:8084/tasks-ws

Please note that the coding examples shown in this manual are also available from this location.