Unified Mediation Bus Adapter Development Guide

Version 1.1
Edition: 1.0

Hewlett Packard Enterprise

Notices

Table of Contents

Figures

Tables

Preface

About this guide

This guide provides an overview of the Unified Mediation Bus product and describes how to create Mediation Adapters to connect Alarm or Event provider and consumer applications.

Product Name: Unified Mediation Bus Adapter Development Toolkit

Product Version: V1.1

Intended Audience

Here are some recommendations based on possible reader profiles:

- Solution Developers
- Software Development Engineers

Software Versions

The software versions referred to in this document are as follows:

Product Version	Supported Operating systems
Unified Mediation Bus Adapter Development Toolkit V1.1	• Windows XP / Vista • Windows Server 2007 • Windows 7 • Red Hat Enterprise Linux Server release 6.x & 7.x

Table 1 - Software versions

Typographical Conventions

`Courier` Font:

- Source code and examples of file contents.
- Commands that you enter on the screen.
- Pathnames
- Keyboard key names

Italic Text:

- Filenames, programs and parameters.
- The names of other documents referenced in this manual.

Bold Text:

- To introduce new terms and to emphasize important words.

Associated Documents

The following documents contain useful reference information:

[R1] `HP Unified Mediation Bus– Installation and Configuration Guide`

Support

Please visit our HP Software Support Online Web site at softwaresupport.hpe.com for contact information, and details about HP Software products, services, and support.

The Software support area of the Software Web site includes the following:

- Downloadable documentation.
- Troubleshooting information.
- Patches and updates.
- Problem reporting.
- Training information.
- Support program information.

Chapter 1
Introduction

This guide gives an overview of the Unified Mediation Bus and explains how to create a new mediation Adapter project with the provided Unified Mediation Bus Adapter Development Toolkit.

1.1 Overview

Unified Mediation Bus allows several applications to exchange Events (and by extension Alarms) with each other. It also provides features for executing actions remotely: alarm operations (creation, grouping, deletion etc.), Trouble ticket operations, command executions (shell scripts, java, etc.)

The Unified Mediation Bus product comes in replacement of the legacy "NGOSS Open Mediation" product with the aim to provide:

- Better performance
- Better robustness
- Easier deployment
- Easier Adapter Development

Unified Mediation Bus is constructed around two main technologies:

- A **common registry**, and remote execution service implemented with the Hazelcast® technology. Hazelcast provides both:
  - o a common registry feature that centralizes configuration, status and monitoring information on all UMB Adapters that are part of the overall UMB solution
  - o a distributed executor service feature that provides a framework for executing actions on UMB Adapters across the whole UMB solution
- A **message broker** based on the Kafka Technology. Apache Kafka / Apache ZooKeeper provide a high-performance, high-availability, reliable framework for producing and consuming collections of alarms or events across the whole UMB solution

A typical UMB solution is composed of (see figure below):

- A UMB Server product installation, usually installed on one or more dedicated UMB Server host(s), that contains Apache Kafka / Apache ZooKeeper
- Several UMB Adapter[1] product installations (one for each Application connected to the UMB solution). Each application has its own dedicated UMB Adapter, usually installed on the same host as the application itself.

[1] UMB Adapters are developed using the UMB Adapter Development Kit. Information on how to install the UMB Adapter Development Kit is provided in the [R1] `HP Unified Mediation Bus– Installation and Configuration Guide`

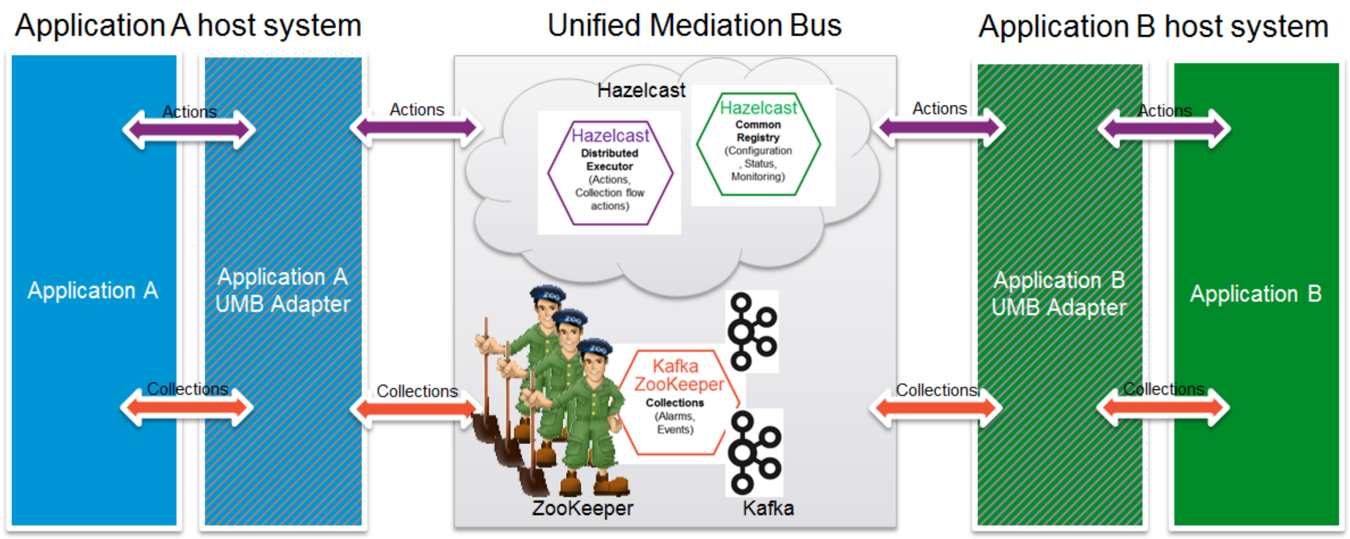

Application A host system
Unified Mediation Bus
Application B host system
Application A
Application A UMB Adapter
Actions
Collections
Hazelcast
Hazelcast Distributed Executor (Actions, Collection flow actions)
Hazelcast Common Registry (Configuration, Status, Monitoring)
Kafka ZooKeeper Collections (Alarms, Events)
ZooKeeper
Kafka
Application B UMB Adapter
Application B

Figure 1 - Unified Mediation Bus architecture overview

The above figure shows UMB interconnecting 2 separate applications: Application A and Application B.

In the figure, Hazelcast appears as a centralized component for simplification's sake: Hazelcast is in fact distributed across both Application A and Application B UMB Adapters. Each of the UMB Adapters is a Hazelcast cluster member. Hazelcast cluster members are interconnected directly, without any centralized component.

Any UMB Adapter can act as an action service provider and/or consumer:

- It provides action services for the Application that it is associated with (in our case Application A or Application B). UMB Adapters act as proxies to execute actions on Applications that they are associated with.
- It consumes action services from other UMB Adapters

On the other hand, Apache Kafka / Apache ZooKeeper are indeed a centralized component. Both Application A and Application B UMB Adapters connect to the same central component. Apache ZooKeeper provides a high performance coordination service for the "cluster" of Apache Kafka brokers. Apache ZooKeeper acts as a front-end to the Apache Kafka brokers. The Apache Kafka brokers provide the messaging service: they store collections of alarms or events (sent by Kafka producers) as Topics. Kafka consumers then retrieve the collections of alarms or events.

Any UMB Adapter can act as Kafka producer and/or Kafka consumer:

- It provides collection services for the Application that it is associated with (in our case Application A or Application B). UMB Adapters act as proxies to collect alarms or events from Applications that they are associated with.
- It consumes collection services from other UMB Adapters

1.1.1 The Mediation Common Registry

The Mediation Common Registry is a common (shared grid in-memory) storage implemented using the Hazelcast® Technology that allows all mediations contributors (the Adapters) to register information.

This information identifies adapters that are part of the mediation solution but also gives a description of the services they provide. The services are of two types:

- Flow services
- Action services

Using the Common Registry information, any adapter is able to know about all the other adapters and also get their status or the definition (description) of the services they offer.

At Adapter startup time the local Adapter configuration is automatically made available by the Adapter Framework to the Common Registry. This prevents any complex configuration on each side when one adapter wants to communicate with another one.

The Common Registry can be schematically represented as follows:

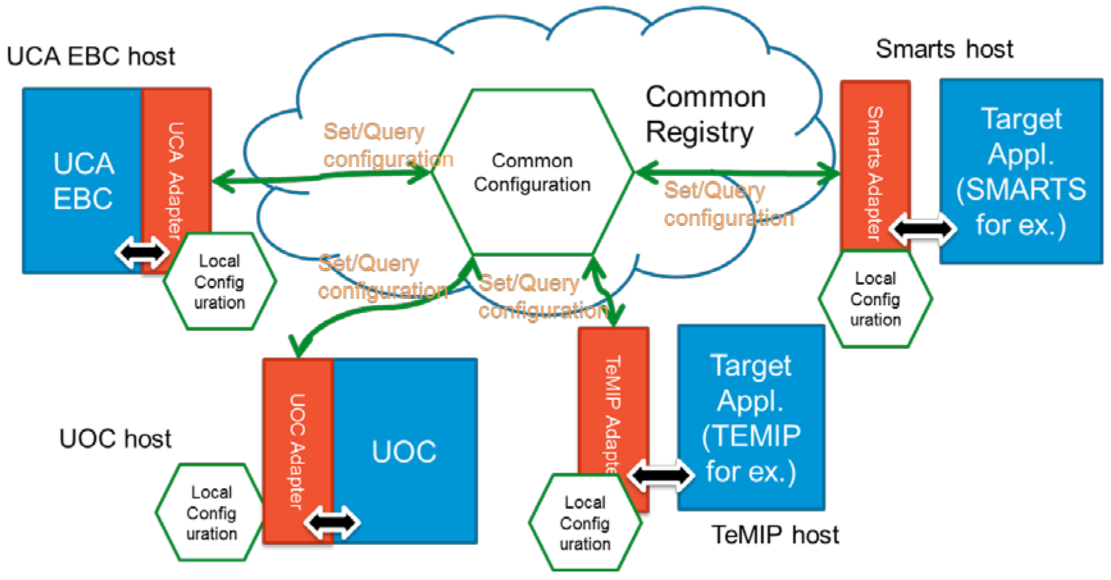


Figure 2 - Mediation Common Registry overview

1.1.1.1 Using Hazelcast® for Actions implementation

Hazelcast® provides an efficient distributed executor service to execute Callable and Runnable instances on the remote cluster members. The Unified Mediation Bus uses this feature to implement Actions. Doing this way there is no additional configuration to perform. Any Mediation member (Adapter) can potentially be an action executor.

Action services are defined in the Adapter Configuration file and made available in the Common Registry.

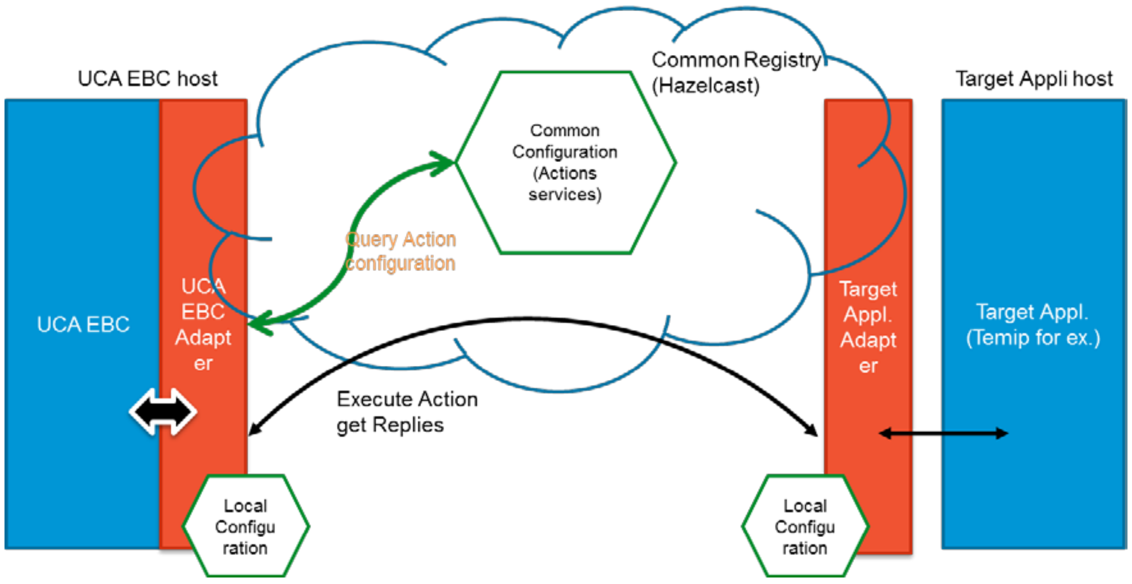

UCA EBC host
Common Registry (Hazelcast)
Target Appli host
Common Configuration (Actions services)
Query Action configuration
UCA EBC
UCA EBC Adapt er
Execute Action get Replies
Target Appl. Adapt er
Target Appl. (Temip for ex.)
Local Configu ration
Local Configu ration

Figure 3 - Unified Mediation Bus Action Mechanism overview

1.1.2 The Message Broker

The Unified Mediation Bus message exchange is based on the Apache Kafka technology.

The Kafka message broker is one of the fastest message brokers. It offers off the shelf message persistency (persistency duration is configurable). It has a strong ordering guarantee and offers High Availability (via redundancy and through the use of ZooKeeper).

The Unified Mediation Bus allows defining Event Collection Flows between an Event producer and an Event Consumer.

The collection flows are of two types:

- Static flows
  - ✓ One Producer for several possible Consumers. Each of the consumers which don't belong to the same group will receive a copy of the produced events
  - ✓ Can produce events even if no Consumer is waiting for them. Events are persisted in the Kafka log system for a configurable amount of time
  - ✓ One Kafka Topic per static flow

- Dynamic flows
  - ✓ One Producer for One Consumer.
  - ✓ Production is done only upon Consumer request (create Flow request). The producer must be up and running for the dynamic flow to be established successfully.
  - ✓ One Kafka Topic per consumer / producer pair

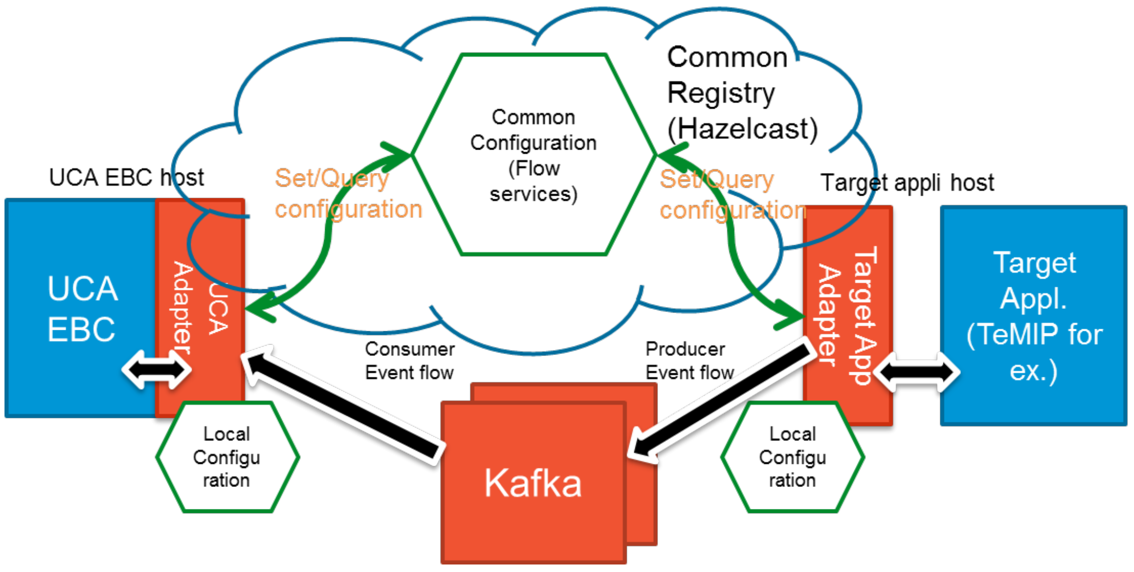


Figure 4 - Unified Mediation Bus flows overview

1.2 Unified Mediation Bus principles

1.2.1 The Unified Mediation Bus Server (broker)

The Unified Mediation Bus Server or broker as mentioned above is implemented with Kafka. To be more precise it is in fact the ZooKeeper / Kafka association that implements the broker.

A simple Unified Mediation Bus Server configuration can be made with one ZooKeeper instance and one Kafka instance on a single Linux box.

However, for a production environment, a highly available Unified Mediation Bus Server should be redundant. As such, it must be made of at least two Kafka servers and three Zookeeper instances.

The Unified Mediation Bus Server kit is delivered for Linux Only. The Zookeeper and Kafka servers are installed as Linux services.

Please refer to the `[R1] Unified Mediation Bus installation and configuration Guide` for details on ZooKeeper and Kafka configuration and administration.

1.2.2 The Unified Mediation Bus Adapters

The Unified Mediation Bus Adapters are key components of the mediation.

- On the Provider side, they are defining and implementing the Flows and Action services.
- On the Consumer side, they are implementing the action requests, and flow consumers.

Of course, an Adapter can be both a service provider and a service consumer at the same time meaning that it can provide services to other adapters while consuming services from another adapter.

A mediation Adapter can be of two types:

- Embedded

  When embedded, the Adapter components (All the Java classes representing the adapter) are running in the same JVM than the application using the Adapter. This allows for a more efficient communication between the application and the adapter components (procedure calls) and an easier monitoring because the adapter has the same life time that the application.

  A typical example of an embedded Adapter is the UCA-EBC Adapter which shares the same process as UCA-EBC.

  Applications implementing an embedded Adapter must provide the Adapter configuration files on their Java class path.

- Standalone

  A Standalone Adapter runs in its own JVM. It must implement a `main()` method and provide its own configuration files.

  A standalone Adapter must implement a communication technology to communicate with the Application it serves. The communication technology choice it usually driven by the application capabilities (Web services, specific API, sockets etc….)

  A standalone Adapter is usually used when there is no way to integrate the Adapter classes into an existing application (3rd party application, or non-Java application)

  A typical example of a Standalone Adapter is the TeMIP Adapter that communicates with TeMIP for collecting alarms and executing actions using the TeMIP Web Services (TWS) component.

An Adapter is identified by its name in the AdapterConfiguration.xml file:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="AdapterName" actionGroup="GroupName" version="1.0"
xmlns="http://hp.com/umb/config">
</adapter>
```

A <adapter>…</adapter> XML element can have the following optional attributes:

- name

Each Adapter part of the same mediation solution (bound to the same Common Registry) must have a distinct name. An attempt to start an Adapter with the same name as another adapter already bound to the Common Registry will result in an error preventing the start of the Adapter.

- actionGroup

The Adapter's actionGroup attribute is optional and used for horizontal scaling of actions. Adapter action group names should be different than adapter names, throughout your whole UMB solution. If several adapters in the UMB solutions share the same action group, one can request execution of actions on the action group itself. This is done by specifying an action group (instead of an adapter name) as the target of the action. The action will then be executed on a randomly selected adapter from the action group. This provides load-balancing among adapters that can perform identical tasks.

The following figure illustrates action execution when load balancing is not used (i.e. the target of each action is a specific adapter):

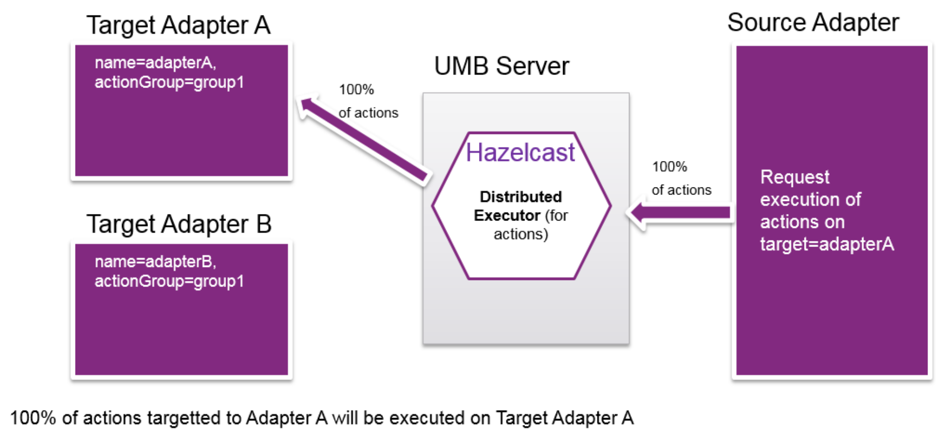

Target Adapter A
name=adapterA, actionGroup=group1
Target Adapter B
name=adapterB, actionGroup=group1
100% of actions
UMB Server
Hazelcast
Distributed Executor (for actions)
100% of actions
Source Adapter
Request execution of actions on target=adapterA
100% of actions targetted to Adapter A will be executed on Target Adapter A

Figure 5 - Action execution without load balancing

In the figure above, load balancing of actions is not used and all actions are executed by the adapter that is specifically targeted.

On the other hand, the following figure illustrates action execution with load balancing (i.e. the target of each action is an action group instead of a specific adapter):

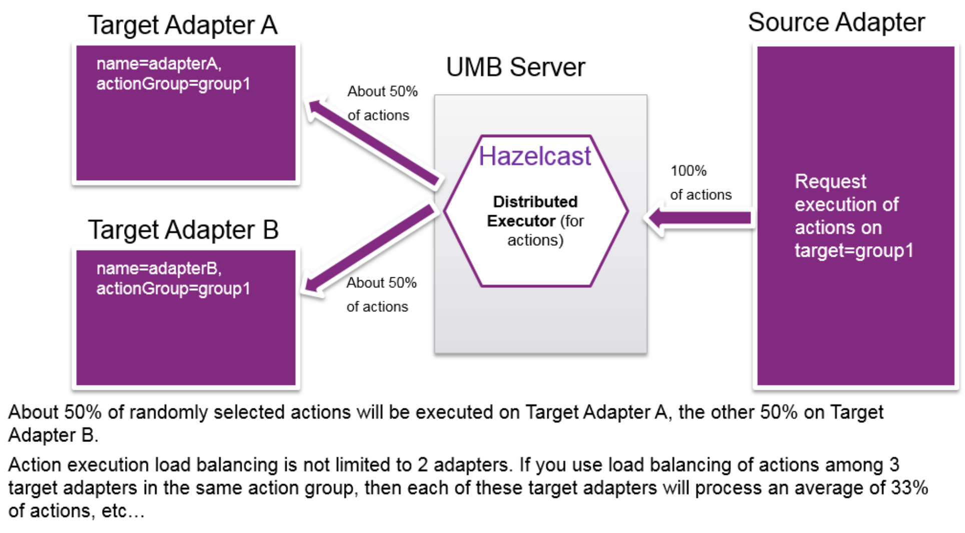

Target Adapter A
name=adapterA, actionGroup=group1
Target Adapter B
name=adapterB, actionGroup=group1
About 50% of actions
About 50% of actions
UMB Server
Hazelcast
Distributed Executor (for actions)
100% of actions
Source Adapter
Request execution of actions on target=group1
About 50% of randomly selected actions will be executed on Target Adapter A, the other 50% on Target Adapter B.
Action execution load balancing is not limited to 2 adapters. If you use load balancing of actions among 3 target adapters in the same action group, then each of these target adapters will process an average of 33% of actions, etc…

Figure 6 - Action execution with load balancing

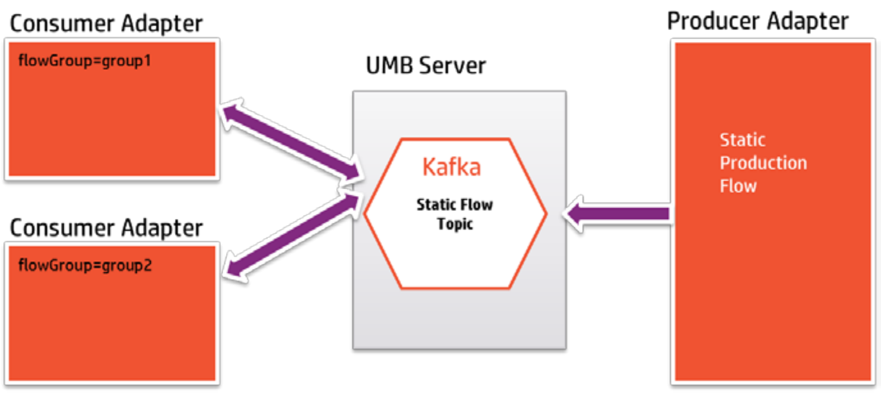
In the figure above, load balancing of actions is used and on average 50% of actions are executed on each adapter part of the targeted action group.

- flowGroup

  The flowGroup labels the adapter as belonging to a consumer group.

  The flowGroup has a meaning only for Static flows for which we can have several consumers for one producer. Each message published to a StaticFlow is delivered to one instance within each subscribing consumer group.

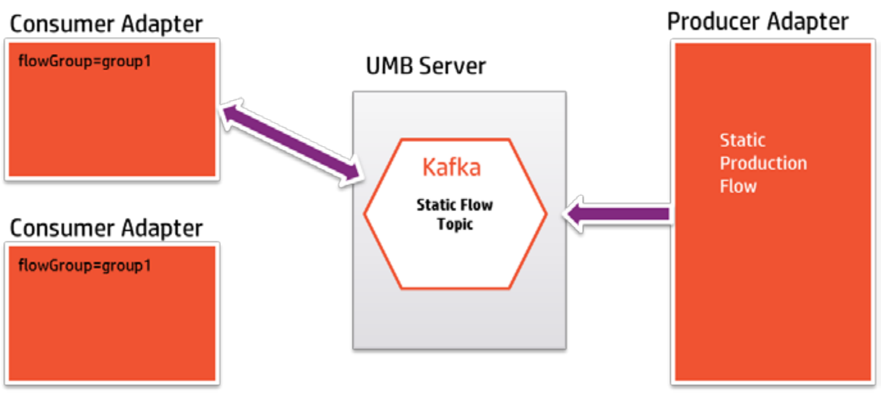
  If all the consumer instances have different consumer groups, then this works like publish-subscribe and **all messages are broadcast to all consumers**.



  Here the two consumer adapters belong to different flow groups and they both receive the same flow of events.

  If all the consumer instances have the same consumer group, then this works just like a traditional queue balancing load over the consumers balancing the topics partitions between consumers.

  For UMB flows that have single one partition (the default), this means that only one of the flow consumer belonging to the same flow group will get the messages:



  In such configuration the second adapter is just on hold. If the adapter receiving the events stops or loses the connection with the Kafka server, then this second adapter will get the messages that were not previously collected by the first adapter.

  In the case the flow is created with several partitions, the partitions will be balanced over the consumers. Each consumer will then receive messages from the partitions that have been allocated to it. If one of the consumer stops, the partitions allocated to it will be reallocated to the other available consumers.

1.2.3 The Unified Mediation Bus Adapter Services

A Unified Mediation Bus Adapter can implement two types of services:

- Flow Services
- Action Services

1.2.3.1 Flow Services

Flow services are defined in the `<flowServices>` section of the `AdapterConfiguration.xml` file.

A Flow definition specifies a collection channel provided by this Adapter.

Example of Flow Service definitions:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="FileAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <flowServices>
        <flow name="AlarmFileStaticFlow" type="Static"
              collectorClass="com.hp.umb.adapter.file.FileCollector">
            <parameters>
                <parameter key="fileName" defaultValue="data/alarms.xml"/>
            </parameters>
        </flow>
        <flow name="AlarmFileDynamicFlow" type="Dynamic"
              collectorClass="com.hp.umb.adapter.file.FileCollector">
            <parameters>
                <parameter key="fileName" defaultValue="data/alarms.xml"/>
            </parameters>
        </flow>
        <flow name="TemperaturesStaticFlow" type="Static"

collectorClass="com.hp.umb.adapter.file.TemperaturesCollector">
            <parameters>
                <parameter key="fileName"
defaultValue="data/temperatures.csv"/>
            </parameters>
        </flow>
        <flow name="TemperaturesDynamicFlow" type="Dynamic"

collectorClass="com.hp.umb.adapter.file.TemperaturesCollector">
            <parameters>
                <parameter key="fileName"
defaultValue="data/temperatures.csv"/>
            </parameters>
        </flow>
    </flowServices>
</adapter>
```

A Flow is identified by a '`name`' attribute and must specify a '`collectorClass`' attribute (i.e. the class that collects information from the application and produces the messages to the UMB flow)

Flows can be of two different types: 'Static' or 'Dynamic'.

- Static flows

  Static flows are automatically started when the Adapter is started (unless the `autoStarted` attribute is set to `'false'`). Message production starts even if there is no requester. The messages are sent to Kafka which stores them for a configurable period of time.

Several consumers can consume the same static production flow. Each of the consumers will receive all the produced messages (at the condition that they do not belong to the same flowGroup). A Static flow can be seen as message broadcasting between the producer Adapter and multiple consumer Adapters.

The name of the Kafka topic for a static Flow follows a specific pattern (without the quotes):

`“producer Adapter name”-“flow name”`

From the example above, the topic name for the flow `AlarmFileStaticFlow` will be:

`FileAdapter-AlarmFileStaticFlow`

- Dynamic flows

  Dynamic Flows are started upon flow consumer request. A Dynamic Flow can be seen as a peer to peer connection between a consumer Adapter and the producer Adapter.

  The Kafka topic name for a Dynamic Flow is constructed as follows (without the quotes):

  `“consumer Adapter name"-"requester Identifier"-"producer Adapter name“-“flow name”`

  As an example, if a UCA-EBC value pack named ‘vp1’ requests the creation of a dynamic flow named `AlarmFileDynamicFlow`, the Kafka topic name will be:

  `UCA-EBC-vp1-FileAdapter-AlarmFileDynamicFlow`

Flow services are defined by the `<flowServices>…</flowServices>` XML element.

Each individual flow service is defined by a `<flow>…</flow>` XML element inside the `<flowServices>…</flowServices>` XML element. There can be as many flow services defined as needed.

A `<flow>…</flow>` XML element can have the following attributes:

Mandatory attributes:

- collectorClass: The full name of Java class that implements the message production. There is no default value. The ‘collectorClass’ attribute is mandatory. The collectorClass Java class has to extend the `com.hp.umb.adapter.collector.BaseCollector` class and implement the `com.hp.umb.adapter.collector.CollectorInterface` Java interface.
- name: The name of the flow. This name will be referenced by UMB Adapters wishing to consume this flow.
- type: The type of the flow: either “Static” or “Dynamic” (see the differences between static and dynamic flows above). There is no default value.

Optional attributes

- autoStarted (optional default true): This is a boolean attribute. This attribute is applicable only if the flow type is “Static”. It indicates that the UMB Framework should start this flow automatically at adapter start.

  If not started automatically, the flow can be started later-on by using the ‘umb’ command.

- broker: this is the name of the ‘broker’ (Kafka server instance) to which the flow is published. This attribute’s value must be added in the `adapter.properties` file.
  For example: for a configuration like the following `<flow … broker=”broker1”></flow>,` the `adapter.properties` file must be updated accordingly by adding the following lines:
  ```
      broker1.producer.bootstrap.servers=brokerhost:9092
  ```

```
broker1.zookeeper.connect=brokerhost:2181
[…]
```

- lastEventReceivedFirstDuringResynchronization (optional default **false**): This is a boolean attribute. It indicates whether collection events/alarms are received in chronological order (in which case the flag has to be set to **false**) or not (flag set to **true**) during resynchronization. By default, if this attribute is not present, the flag is assumed to be **false**, which means that events/alarms are received in chronological order during resynchronization.
- logCleanupPolicy (optional, default value = **'`delete`'**): Specifies the cleanup policy for the flow: '`delete`' or '`compact`'. When the '`delete`' policy is chosen (the default), messages in the topics are deleted after the '`retentionMinutes`' time as elapsed (See below for more information on the '`retentionMinutes`' attribute). When the '`compact`' policy is chosen, only the last instance of messages having the same identifier is kept in the topic. (See Apache Kafka log compaction policy for details).
- monitoringRestartPeriod (optional. Unit: milliseconds default **30000**): The monitoring restart period in milliseconds. This is the time between two re-start attempts in case of flow disconnection. Default value is 30000 milliseconds, i.e. 30 seconds.
- numberOfPartitions (optional default **1**): Creates the topic with the specified number of partitions.
  Note: refer to section "4.7 Specifying a custom Partitioner" in this document for instructions on how to create and use a partitioner class.

NOTE:
It is a known limitation that the `numberOfPartitions` attribute only works on new Topics. Existing topics will ignore the `numberOfPartitions` attribute.

- replicationFactor (optional default **1**): Defines the number of servers where the topic will be replicated. Replication allows automatic failover in case of failure of one or more servers in the cluster. Default value is 1 meaning no replication.
- retentionMinutes (optional): sets the maximum retention time in minutes before cleaning old segments of the log. After the retention time has elapsed, if the *logCleanupPolicy* (see above for more information on the logCleanupPolicy attribute) is set to '`delete`' the elapsed logs are deleted. If the *logCleanupPolicy* is set to 'compact', the logs are compacted keeping the last occurrence of events having the same identifier. Default value is 60 minutes

  serializerClass (optional): The full name of the Java class that serializes the flow messages. When no value is specified, the default serializer class (the `com.hp.umb.adapter.internal.utilities.JavaClassSerializer` class which uses the standard Java class serialization mechanism) is used. A custom serialization class must implement the following interfaces:

```
org.apache.kafka.common.serialization.Serializer<Object>
org.apache.kafka.common.serialization.Deserializer<Object>
```

- topicName (optional): Explicitly sets the name of the topic that will be used by a static flow. The topic will be created only if it does not already exist, otherwise the existing one will be used.

Each Flow can define a list of parameters using a `<parameters>`…`</parameters>` XML element inside a `<flow>`…`</flow>` XML element.

1.2.3.1.1 Flow Parameters

The parameters are a list of configuration values (key/value pairs) that can be specified by the flow creation requester.

A set of attribute help specifying parameters properties. Such attributes are:

- key: The ‘key’ attribute specifies the Parameter name. It is mandatory to specify a value for the ‘key’ attribute.
- defaultValue: this attribute gives the Parameter a default value. In case this parameter is not specified by the flow creation requester, the flow service provider will set this parameter with this default Value at flow creation time.
- overridable: the ‘overridable’ attribute is a boolean attribute. When set to ‘false’, the flow creation requester cannot override the parameter. When omitted the Parameter remains overridable (similar to overridable=’true’). This is particularly useful when the flow service developer wants to protect the parameter definition.
- occurs: the ‘occurs’ attribute can take the value ‘once’ or ‘many’. By default, the same parameter can only be specified once by the requester. If the occurs=’many’ attribute is not set, specifying the same parameter more than once will lead to a flow creation failure.
- mandatory: the ‘mandatory’ attribute indicates that this parameter must be specified by the requester. If not specified, the flow creation execution will return a failure.

1.2.3.2 Defining Flow Service Consumers by configuration

If the `AdapterConfiguration.xml` file allows for the definition of flow services (production side), it also allows for the definition of flow consumers that are automatically created when the adapter is started.

Consumer flows that start automatically are defined by adding the `<autoConsumers>…</autoConsumers>` XML element inside the enclosing `<adapter>…</adapter>` root XML element.

Each “auto” consumer flow is defined by adding a `<autoConsumer>…</autoConsumer>` XML element inside the enclosing `<autoConsumers>…</autoConsumers>` XML element.

Each `<autoConsumer>…</autoConsumer>` XML element must define all of the following mandatory attributes:

- consumerIdentifier: an identifier of the consumer of the flow
- targetAdapterName: this is the name of the Adapter producing the collection flow to consume from
- targetFlowName: this is the name of the collection flow to consume from (as per the definition of the producer collection flow on the target Adapter)
- messageConsumerClass: this is the name of the Java class (including the Java package name) implementing the flow consumer.
  For example: `com.example.MyMessageConsumer`. This class must extend the com.hp.umb.adapter.consumer.BaseConsumerMessageHandler class and has to implement the `com.hp.umb.adapter.consumer.ConsumerMessagesHandlerInterface<K extends Event>` Java interface (K being the type of message object to consume. K has to extend both the com.hp.uca.expert.event.Event and java.io.Serializable interfaces).

Please refer to section “`3.2.4.3 Defining the flow message consumer class`” for a full description of how to define a Message Consumer object.

The following optional attributes can be defined:

- messageConsumerTimeout (in milliseconds default 1000): this attribute is applicable only if the specified messageConsumerClass implements the `com.hp.umb.adapter.consumer.ConsumerMessagesHandlerInterface<K extends Event>`. It indicates the time in milliseconds to wait while no message arrives before returning the actual message set.
- messageConsumerMaxSetSize: this attribute is applicable only if the specified messageConsumerClass implements the `com.hp.umb.adapter.consumer.ConsumerMessagesHandlerInterface<K extends Event>`. It indicates the maximum size of the message set to return.
- manualCommit (optional default false): Specifies whether or not the commit of consumer offsets is done automatically by the UMB framework or left to the developer. If set to true, the default automatic offsets commit is disabled and it is the responsibility of the adapter developer to execute the commit through the methods of the BaseConsumerFlow class:

  - *commit(message)* – calling this method commits the partition and offset of the message given as parameter.
  - *commitAllPartitions()* – commit all consumed messages for all partitions

  Note: this feature is only available when the flow Consumer messageConsumerClass implements the `com.hp.umb.adapter.consumer.ConsumerMessagesHandlerInterface<K extends Event>` interface.

- **monitored (Boolean** default **true)**: a boolean flag to indicate whether the consumer flow is monitored by the UMB Framework (in which case the flag has to be set to **true**) or not (flag set to **false** in this case). By default, if this attribute is not present, the flag is assumed to be **true**, which means that the consumer flow is monitored. Monitored flows are attempted to be restarted automatically by the UMB Framework is they fail.
- **monitoringRestartPeriod (in milliseconds** default **30000)**: in case of Consumer Flow start failure, if the Flow is monitored (the default), this represents the restart attempt period.
- **readOffset:** this parameter defines the offset from which the consumer consumes messages in the Kafka server. It can be set to 'earliest' or 'latest'. The parameter is optional. When it is not set, the value 'earliest' is used.
  When the attribute **readOffset** is set to 'earliest', the consumer consumes messages from the very first ones i.e. all messages that the producer has pushed into Kafka.
  When the attribute **readOffset** is set to 'latest', the consumer consumes only messages starting from the latest ones i.e. only messages which the producer has sent to Kafka after the start of the consumer.
- **serializerClass:** this parameter can specify a custom serialization Class. The serialization class is the class in charge of serializing (de-serializing) the Event message into (and from) a byte array. The default serialization class is the UMB framework provided `com.hp.umb.adapter.internal.utilities.JavaClassSerializer` class which uses the standard java class serialization mechanism.

  A custom serialization class must implement the following legacy interfaces:

```
   kafka.serializer.Encoder.Encoder<Object>
   kafka.serializer.Decoder.Decoder<Object>
```

  or the new interfaces:

```
org.apache.kafka.common.serialization.Serializer<Object>
org.apache.kafka.common.serialization.Deserializer<Object>
```

  The default `JavaClassSerializer` implements all of them, allowing the use of old and new custom serializers.

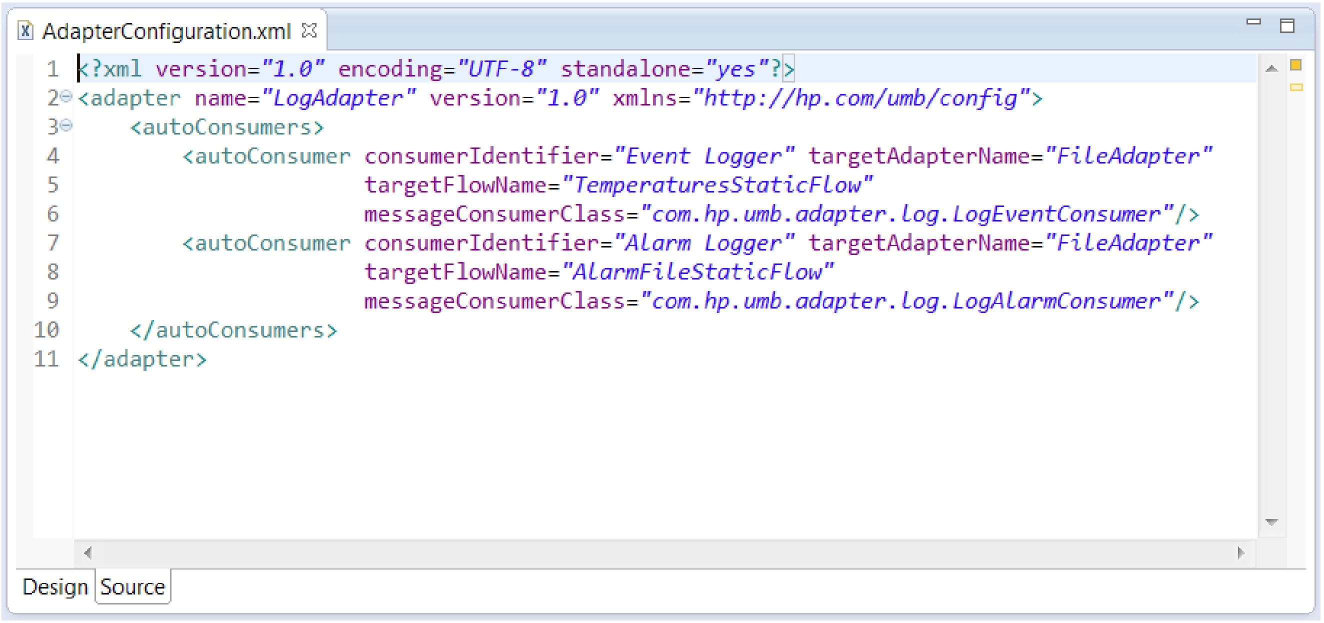
Each `<autoConsumer>…</autoConsumer>` XML element can also define parameters associated with the flow by adding the optional `<flowParameters>…</flowParameters>` XML element.

Inside the `<flowParameters>…</flowParameters>` XML element, each parameter is defined by a `<flowParameter>…</flowParameter>` XML element. Each parameter must define all of the following mandatory attributes:

- key: this is the name of the flow parameter
- value: this is the value of the flow parameter

The parameters defined in the `<flowParameters>…</flowParameters>` XML element will be used (alongside the properties of the flow defined in the target Adapter's `AdapterConfiguration.xml` file) when the flow is created.

Below is an example of an `AdapterConfiguration.xml` file that defines two "automatic" consumer flows:



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="LogAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <autoConsumers>
        <autoConsumer consumerIdentifier="Event Logger" targetAdapterName="FileAdapter"
                      targetFlowName="TemperaturesStaticFlow"
                      messageConsumerClass="com.hp.umb.adapter.log.LogEventConsumer"/>
        <autoConsumer consumerIdentifier="Alarm Logger" targetAdapterName="FileAdapter"
                      targetFlowName="AlarmFileStaticFlow"
                      messageConsumerClass="com.hp.umb.adapter.log.LogAlarmConsumer"/>
    </autoConsumers>
</adapter>
```

Figure 7 - Example of "auto" consumer flows in the AdapterConfiguration.xml file

You can find example of consumer flows that start automatically in the Log Adapter described in this document. For more information on the Log Adapter, please refer to chapter 5.3 "Log Adapter".

1.2.3.2.1 Defining Non-UMB consumer flows

The UMB framework offers the possibility to consume messages from Kafka topics where messages are not produced by an UMB adapter, but by any other Kafka producer.

In such a case, there is no Adapter providing the Flow service. The message source is therefore identified by the Topic name itself.

Such consumers can be defined in the <`autoConsumer`> section by using the <`autoNonUMBConsumer` … /> tag.

Each < `autoNonUMBConsumer` >…</ `autoNonUMBConsumer` > XML element must define all of the following mandatory attributes:

- consumerIdentifier: an identifier of the consumer of the flow
- topicName: this is the name of the kafka Topic from which the messages are retrieved.
- messageConsumerClass: this is the name of the Java class (including the Java package name) implementing the flow consumer. (Same definition as for `autoConsumers)`.

The optional attributes are the same as for the standard autoConsumers. No parameters can be defined.

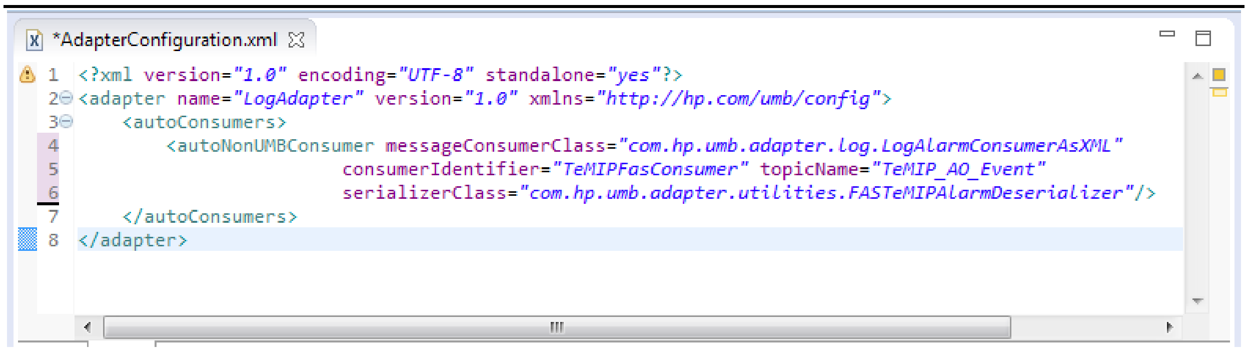
NOTE: When using non UMB flows, a specific attention must be paid to the deserialization process.

The format of the message pushed by the Kafka (nonUMB) producer is by definition application specific.

The nonUMBConsumer must therefore provide a Java Class that will be able to de-serialize the messages and turn them into Java objects.

Such de-serializer is specified by the serializerClass attribute.

Below is an example of an `AdapterConfiguration.xml` file that defines an "autoNonUMBConsumer" consumer:



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="LogAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <autoConsumers>
        <autoNonUMBConsumer messageConsumerClass="com.hp.umb.adapter.log.LogAlarmConsumerAsXML"
                            consumerIdentifier="TeMIPFasConsumer" topicName="TeMIP_AO_Event"
                            serializerClass="com.hp.umb.adapter.utilities.FASTeMIPAlarmDeserializer"/>
    </autoConsumers>
</adapter>
```

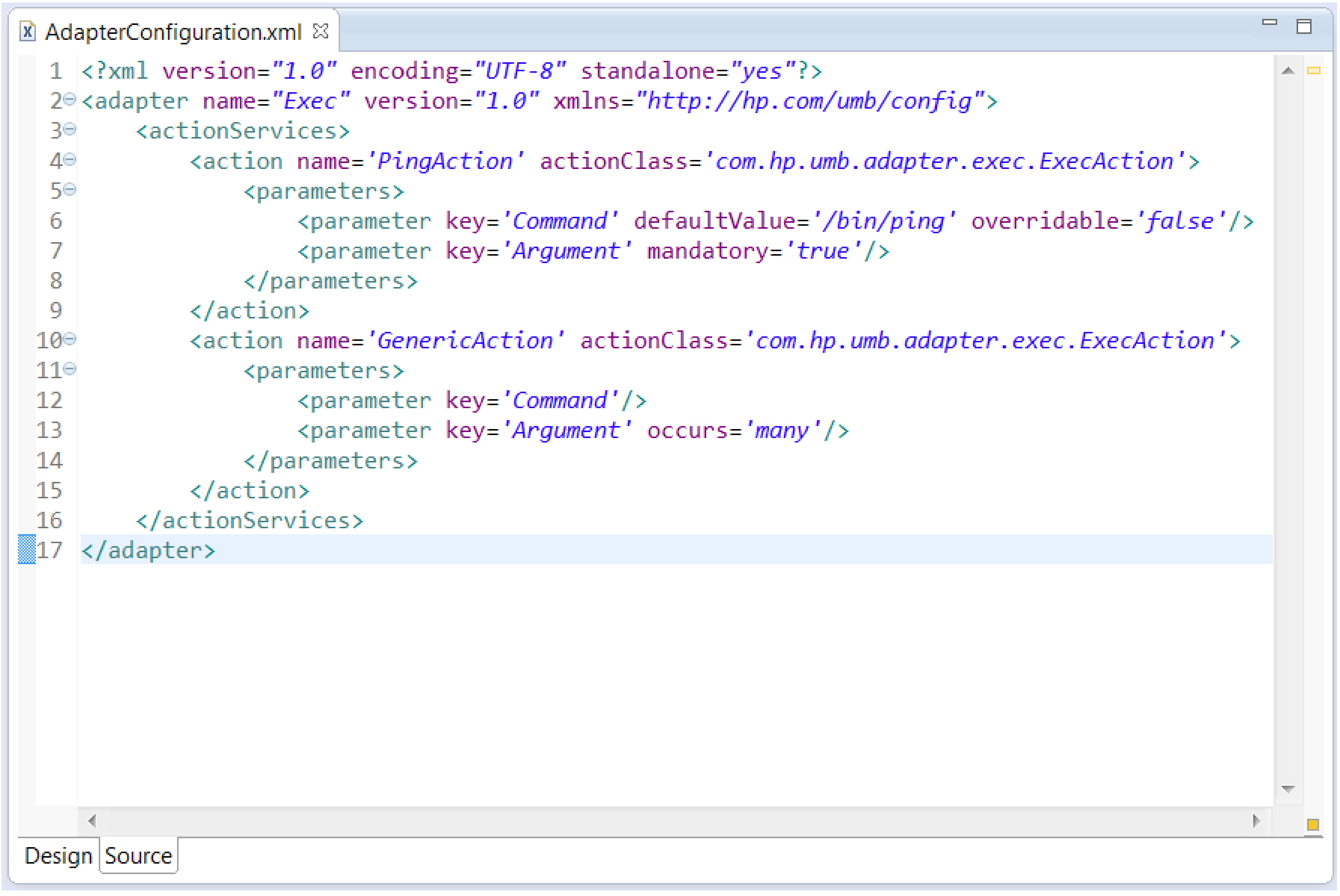
Figure 8 - Example of autoNonUmbConsumer flow definition in the AdapterConfiguration.xml file

1.2.3.3 Action Services

Action services are defined in the `<actionServices>` section of the `AdapterConfiguration.xml` file.

An action definition specifies an action that can be executed by the Adapter.

Example of an Action Service definition:



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="Exec" version="1.0" xmlns="http://hp.com/umb/config">
    <actionServices>
        <action name='PingAction' actionClass='com.hp.umb.adapter.exec.ExecAction'>
            <parameters>
                <parameter key='Command' defaultValue='/bin/ping' overridable='false'/>
                <parameter key='Argument' mandatory='true'/>
            </parameters>
        </action>
        <action name='GenericAction' actionClass='com.hp.umb.adapter.exec.ExecAction'>
            <parameters>
                <parameter key='Command'/>
                <parameter key='Argument' occurs='many'/>
            </parameters>
        </action>
    </actionServices>
</adapter>
```

Figure 9 - Example of an Action Service Definition

Action services are defined by the `<actionServices>…</actionServices>` XML element.

Each individual action service is defined by an `<action>…</action>` XML element inside the `<actionServices>…</actionServices>` XML element. There can be as many action services defined as needed.

An action service (or action) is identified by a 'name' (i.e. the identifier for the action) and an 'actionClass' (i.e. the Java class that will execute the action).

An `<action>…</action>` XML element can have the following attributes:

- name: The name of the action. This name will be referenced by UMB Adapters wishing to execute this action. It is mandatory to specify a value for the 'name' attribute.
- actionClass: The full name of the Java class that implements the action. It is mandatory to specify a value for the 'actionClass' attribute.
- inherits: The name of the action that the current action inherits from. This attribute is optional. If a child action inherits from a parent action, all the parameters defined in the parent action are implicitly also defined for the child action (See chapter 1.2.3.3.1 "Action Parameters" for more information on action parameters.).

  For example, it could be useful to use action inheritance if some parameters are common to several actions.

  It is optional to specify a value for the 'inherits' attribute.

Each Action can define a list of parameters using a `<parameters>…</parameters>` XML element inside an `<action>…</action>` XML element.

1.2.3.3.1 Action Parameters

The parameters are a list of configuration values (key/value pairs) that can be specified by the action service requester at the time of execution.

Each parameter is defined by a `<parameter>…</parameter>` XML element inside the `<parameters>…</parameters>` XML element. There can be as many parameters defined as needed.

A `<parameter>…</parameter>` XML element can have the following attributes:

- key: The 'key' attribute specifies the Parameter name. It is mandatory to specify a value for the 'key' attribute.
- defaultValue (optional): the 'defaultValue' attribute gives the Parameter a default value. In case this parameter is not specified by the requester, the default value is used.

  In the example above, the 'Command' Parameter of the 'PingAction' action is set with the defaultValue of '/bin/ping' which is the operating system command to execute.

  Doing so, the action requester does not have to specify this argument each time the 'PingAction' action is called.

- overridable (optional): the 'overridable' attribute is a boolean attribute. When set to 'false', the Action requester cannot override the parameter. When omitted the Parameter remains overridable (similar to overridable='true').

  This is particularly useful when the Action service developer wants to protect the parameter definition.

  Again in the example above, giving the possibility for the requester to override the 'Command' parameter would have no sense for an action called 'PingAction'.

- occurs (optional): the 'occurs' attribute can take the value 'once' or 'many'. By default, the same parameter can only be specified once by the requester. If the occurs='many' attribute is not set, specifying the same parameter more than once will lead to an action failure.
- mandatory (optional): the 'mandatory' attribute indicates this parameter must be specified by the requester. By default, parameters are not mandatory. If a mandatory parameter is not set for an action it will fail.

  With the PingAction action provided by the Exec Adapter, the 'Argument' parameter is mandatory and must be set by the requester with the IP Address of the host to ping or some other ping command-line option.

It is optional to specify a value for the 'mandatory' attribute.

- listValues (optional): the 'listValues' attribute indicates that this parameter can only have a value that is part of this comma-separated list. By default, all values are authorized.

1.2.4 The Unified Mediation Bus messages

Unified Mediation Bus messages can be any Java Objects with the following restrictions:

1. The message class must extend the `com.hp.uca.expert.event.DefaultEvent` Class.
2. The message class must implement the `java.io.Serializable` interface.

The Unified Mediation Bus framework uses the standard Java Serialization for serializing the message Objects at the time they are pushed to the Kafka server. In the same manner, the Objects are de-serialized when read from the Kafka server on the consumer side.

Note that the message Serialization/de-Serialization class can be changed in the Flow service definition.

1.2.4.1 Logging and testing considerations (UCA-EBC)

During the UCA-EBC value pack development phase, it may be very useful to collect samples of collected messages in order to replay them, or use them in the context of JUnit tests. This can be done by activating the UCA-EBC collector logging feature that will dump the collected messages using XML marshalling.

Refer to: UCA-EBC Admin, Configuration and Troubleshooting guide section "Collector Logging" for full explanation on how to activate UCA Collector logging.

For this reason, it is recommended that the Unified Mediation Bus message classes offer XML marshalling/un-marshalling capabilities based on JAXB.

One simple way to achieve that is to start from an XML schema and use the maven-jaxb2-plugin to produce a Java Class as shown in the example below.

Example of message schema:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:tns="http://hp.com/uca/expert/demo"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
    targetNamespace="http://hp.com/uca/expert/demo"
elementFormDefault="qualified" version="1.0"
    xmlns:inheritance="http://jaxb2-commons.dev.java.net/basic/inheritance"
jaxb:version="2.1"
    jaxb:extensionBindingPrefixes="xjc inheritance">

    <!-- FORCE ALL CLASSES IMPLEMENTS SERIALIZABLE -->
    <xs:annotation>
        <xs:appinfo>
            <jaxb:globalBindings generateIsSetMethod="true">
                <xjc:serializable uid="123456" />
            </jaxb:globalBindings>
        </xs:appinfo>
    </xs:annotation>

    <!-- -->
    <!-- ELEMENTS DEFINITION -->
    <!-- -->
    <xs:element name="temperature">
```

```
        <xs:complexType>
            <xs:annotation>
                <xs:appinfo>

<inheritance:extends>com.hp.uca.expert.event.DefaultEvent</inheritance:exte
nds>
                </xs:appinfo>
            </xs:annotation>

            <xs:sequence>
                <xs:element name="value" type="xs:double" minOccurs="1" />
            </xs:sequence>

        </xs:complexType>
    </xs:element>
</xs:schema>
```

Maven plugin configuration:

```
<plugin>
    <groupId>org.jvnet.jaxb2.maven2</groupId>
    <artifactId>maven-jaxb2-plugin</artifactId>
    <configuration>
        <schemaDirectory>src/main/resources/schemas</schemaDirectory>
        <extension>true</extension>
        <verbose>true</verbose>
        <forceRegenerate>true</forceRegenerate>
        <removeOldOutput>true</removeOldOutput>
        <args>
             <arg>-Xinheritance</arg>
        </args>
        <plugins>
             <plugin>
                 <groupId>org.jvnet.jaxb2_commons</groupId>
                 <artifactId>jaxb2-basics</artifactId>
                 <version>${jaxb2-basics.version}</version>
             </plugin>
        </plugins>
    </configuration>
    <executions>
        <execution>
             <id>Generate XML Marshallers</id>
             <phase>generate-sources</phase>
             <goals>
                  <goal>generate</goal>
             </goals>
        </execution>
    </executions>
</plugin>
```

Another approach is to directly add JAXB annotations to the Java Class as shown below:

```
package com.hp.uca.expert.demo;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;
import com.hp.uca.expert.event.DefaultEvent;


@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "value"
})
@XmlRootElement(name = "temperature")
public class Temperature
```

```
    extends DefaultEvent
    implements Serializable
{

    private final static long serialVersionUID = 123456L;
    protected double value;

    /**
     * Gets the value of the value property.
     *
     */
    public double getValue() {
        return value;
    }

    /**
     * Sets the value of the value property.
     *
     */
    public void setValue(double value) {
        this.value = value;
    }

    public boolean isSetValue() {
        return true;
    }

}
```

Chapter 2
Getting started with Unified Mediation Bus Development Kit

2.1 Installing the Unified Mediation Bus Adapter Development Kit

Detailed information on how to install UMB Adapter Development Kit is provided in the [R1] `HP Unified Mediation Bus– Installation and Configuration Guide`

2.2 Adapter Development Pre-requisites

2.2.1 Eclipse IDE

The UMB Adapter Development Kit has been designed for an easy integration with the Eclipse Integrated Development Environment (IDE) tool.

Before starting the development of any UMB value pack, it is necessary to download and install the Eclipse^{TM} application development environment.

The following table lists the Eclipse IDE pre-requisites for UMB Adapter Development Kit:

Software	Version
Eclipse IDE	4.4 (Luna) or higher

Table 2 - Eclipse IDE Prerequisites for UMB Adapter Development Kit

The minimum version of Eclipse IDE required by the UMB Development Kit is version 3.4 but we recommended Eclipse IDE version 4.4 (Luna) or higher.

If you want to install Eclipse IDE, please go to the following URL for downloading Eclipse IDE:
http://www.eclipse.org/downloads/

At the time of writing, the Eclipse IDE version is Neon 4.6.

We recommend you to download either (other choices may also be valid):

- Eclipse IDE for Java Developers, or
- Eclipse IDE for Java EE Developers

Then you need to choose to install either the 32-bit or 64-bit version of Eclipse IDE depending on whether you have a 32-bit or 64-bit operating system.

2.2.2 Post-install Environment Setup

2.2.2.1 The UMB_DEV_HOME Variable

The variable environment variable UMB_DEV_HOME is necessary for various development phases of a UMB Adapter, especially the build and packaging phases.

On Windows:

The Unified Mediation Bus Development Kit installation procedure adds the %UMB_DEV_HOME% environment variable to your user environment.

This variable is necessary for various development phases of a UMB Adapter development, especially the build and packaging phases.

To verify that this variable is correctly set after the UMB Adapter Development Kit has been installed, open a command-line (Run... -> cmd.exe) and type:

```
C:\> echo %UMB_DEV_HOME%
```

You should get an output similar to the following:

```
C:\UMB-DEV\
```

On Linux:

This Variable must be manually set in the user's environment, as specified in the [R1] Unified Mediation Bus Installation and Configuration Guide.

To verify that this variable is correctly set, please perform the following command:

```
$ echo ${UMB_DEV_HOME}
```

You should get an output similar to the following:

```
/opt/UMB-DEV
```

2.2.2.2 Maven Configuration

The UMB Adapter packaging is based on the use of the Apache Maven tool. If you don't have Apache Maven installed on your development system, please download it from maven.apache.org and follow the installation instructions.

The minimum required version of Apache Maven is 3.3.5

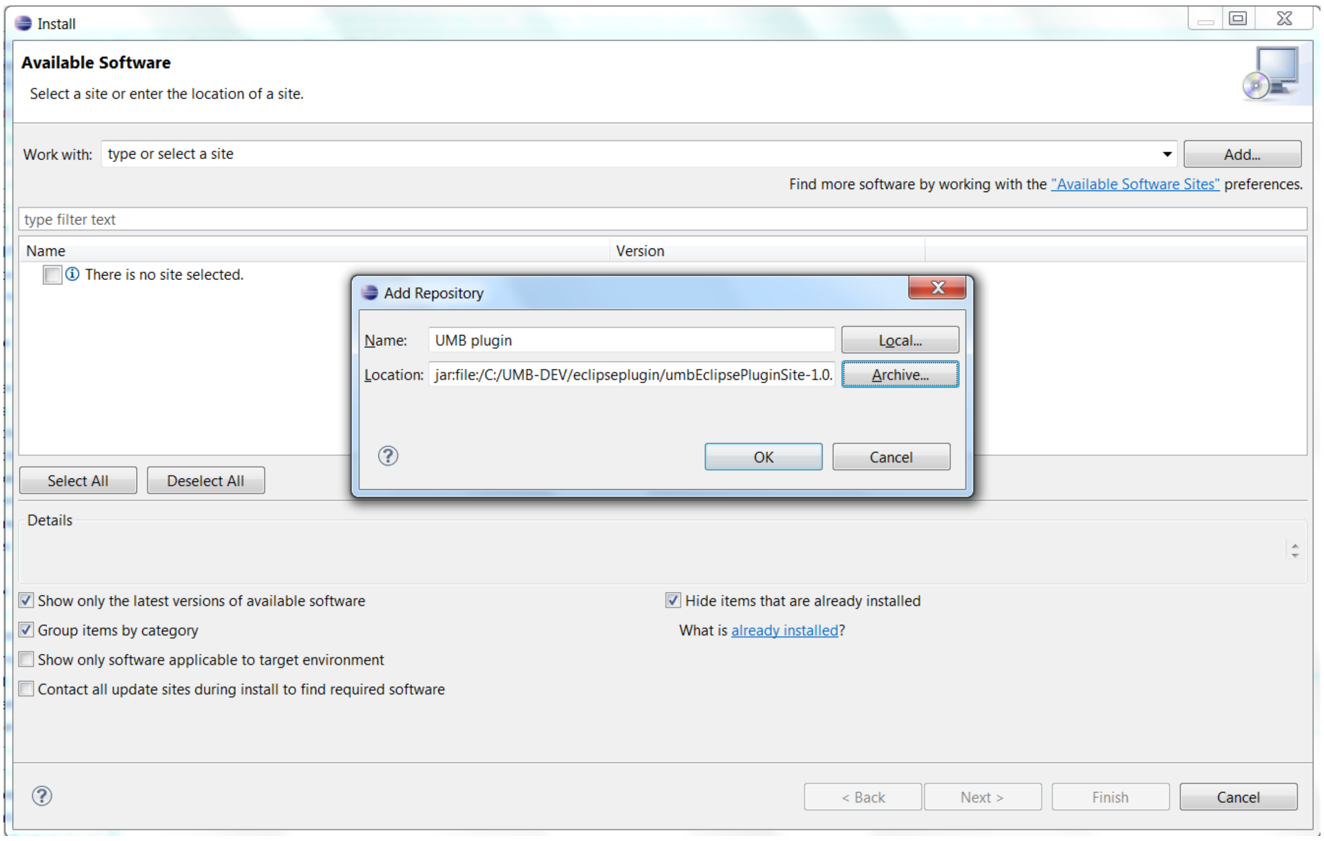
2.2.3 Unified Mediation Bus Eclipse plug-in installation instructions

The UMB Adapter Development Kit delivers an Eclipse plug-in that eases UMB project creation on Eclipse.

This plugin is delivered by the %UMB_DEV_HOME%\eclipseplugin\umbEclipsePluginSite-1.1.2-assembly.zip file.

The installation of this plug-in is done as follows:

- From the Eclipse 'Help' menu, choose 'Install new software' and then click on the Add... button.
- Select the Unified Mediation Bus eclipse plug-in ZIP file using the Archive... button and give it the name "UMB plugin" as shown in the picture below:


Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
There is no site selected.	

Add Repository

Name: UMB plugin

Location: jar:file:/C:/UMB-DEV/eclipseplugin/umbEclipsePluginSite-1.0.

Details

[✓] Show only the latest versions of available software

[✓] Group items by category

[ ] Show only software applicable to target environment

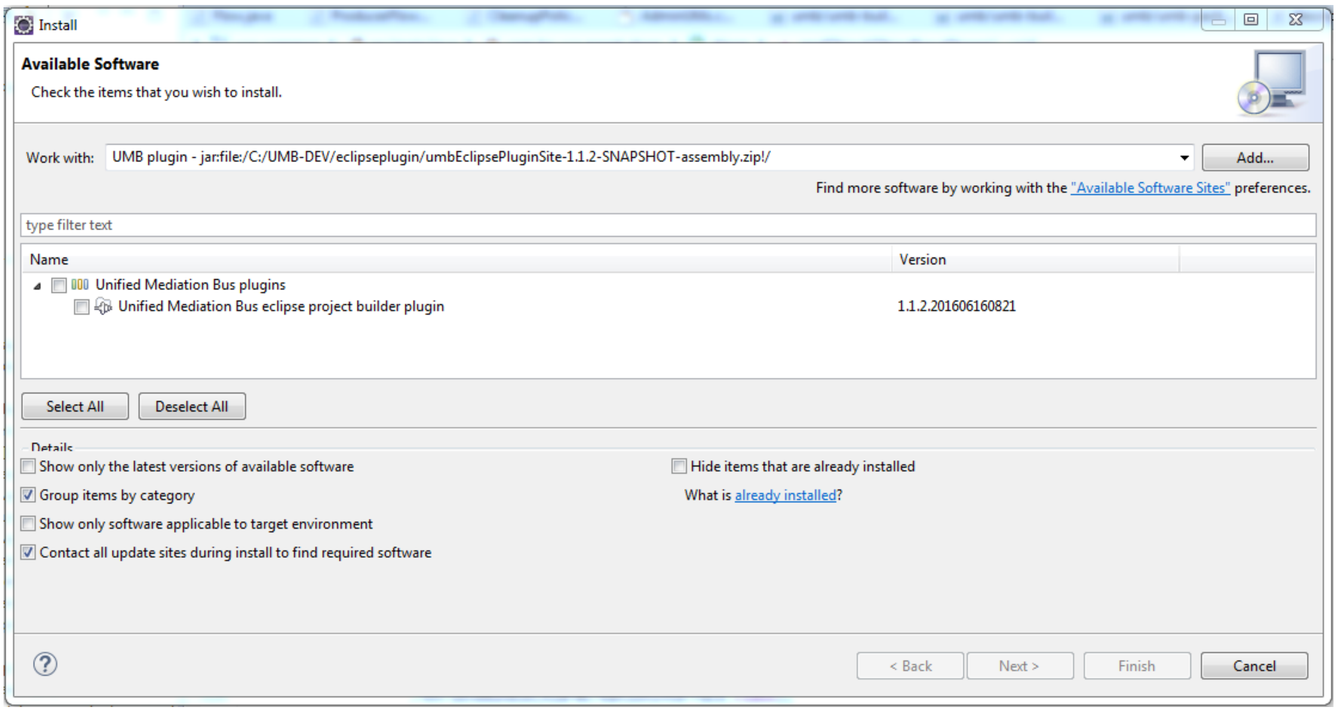
[ ] Contact all update sites during install to find required software

[✓] Hide items that are already installed

What is already installed?

Figure 10 - Unified Mediation Bus plug-in: Installation step 1

- Then click on the OK button. The screen should then display the archive content as follow:


Install

Available Software

Check the items that you wish to install.

Work with: UMB plugin - jar:file:/C:/UMB-DEV/eclipseplugin/umbEclipsePluginSite-1.1.2-SNAPSHOT-assembly.zip!/

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
[ ] Unified Mediation Bus plugins	
[ ] Unified Mediation Bus eclipse project builder plugin	1.1.2.201606160821

Details

[ ] Show only the latest versions of available software

[✓] Group items by category

[ ] Show only software applicable to target environment

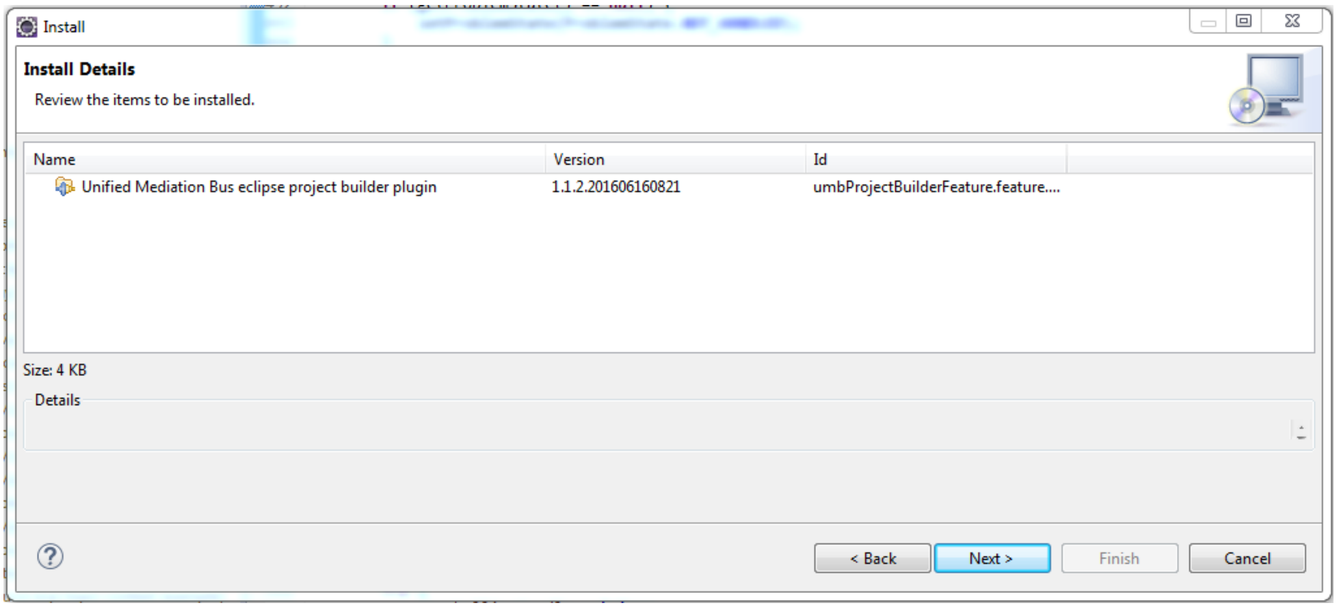
[✓] Contact all update sites during install to find required software

[ ] Hide items that are already installed

What is already installed?

Figure 11 - Unified Mediation Bus Eclipse plug-in: Installation step 2

- Check the “Unified Mediation Bus plugins” checkbox, uncheck the “Contact all update sites…”, and then click on the Next > button. The following screen is displayed:


Install

Install Details

Review the items to be installed.

Name	Version	Id
Unified Mediation Bus eclipse project builder plugin	1.1.2.201606160821	umbProjectBuilderFeature.feature....

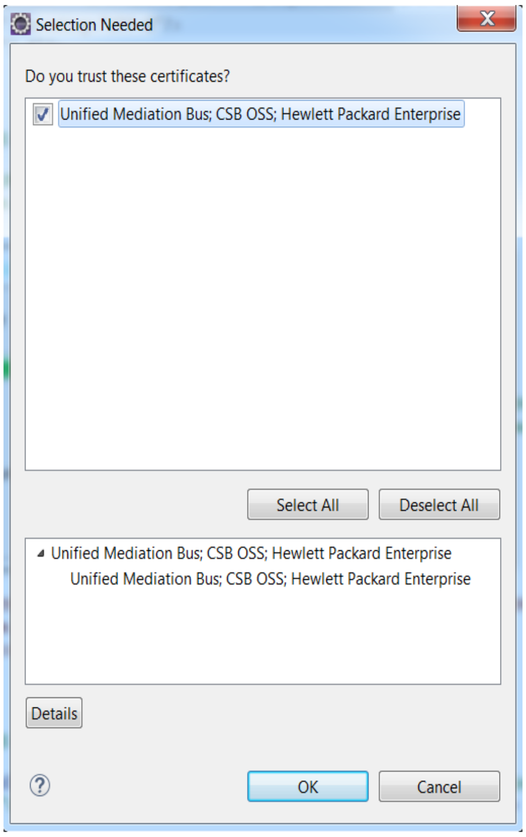
Size: 4 KB

Details

Figure 12 - Unified Mediation Bus Eclipse plug-in: Installation step 3

- Click on the Next > button for installing the plug-ins after accepting the license terms.

CAUTION: The following message appears during the installation. This is a normal message that is displayed because the UMB Eclipse Plugin is signed


Selection Needed

Do you trust these certificates?

☑ Unified Mediation Bus; CSB OSS; Hewlett Packard Enterprise

Unified Mediation Bus; CSB OSS; Hewlett Packard Enterprise
Unified Mediation Bus; CSB OSS; Hewlett Packard Enterprise

. Select the listed Certificates and Click OK to continue the installation

The plug-in installation requires a restart of your Eclipse IDE environment. Please restart Eclipse before any attempt to create a UMB Adapter project.

Chapter 3
Unified Mediation Bus Adapters development

3.1 Creating a new UMB Adapter

UMB Adapters provide connectivity between a target application and the UMB framework, and from there to other UMB Adapters and applications.

Each UMB Adapter can provide:

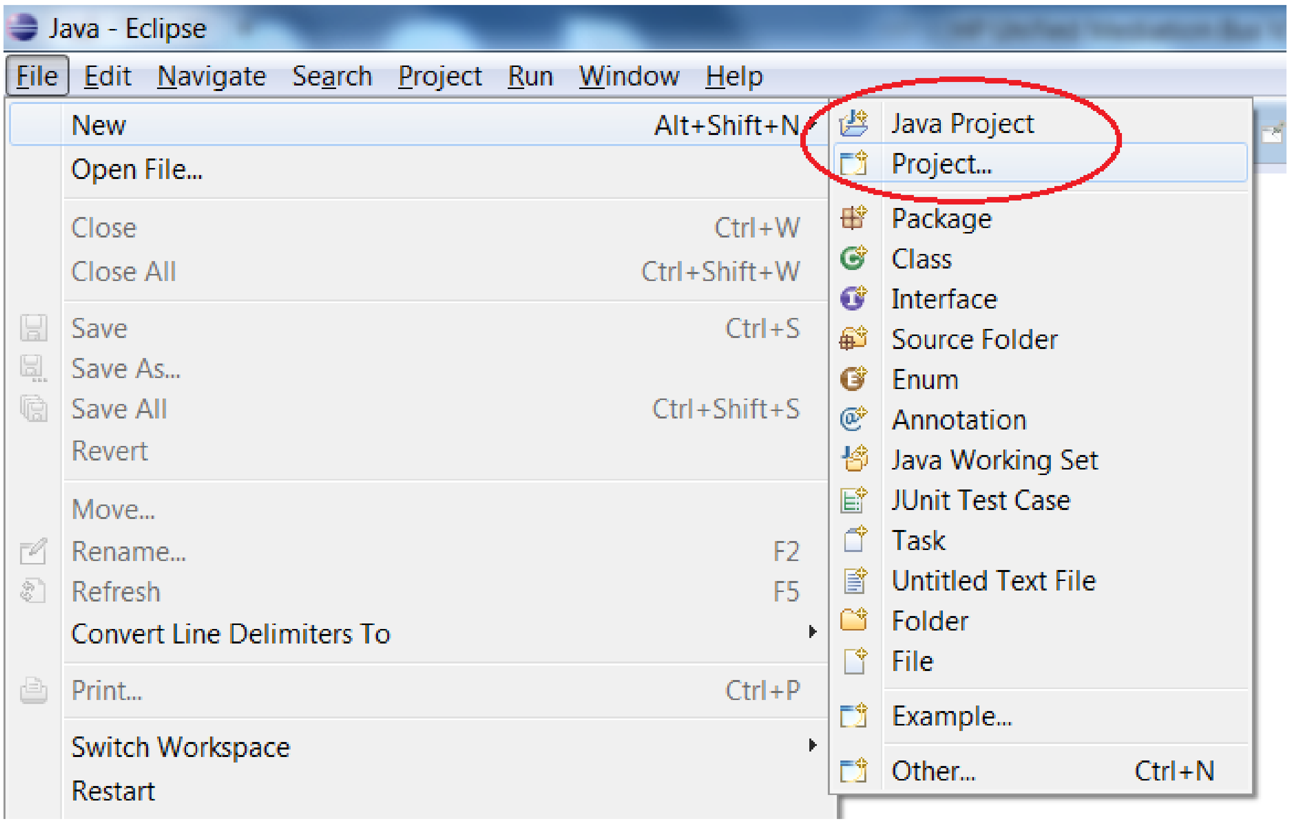
- Flow collection services
- Action services

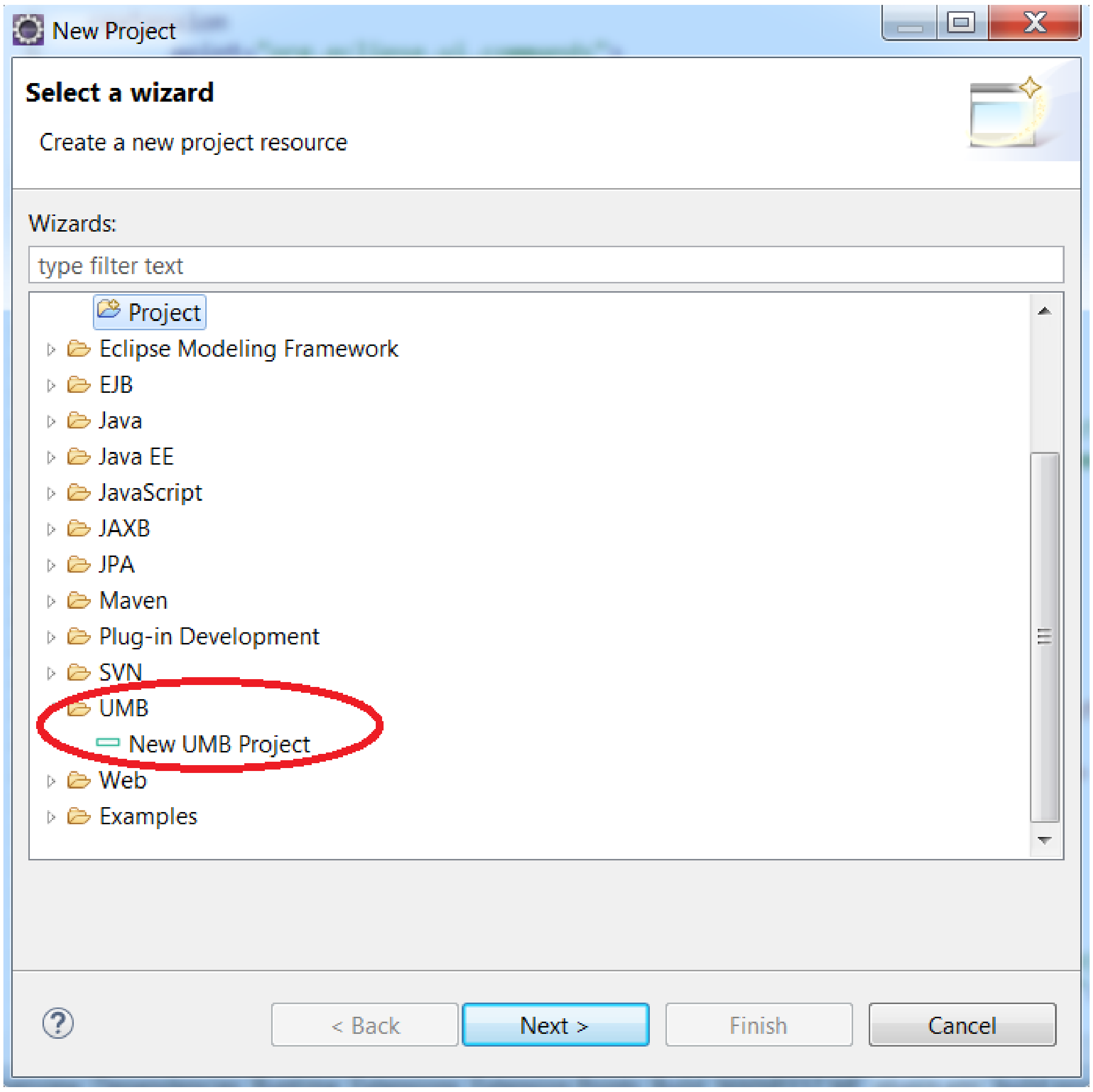
Each UMB Adapter can also act as a consumer of other UMB Adapters flow or action services.

3.1.1 Creating a UMB Adapter project within Eclipse

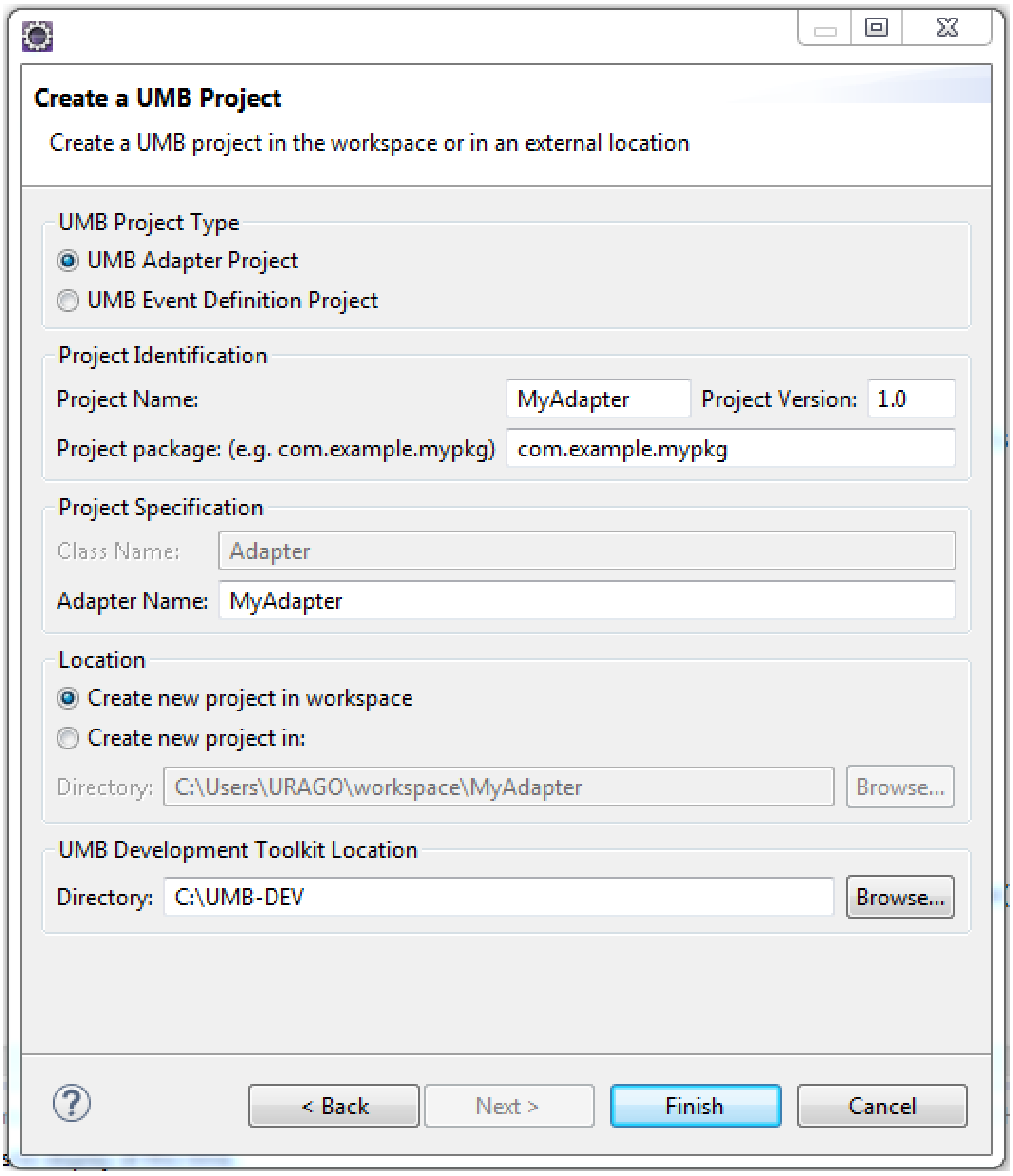
The UMB eclipse plug-in brings a project creation wizard that allows for the creation of a new UMB Adapter project in just a few clicks. The plug-in also allows for the creation of a new UMB Event Definition project (see 4.1 for details).

This plug-in can be launched from the Eclipse menus by selecting File -> New Project:





This launches the UMB Project wizard:


Create a UMB Project

Create a UMB project in the workspace or in an external location

UMB Project Type

[✓ UMB Adapter Project]

UMB Event Definition Project

Project Identification

Project Name: MyAdapter

Project Version: 1.0

Project package: (e.g. com.example.mypkg) com.example.mypkg

Project Specification

Class Name: Adapter

Adapter Name: MyAdapter

Location

[✓ Create new project in workspace]

Create new project in:

Directory: C:\Users\URAGO\workspace\MyAdapter

UMB Development Toolkit Location

Directory: C:\UMB-DEV

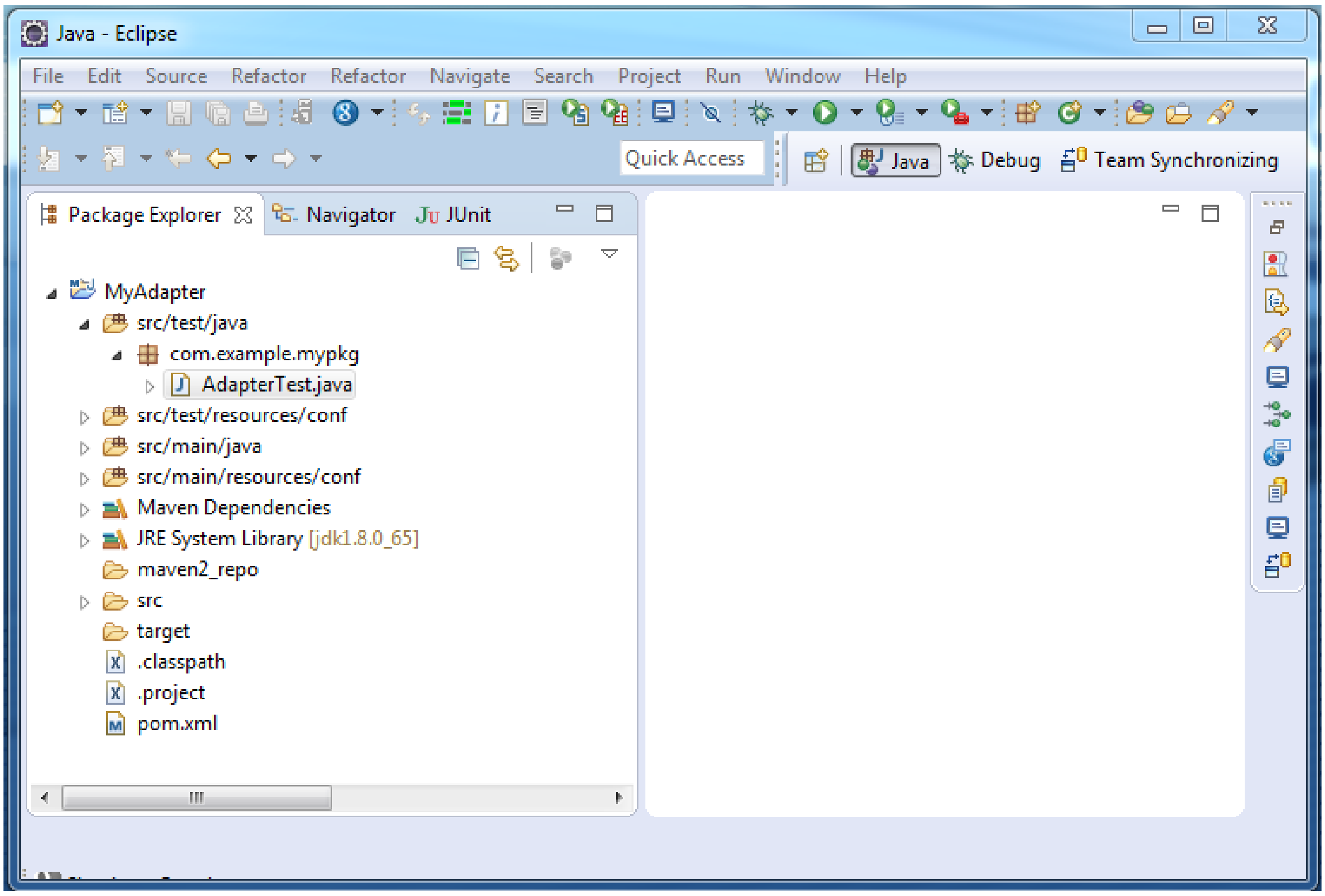
Figure 13 - UMB project creation wizard Step1

From the UMB project wizard window, you can specify information regarding your UMB project. It is possible to create two kinds of UMB Projects: UMB Adapter Project or UMB Event Definition Project (see 4.1 for details). Therefore, the type of UMB project has to be selected first before going further. If 'UMB Adapter project' is selected, the following parameters have to be specified:

- Project type: the type of UMB Adapter project to create
- Project name: the name of the UMB Eclipse Adapter project to create
- Project version: the version of the UMB Adapter project which is also the adapter version.
- Project package: the name of the Java package to be used for the Java classes of the UMB Adapter
- Class Name: a UMB Adapter class name is given by default and its value is "Adapter". The class name is used in the adapter's `startup.conf` configuration file.
- Adapter Name: The name of the UMB Adapter is used to identify the Adapter on the UMB framework
- Project location: the location of the UMB Adapter Eclipse project on the file system, either in the Eclipse workspace or anywhere on the file system

- **UMB Development Toolkit location**: location on the file system of the installation directory of the UMB Adapter Development Kit[2]. The default value is the value of the `%UMB_DEV_HOME%` environment variable on Windows systems (`${UCA_EBC_DEV_HOME}` on Linux): `C:\UMB-DEV` by default on Windows systems (`/opt/UMB-DEV` on Linux)

When you click on the Finish button, your UMB Adapter Eclipse project is created. It already has a Maven nature and also a minimum set of configuration, Java and JUnit files. It can be successfully compiled and unit tested.



```
MyAdapter
  src/test/java
    com.example.mypkg
      AdapterTest.java
  src/test/resources/conf
  src/main/java
  src/main/resources/conf
  Maven Dependencies
  JRE System Library [jdk1.8.0_65]
  maven2_repo
  src
  target
  .classpath
  .project
  pom.xml
```

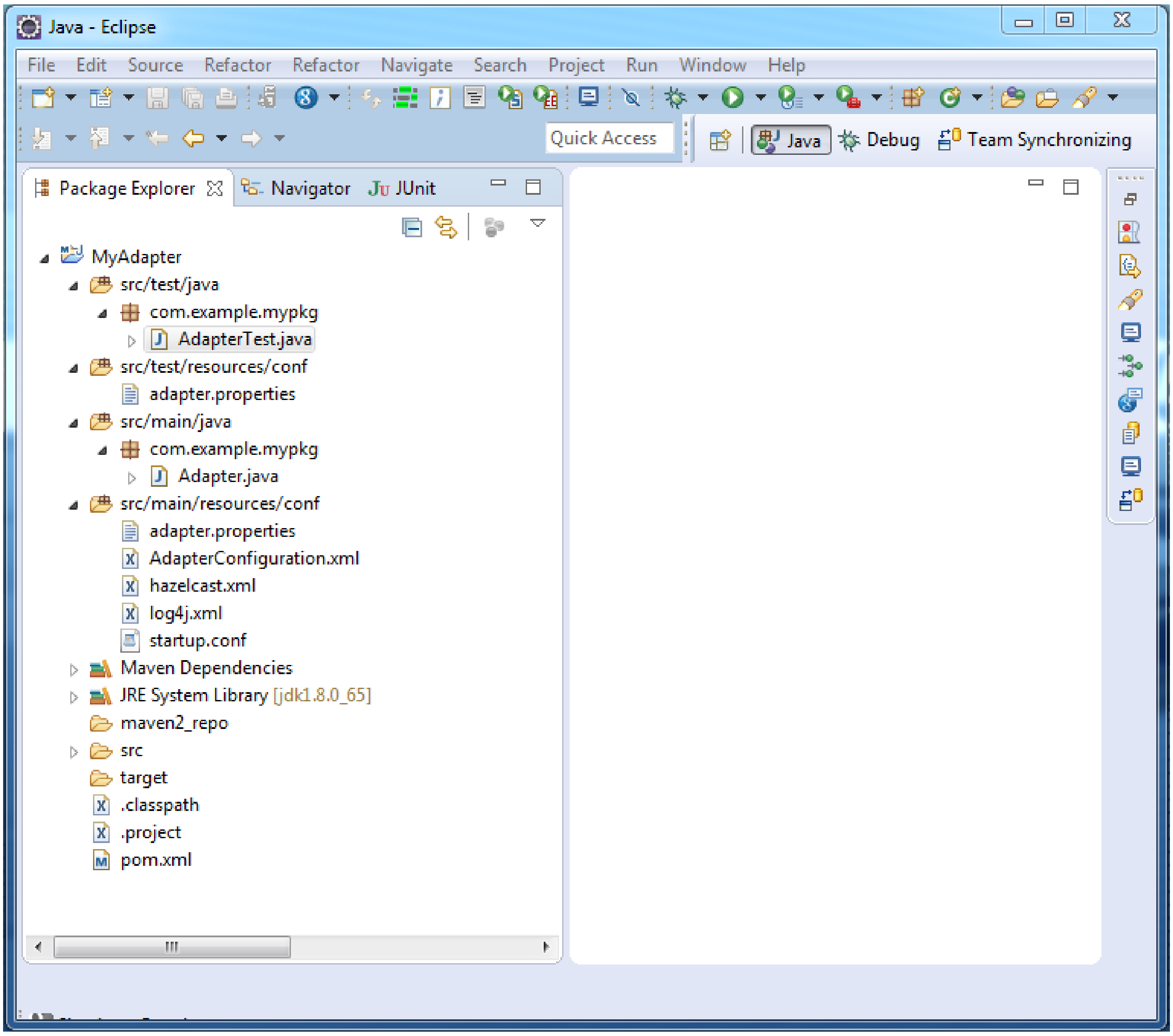
Figure 14 – New UMB Adapter project

3.1.2 Anatomy of the created project

Using Eclipse IDE, you can browse through the different directories that compose the newly created UMB Adapter project.

The structure of the created UMB Adapter project is as follows:

[2] Please refer to [R1] *Unified Mediation Bus installation and configuration Guide* for more information on how to install the UMB Adapter Development Kit



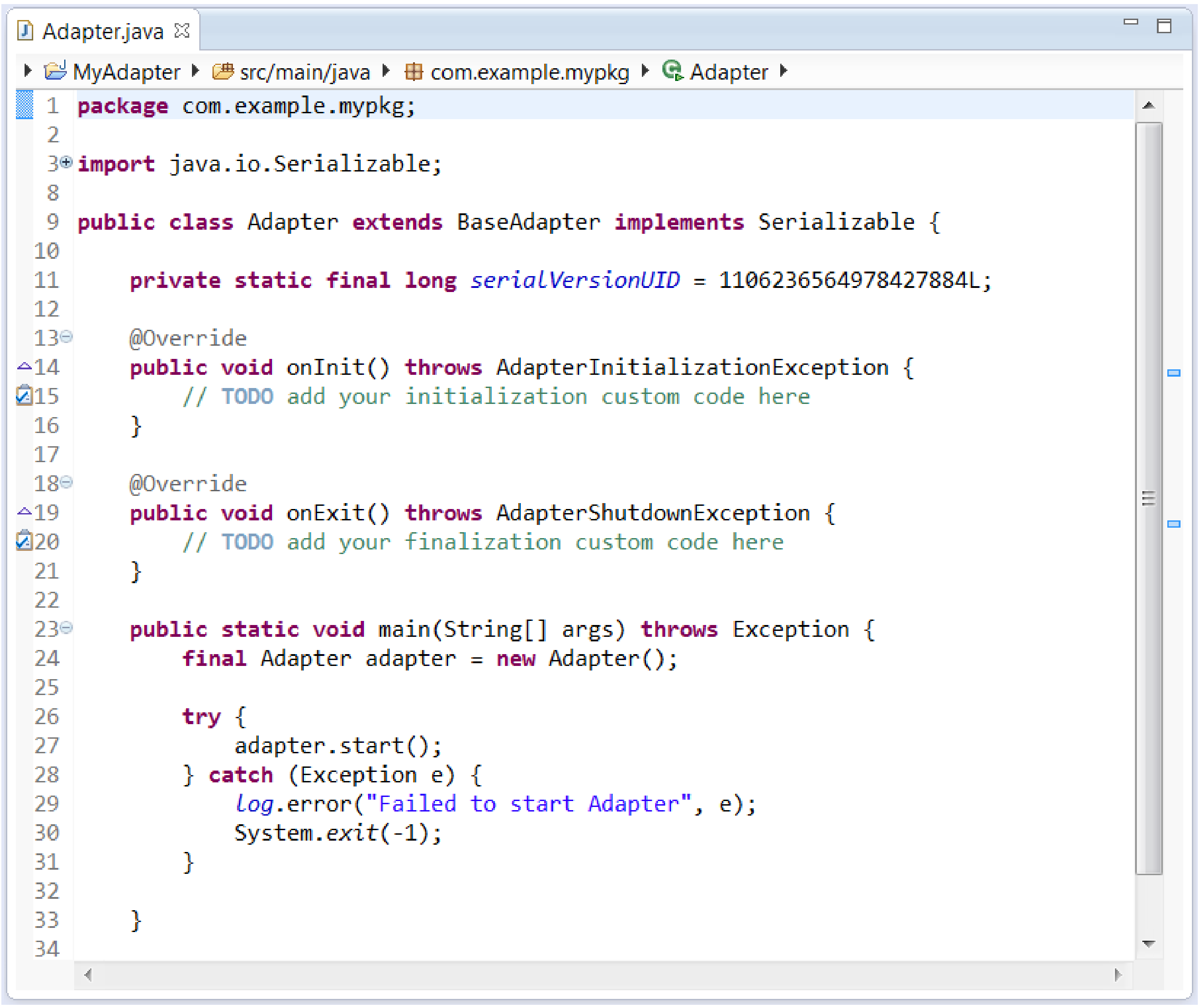
```
MyAdapter
    src/test/java
        com.example.mypkg
            AdapterTest.java
    src/test/resources/conf
        adapter.properties
    src/main/java
        com.example.mypkg
            Adapter.java
    src/main/resources/conf
        adapter.properties
        AdapterConfiguration.xml
        hazelcast.xml
        log4j.xml
        startup.conf
    Maven Dependencies
    JRE System Library [jdk1.8.0_65]
    maven2_repo
    src
    target
    .classpath
    .project
    pom.xml
```

Figure 15 - Folder structure of the new UMB Adapter project

The configuration files of the Adapter are located in the `src/main/resources/conf` folder:

- The `adapter.properties` file defines properties for the adapter including connection information for the UMB Kafka/ZooKeeper instance(s)
- The `AdapterConfiguration.xml` file defines the flow and action services provided by the adapter as well as "automatic" consumer flows
- The `hazelcast.xml` file defines how to connect to the UMB Hazelcast instance(s)
- The `log4j.xml` file defines the Adapter's Log4j configuration

The Adapter Java classes that define the behavior of the Adapter are located in the `src/main/java` folder. As the Adapter has just been created, there's only one Java class present: the `Adapter.java` class. An object of this class represents an instance of the Adapter. By default, this class also has a `main(String[] args)` method that creates one instance of the Adapter and starts it.



```java
package com.example.mypkg;

import java.io.Serializable;

public class Adapter extends BaseAdapter implements Serializable {

    private static final long serialVersionUID = 1106236564978427884L;

    @Override
    public void onInit() throws AdapterInitializationException {
        // TODO add your initialization custom code here
    }

    @Override
    public void onExit() throws AdapterShutdownException {
        // TODO add your finalization custom code here
    }

    public static void main(String[] args) throws Exception {
        final Adapter adapter = new Adapter();

        try {
            adapter.start();
        } catch (Exception e) {
            log.error("Failed to start Adapter", e);
            System.exit(-1);
        }

    }

```

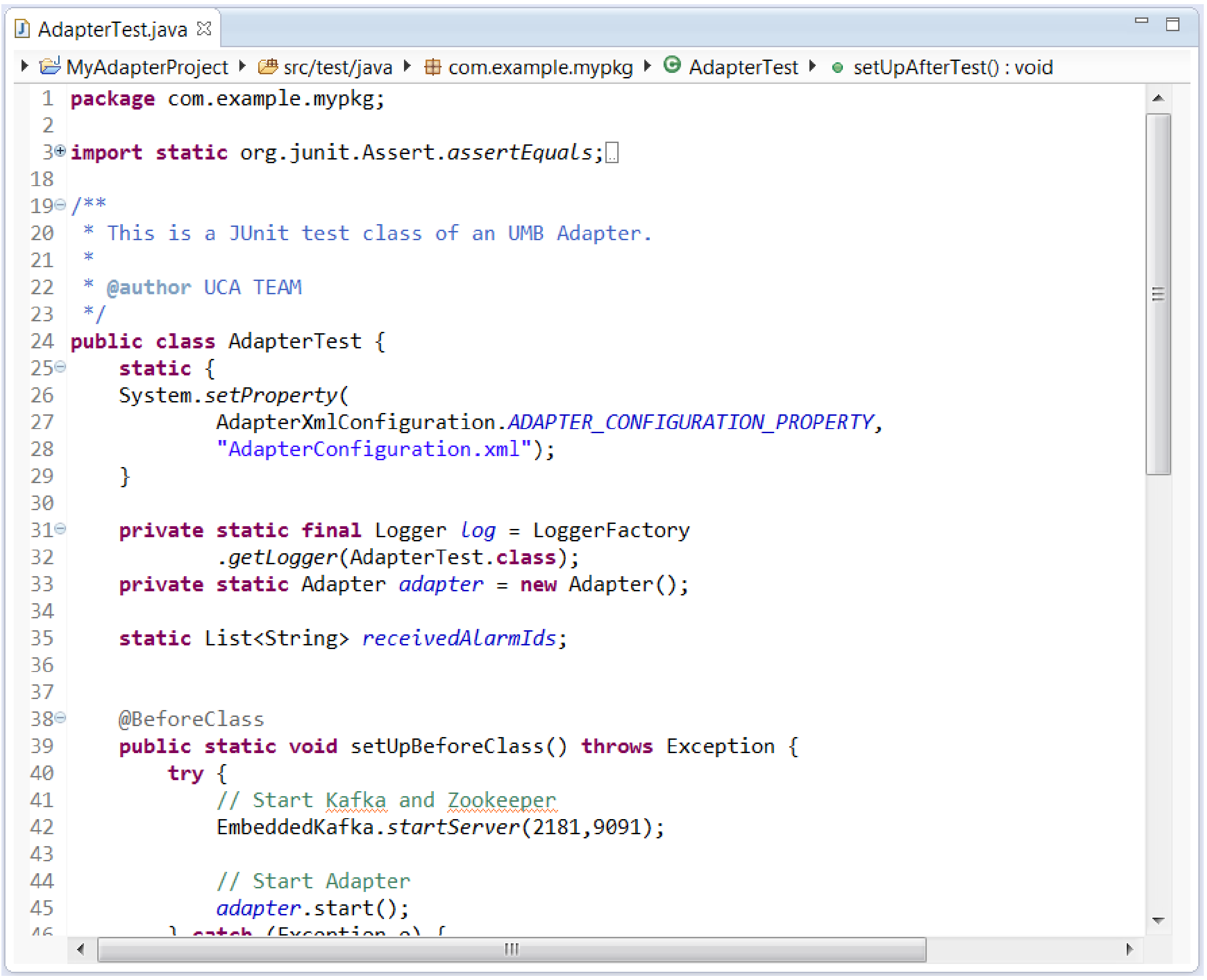
Figure 16 - Adapter.java Java class of the new UMB Adapter project

The created UMB Adapter project also comes with an Apache Maven `pom.xml` file that is used for building and packaging the UMB Adapter outside of the Eclipse IDE.

3.1.3 Validation of the created project

The Adapter's `src/test/java` folder contains a Junit test Class named `AdapterTest.java`. This is a simple test that simply starts the Adapter and checks it is in the 'RUNNING' state.

This Test class is a template that can be extended to test the Adapter's capabilities (flow services and action services).


AdapterTest.java
MyAdapterProject ▸ src/test/java ▸ com.example.mypkg ▸ AdapterTest ▸ setUpAfterTest() : void

```java
package com.example.mypkg;

import static org.junit.Assert.assertEquals;
/**
 * This is a JUnit test class of an UMB Adapter.
 *
 * @author UCA TEAM
 */
public class AdapterTest {
    static {
    System.setProperty(
            AdapterXmlConfiguration.ADAPTER_CONFIGURATION_PROPERTY,
            "AdapterConfiguration.xml");
    }

    private static final Logger log = LoggerFactory
            .getLogger(AdapterTest.class);
    private static Adapter adapter = new Adapter();

    static List<String> receivedAlarmIds;


    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        try {
            // Start Kafka and Zookeeper
            EmbeddedKafka.startServer(2181,9091);

            // Start Adapter
            adapter.start();
```

Figure 17 - AdapterTest.java JUnit test class of the new UMB Adapter project

3.2 Customizing the created UMB Adapter project

The project generated by the UMB Eclipse plug-in provides an “empty shell” Adapter that does not provide any collection flow or action services and that does not consume any collection flows.

This is basically a class that extends the `com.hp.umb.adapter.BaseAdapter` class. It then must implement:

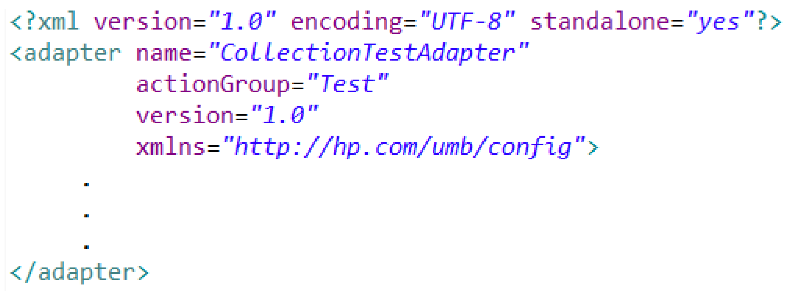
- Production flow services (if this is an adapter producing Events)
- Consumer flows (if this is an adapter consuming Events)
- Actions services (if this adapter is an action service provider)

The following chapters will explain how to turn the Adapter into an Adapter that does these things. For this you have to customize:

- The Adapter configuration files, mostly the `AdapterConfiguration.xml` file
- The Adapter Java files
- The Adapter JUnit files

3.2.1 Customizing the Adapter Name

Each Adapter that is part of an UMB Mediation solution must have a unique Name. This name is defined in the `AdapterConfiguration.xml` file by setting the `name` attribute of the `<adapter>…</adapter>` XML element:



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="CollectionTestAdapter"
         actionGroup="Test"
         version="1.0"
         xmlns="http://hp.com/umb/config">
     .
     .
     .
</adapter>
```

Figure 18 - Customizing the Adapter Name

Additionally an actionGroup name (used for load-balancing action execution on a group of adapters) and version number can be defined for the adapter. The version number does not play any role in the adapter's identification however.

3.2.2 Adding producer flow services

In order to add producer collection flow services to your Adapter, you first have to define the collection flows or types of collection flows that you want to provide in the `AdapterConfiguration.xml` file.

In this file, producer collection flow services are defined by adding the `<flowServices>…</flowServices>` XML element inside the enclosing `<adapter>…</adapter>` root XML element.

Each producer collection flow service is defined by adding a `<flow>…</flow>` XML element inside the enclosing `<flowServices>…</flowServices>` XML element.

Each `<flow>…</flow>` XML element must define all of the following mandatory attributes.

Refer to section "1.2.3.1 Flow Services" in this documents for a full description of the flow service attributes.

Each `<flow>…</flow>` XML element can also define parameters associated with the flow by adding the optional `<parameters>…</parameters>` XML element. Inside the `<parameters>…</parameters>` XML element, each parameter is defined by a `<parameter>…</parameter>` XML element.

Refer to section "1.2.3.1.1Flow Parameters" in this documents for a full description of the flow service parameter attribute definitions.

The parameters defined in the `<parameters>…</parameters>` XML section are positioned by a flow consumer when the flow is started.

Below is an example of an `AdapterConfiguration.xml` file that defines one static flow named `AlarmFileStaticFlow`:

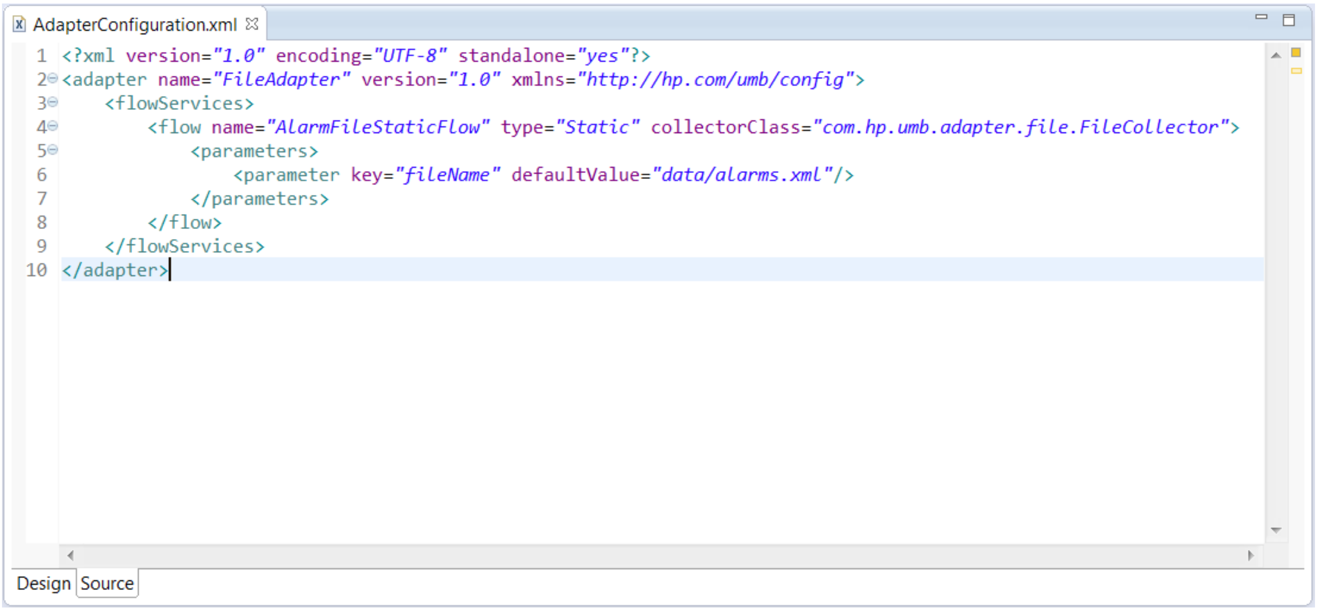

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="FileAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <flowServices>
        <flow name="AlarmFileStaticFlow" type="Static" collectorClass="com.hp.umb.adapter.file.FileCollector">
            <parameters>
                <parameter key="fileName" defaultValue="data/alarms.xml"/>
            </parameters>
        </flow>
    </flowServices>
</adapter>
```

Figure 19 - Example of a flow in the AdapterConfiguration.xml file

Once the producer collection flow services have been defined in the `AdapterConfiguration.xml` file, it is necessary to create a “Collector” Java class for each producer collection flow. The name of the class has to match the value of the collectorClass attribute of the `<flow>`...`</flow>` XML element in the `AdapterConfiguration.xml` file. It is mandatory that this class extends the `com.hp.umb.adapter.collector.BaseCollector` class.

You can create a “Collector” Java class by using the context menus in your Adapter project:

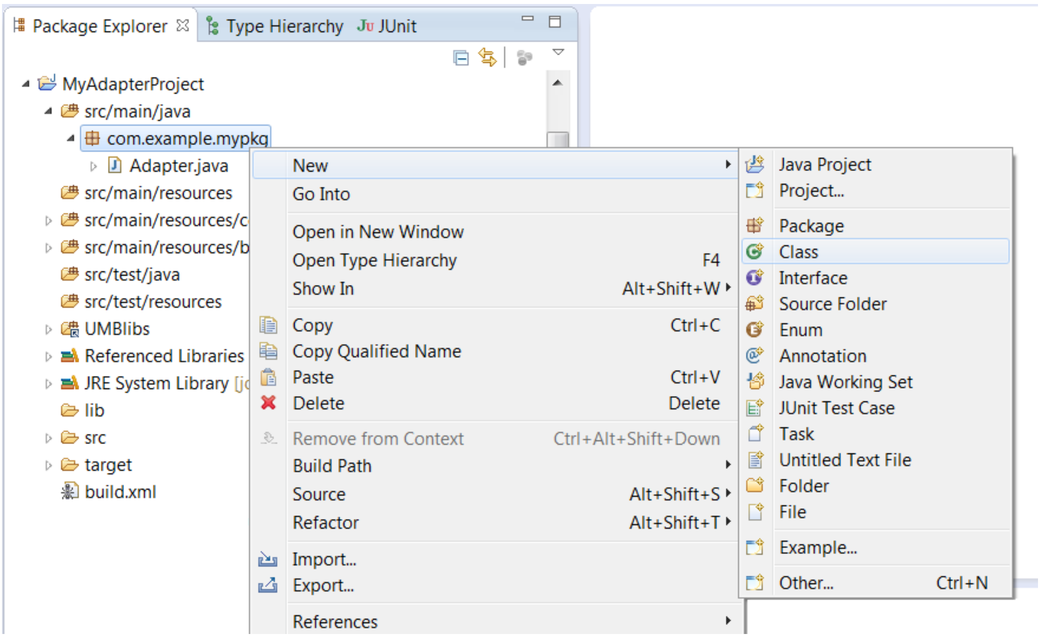
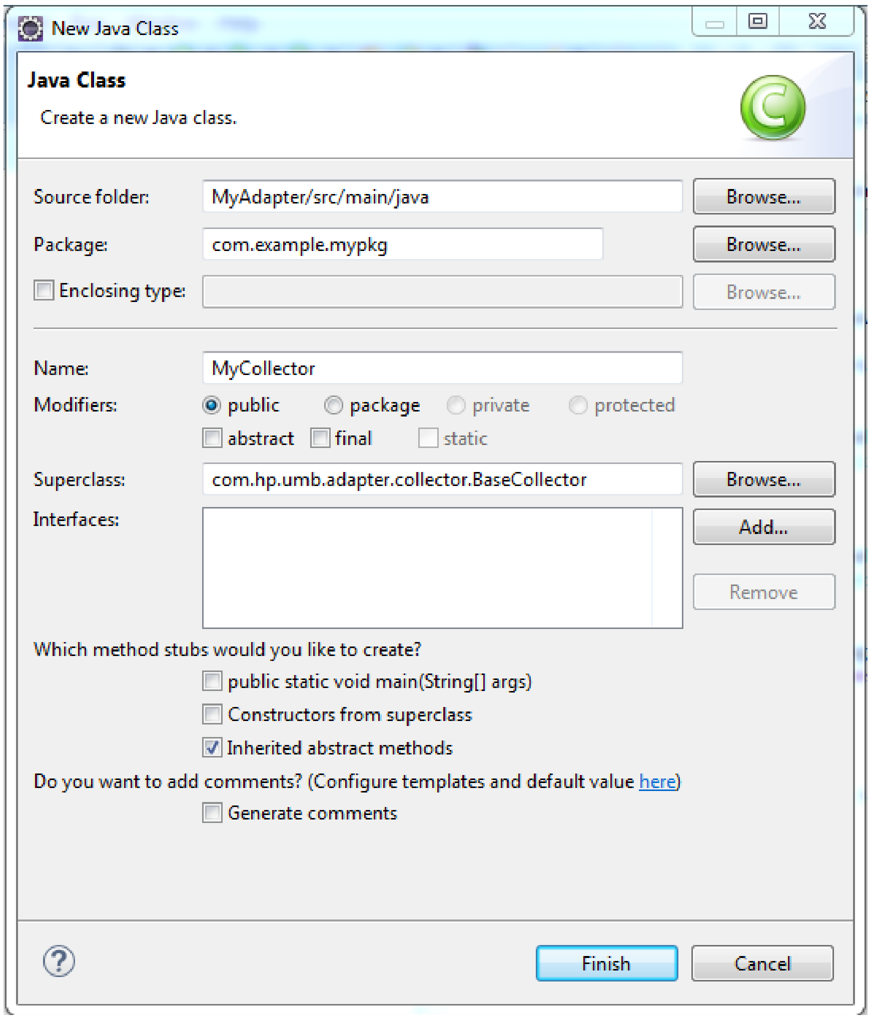


Figure 20 - Creating a “Collector” Java class – Step 1

This opens the New Class window.


New Java Class

Java Class

Create a new Java class.

Source folder: MyAdapter/src/main/java

Package: com.example.mypkg

☐ Enclosing type:

Name: MyCollector

Modifiers: [✓ public] package private protected

☐ abstract ☐ final ☐ static

Superclass: com.hp.umb.adapter.collector.BaseCollector

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

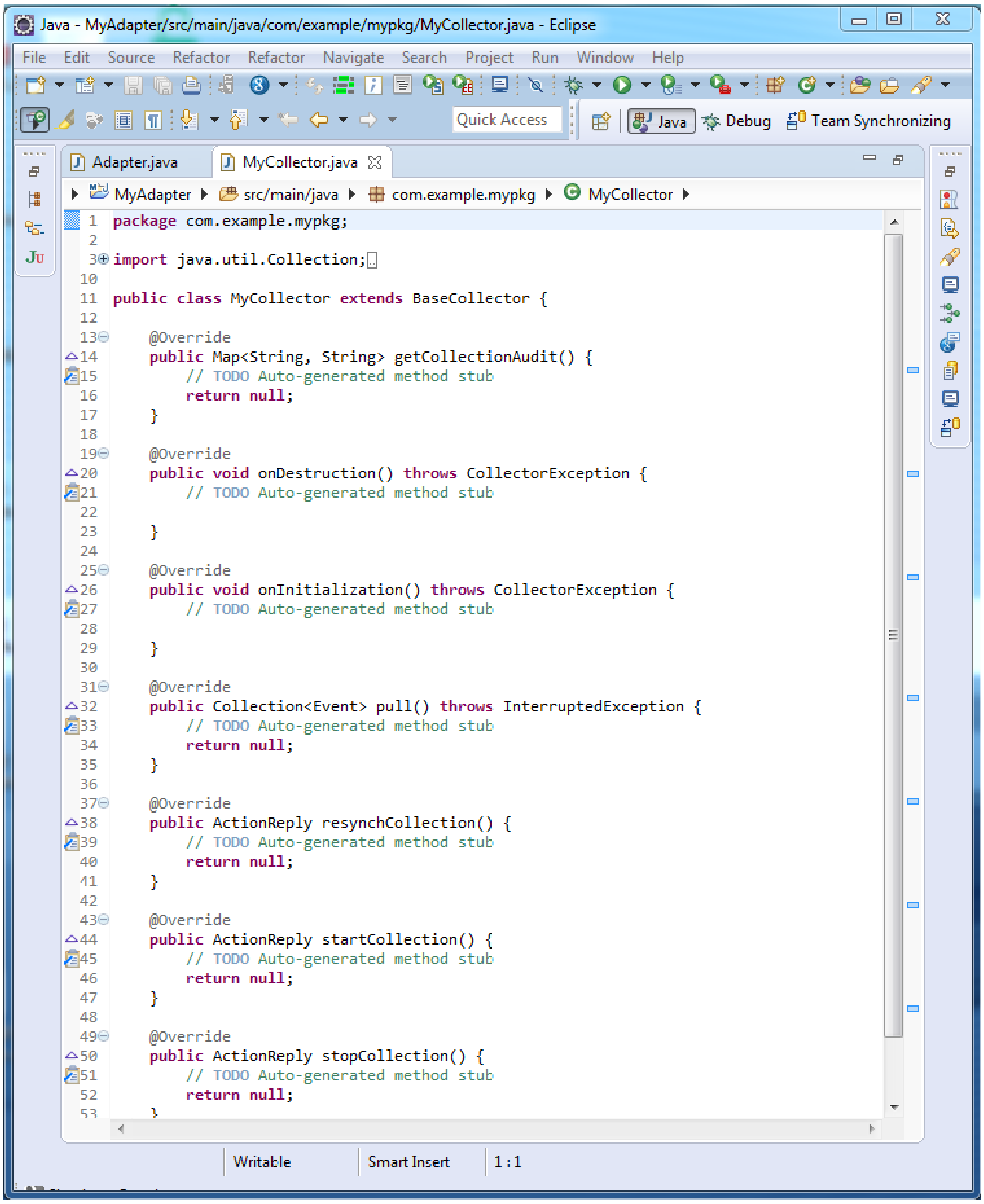
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Figure 21 - Creating a "Collector" Java Class Step 2

Please make sure that the new "Collector" Java class that you're creating extends the `com.hp.umb.adapter.collector.BaseCollector.` Eclipse will automatically create the methods from the `com.hp.umb.adapter.collector.CollectorInterface` Java interface that need to be overriden.

Once you click on the `Finish` button, the new "Collector" Java class is created as shown below:


Java - MyAdapter/src/main/java/com/example/mypkg/MyCollector.java - Eclipse
File Edit Source Refactor Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
Team Synchronizing
Adapter.java
MyCollector.java
MyAdapter
src/main/java
com.example.mypkg
MyCollector
Writable
Smart Insert
1 : 1

```java
package com.example.mypkg;

import java.util.Collection;

public class MyCollector extends BaseCollector {

    @Override
    public Map<String, String> getCollectionAudit() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void onDestruction() throws CollectorException {
        // TODO Auto-generated method stub

    }

    @Override
    public void onInitialization() throws CollectorException {
        // TODO Auto-generated method stub

    }

    @Override
    public Collection<Event> pull() throws InterruptedException {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public ActionReply resynchCollection() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public ActionReply startCollection() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public ActionReply stopCollection() {
        // TODO Auto-generated method stub
        return null;
    }
```

Figure 22 - Creating a "Collector" Java class – Step 3

In order to finalize the new "Collector" Java class, several methods need to be implemented:

- Initialization and destruction methods:
  - `onInitialization(…)`: This method initializes the creation of the collector. It is automatically called by the UMB framework when the collector is created. If you need to initialize objects or resources associated with the "Collector" object, this is the place to do it.
  - `onDestruction()`: This method finalizes the destruction of the collector. It is automatically called by the UMB framework when the collector is destroyed. If you need to free objects or resources associated with the "Collector" object, this is the place to do it.
- Collection flow action methods:

- `startCollection()`: This method starts the Collection flow. It is called by the UMB framework when a `startCollection()` request is made on the ConsumerFlow side. This method returns an `ActionReply` object to indicate whether the start of the collection flow was successful or not.
- `resynchCollection()`: This method re-synchronizes the Collection flow. It is called by the UMB framework when a `resyncCollection()` request is made on the ConsumerFlow side. This method returns an `ActionReply` object to indicate whether the re-synchronization of the collection flow was successful or not.

  The re-synchronization mechanism requires that the Event Provider offers a re-synchronization feature. This is the case for TeMIP for example, that stores alarms and then is capable of re-sending all alarms at any given point in time, but will be true for any event provider that stores its event and is able to send them back again on request.

  A resynchronized flow of events must be surrounded by two additional events: one indicating the beginning of the synchronization flow, and another one indicating the end of the resynchronization flow.

  Such events must not contain any other information than the start and end of synchronization flags.

  The Begin of Synchronization event is any event that has the `beginOfSynchronization` flag set to `true` and the End of synchronization event is is any event that has the `endOfSynchronization` flag set to `true`. Any event type extending the `com.hp.uca.expert.event.DefaultEvent` will inherit the two methods: `setBeginOfSynchronization()` and `setEndOfSynchronization`() that allow for setting such flags.

  Any Event Type extending the `DefaultEvent` class can therefore be used as begin and end of synchronization events by setting these falgs. It is imperative however that these events not be “real” events, and thus not contain “real” event data. These events should only be used to flag the start and end of the resynchronization.

  In the case of an Alarm flow, the resynchronization event flow starts with the specific `com.hp.uca.expert.alarm.internal.BeginSynchronization` event and is ter minated by sending the `com.hp.uca.expert.alarm.internal.EndSynchronizat ion` event.

- `getCollectionAudit():` This method returns a description of the Collection flow. It is mainly used for troubleshooting purposes. It is called by the UMB framework when an audit request on the collection flow is received (`AuditFlow` request). This method returns a `MAP<String,String>` representing a key/value collection describing the collector information.

  Such information is retrieved by the consumer side by calling the `auditCollection()` method on the ConsumerFlow object.

- `stopCollection()`: This method stops the Collection flow. It is called by the UMB framework when a `stopCollection()` request is made on the ConsumerFlow side. This method returns an `ActionReply` object to indicate whether the stop of the collection flow was successful or not.

• Collection flow method:
  - `pull()`: This method pulls a set of messages from a collection source. It returns a Collection of events to the UMB framework. The framework pushes the collection of returned event to the production Topic. Implementations of this method should throw `InterruptedException` if the current thread is interrupted. This is the main method of any “Collector” Java class because it is the one that actually collects alarms/events.

IMPORTANT NOTE:
When implementing this method, please use a `java.util.List` instead of merely a `java.util.Collection` as the return type of the `pull()` method. This will preserve the order of events being collected.

You can find examples of implementation of “Collector” Java classes in the Camel Adapter (`CamelCollector.java` class) and File Adapter (`FileCollector.java` and `TemperaturesCollector.java` classes) described in this document:

NOTE:

- For more information on the Camel Adapter, please refer to chapter 5.1 “Camel Adapter”
- For more information on the File Adapter, please refer to chapter 5.2 “File Adapter”.

Once you have both declared a producer collection flow in the `AdapterConfiguration.xml` file and implemented the associated “Collector” class, you have successfully added a producer collection flow service to your UMB Adapter.

If the producer collection flow that you have created is static, then it will be automatically started by the UMB Framework when the Adapter starts (unless the `autoStarted` attribute of the `<flow>` XML element is set to `false`). On the other hand, if the producer collection flow is dynamic, the Adapter will wait for a `CreateFlow` collection action request initiated by a flow consumer to start the producer collection flow.

Producer collection flows can be consumed by any UMB Adapter. Setting up consumer flows is explained in detail in chapter 3.2.4 “Adding consumer flows”.

3.2.2.1 Managing the Collectors state

A state is associated to each Collector instance. This state can be retrieved by the `getStatus()` method of the `BaseCollector` Class. This method returns an object of type `FlowStatus` that can take the following values: (`UNKNOWN, DISABLED, STARTING, ACTIVE, FAILOVER, STOPPING, INACTIVE, FAILED`)

The different states are managed by the framework itself during the Collector lifecycle. This is the case for the UNKNOWN->STARTING->ACTIVE transitions and for the ACTIVE->STOPPING->INACTIVE transitions.

However, the Collector class can set its state using the `setStatus()`method mainly to notify collection errors.

For such a purpose, there are two states available:

- The FAILED state:

  This state is use to notify a fatal collection error. In such a case, the Collection Flow is considered not usable anymore. This state will be propagated to the Consumer side which in turn will be set in the FAILED state. If the consumer is monitored, the framework will automatically restart the flow.

- The FAILOVER state:

  This state indicates that there is a collection error, but the Collector itself will try to re-establish the collection. This FAILOVER state is propagated to the Consumer side for information, but no specific action is taken from the framework. It is the responsibility of the Collector to re-establish the collection itself. When the collection is successfully re-established, the Collector state must be set back to the ACTIVE state. In case of failure, the state should be changed to FAILED.

NOTE:
Any attempt to change the flow state to a value other than FAILED or FAILOVER or turn it back to ACTIVE when it was FAILOVER will lead to an exception.

3.2.3 Adding action services

In order to add action services to your Adapter, you first have to define the actions or types of actions that you want to provide in the `AdapterConfiguration.xml` file.

In this file, action services are defined by adding the `<actionServices>…</actionServices>` XML element inside the enclosing `<adapter>…</adapter>` root XML element.

Each action service is defined by adding a `<action>…</action>` XML element inside the enclosing `<actionServices>…</actionServices>` XML element.

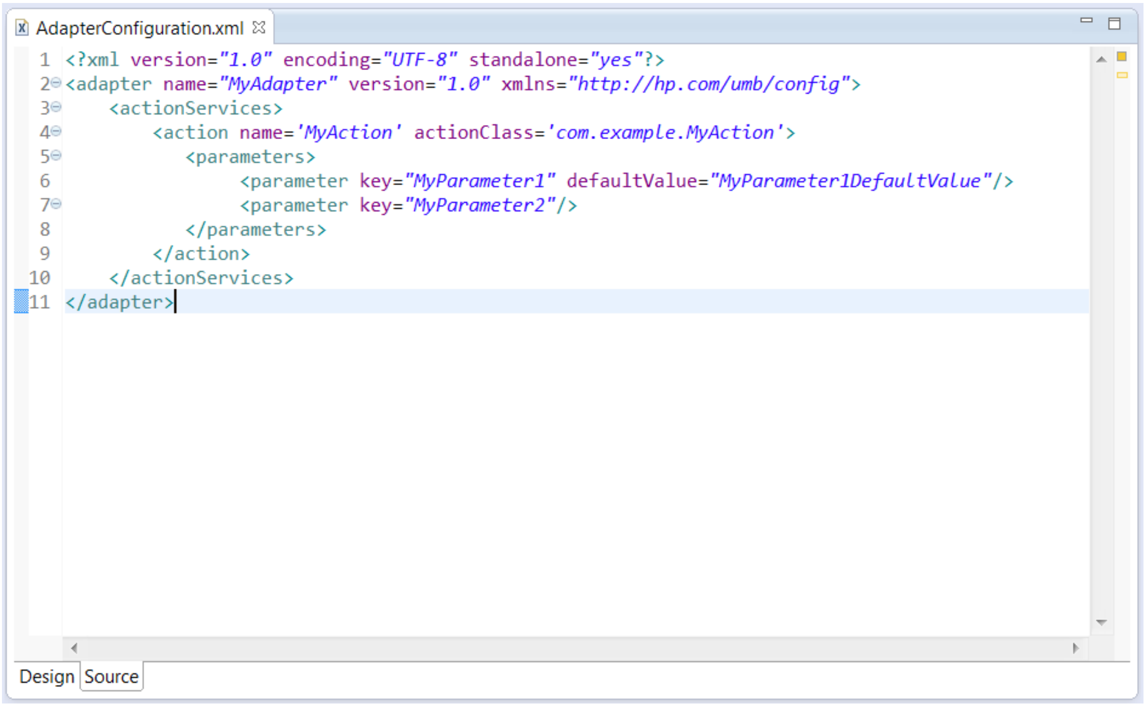
Each `<action>…</action>` XML element must define all of the following mandatory attributes:

Please refer to section “1.2.3.3 Action Services” in this documents for a full description of the action service attributes.

Each `<action>…</action>` XML element can also define parameters associated with the action by adding the optional `<parameters>…</parameters>` XML element. Inside the `<parameters>…</parameters>` XML element, each parameter is defined by a `<parameter>…</parameter>` XML element.

Refer to section “1.2.3.3.1 Action Parameters” in this documents for a full description of the action parameter attribute definitions.

Below is an example of an `AdapterConfiguration.xml` file that defines one action:



AdapterConfiguration.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="MyAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <actionServices>
        <action name='MyAction' actionClass='com.example.MyAction'>
            <parameters>
                <parameter key="MyParameter1" defaultValue="MyParameter1DefaultValue"/>
                <parameter key="MyParameter2"/>
            </parameters>
        </action>
    </actionServices>
</adapter>
```

Figure 23 - Example of an action in the AdapterConfiguration.xml file

Once the action services have been defined in the `AdapterConfiguration.xml` file, it is necessary to create an "Action" Java class for each action. The name of the class has to match the value of the **actionClass** attribute of the `<action>…</action>` XML element in the `AdapterConfiguration.xml` file. It is mandatory that this class extends the `com.hp.umb.adapter.BaseAction` abstract Java class.

You can create an "Action" Java class by using the context menus in your Adapter project:

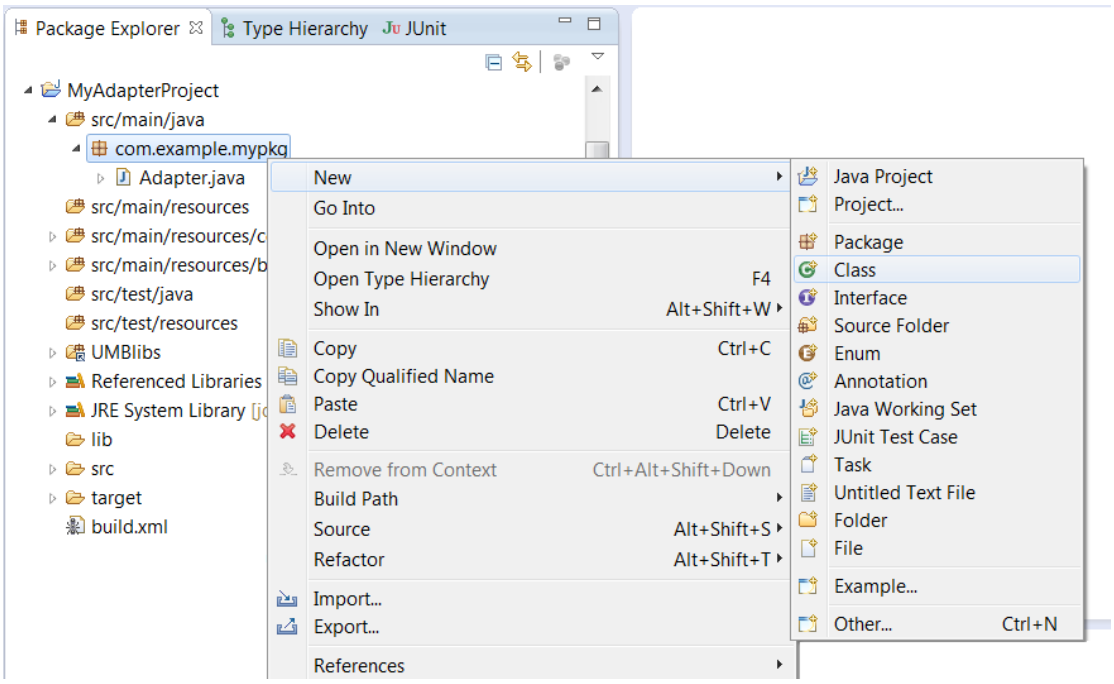
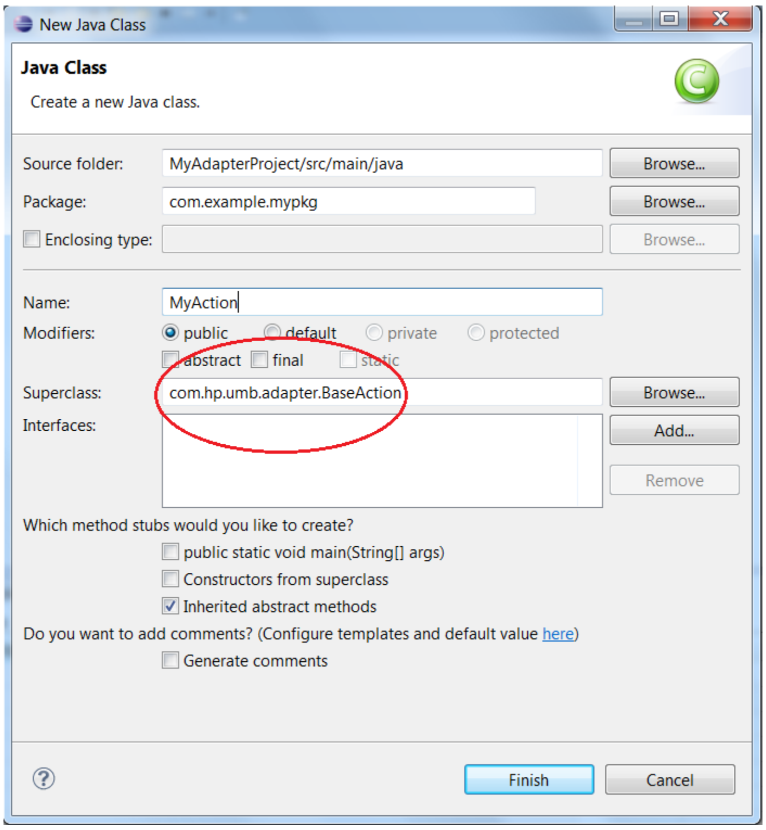


Figure 24 - Creating an "Action" Java class – Step 1

This opens the New Class window.


New Java Class

Java Class

Create a new Java class.

Source folder: MyAdapterProject/src/main/java

Package: com.example.mypkg

☐ Enclosing type:

Name: MyAction

Modifiers: [✓ public] default private protected

☐ abstract ☐ final ☐ static

Superclass: com.hp.umb.adapter.BaseAction

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

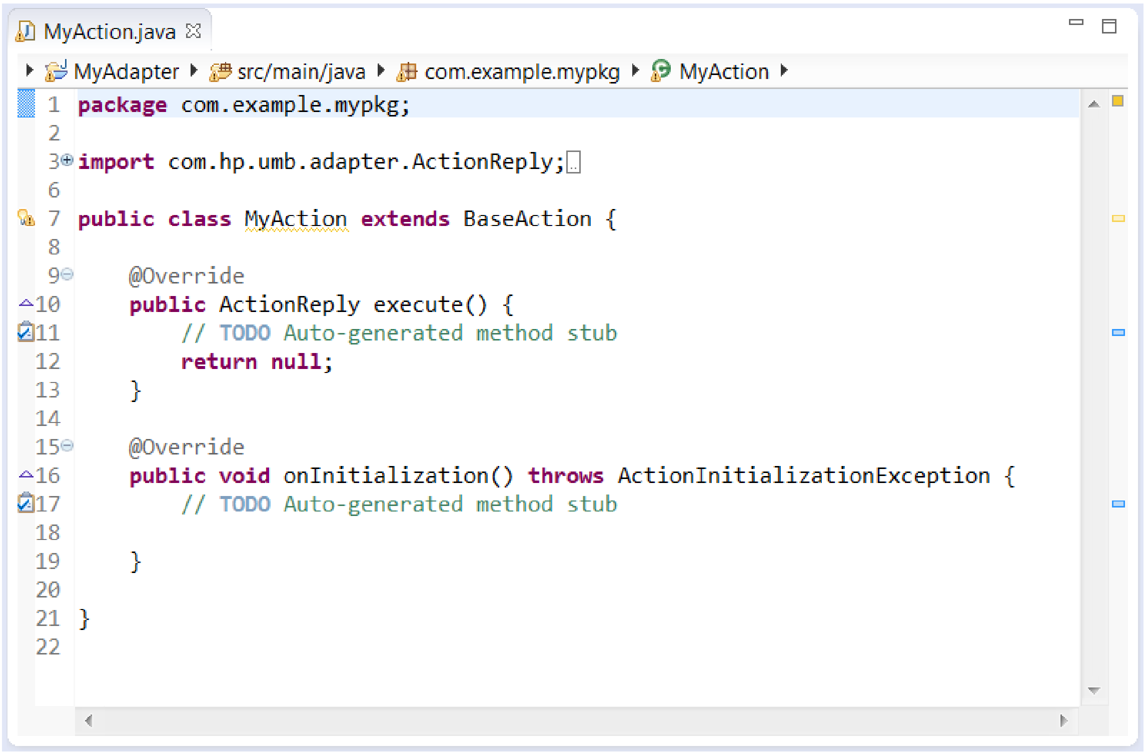
Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Figure 25 - Creating an "Action" Java class – Step 2

Please make sure that the new “Action” Java class that you're creating extends the `com.hp.umb.adapter.BaseAction` abstract Java class.

Once you click on the `Finish` button, the new “Action” Java class is created as shown below:



MyAction.java

MyAdapter ▸ src/main/java ▸ com.example.mypkg ▸ MyAction ▸

```java
package com.example.mypkg;

import com.hp.umb.adapter.ActionReply;

public class MyAction extends BaseAction {

    @Override
    public ActionReply execute() {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public void onInitialization() throws ActionInitializationException {
        // TODO Auto-generated method stub

    }

}
```

Figure 26 - Creating an “Action” Java class – Step 3

In order to finalize the new “Action” Java class, several methods need to be implemented:

- Initialization method:
  - `onInitialization(…)`: This method initializes the creation of an action. It is automatically called by the Unified Mediation Bus framework when the action is created. If you need to initialize objects or resources associated with the “Action” object, this is the place to do it.
- Action method:
  - `execute()`: This method executes an action. It is automatically called by the Unified Mediation Bus framework when the action is executed. If the action being executed is cancelled (by the requester), the current thread will be interrupted. This is the main method of any “Action” Java class because it is the one that actually executes the action.

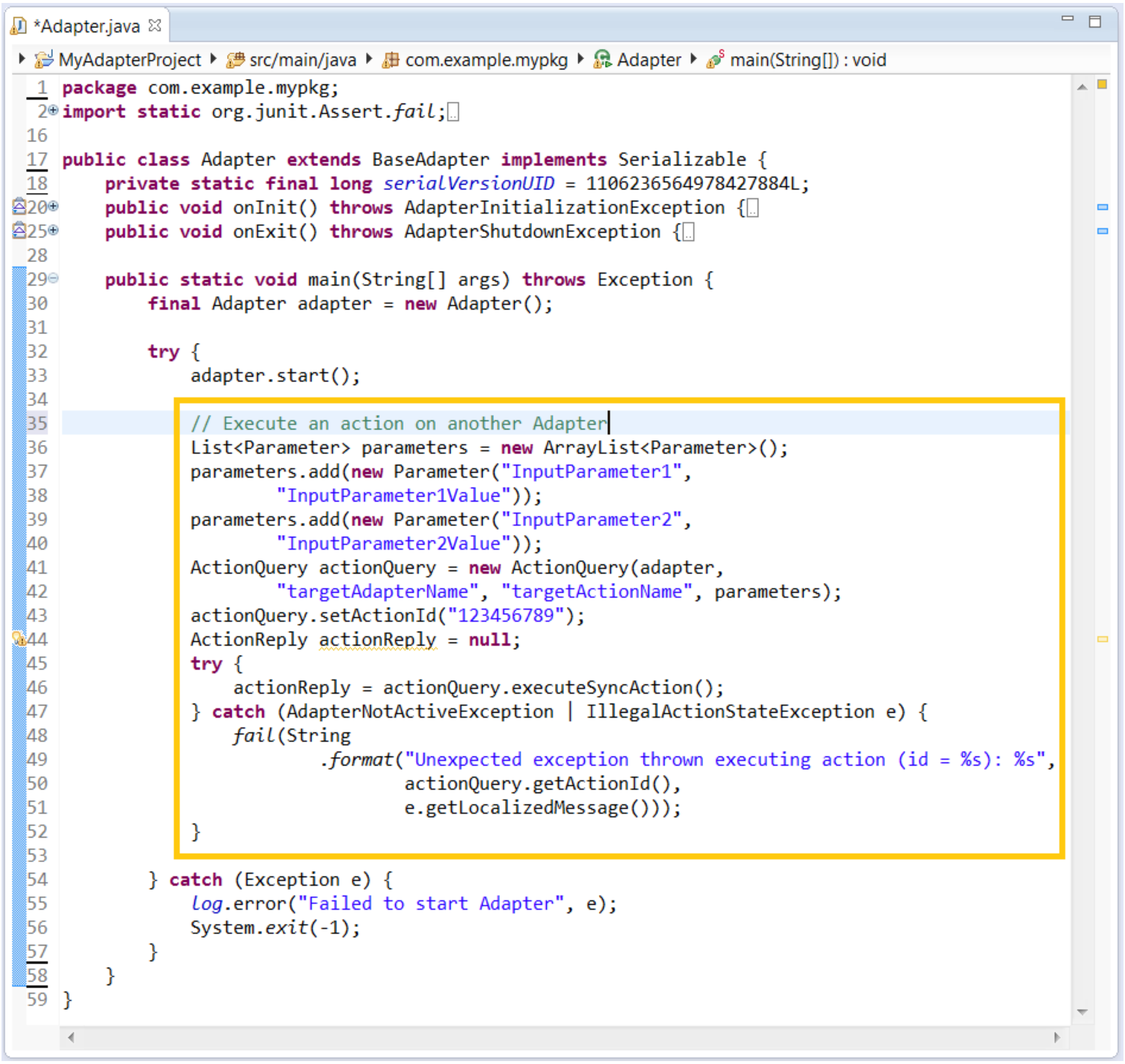
You can find example of implementation of “Action” Java classes in the Camel Adapter (`CamelAction.java` class) described in this document:

NOTE: For more information on the Camel Adapter, please refer to chapter 5.1 “Camel Adapter”.

Once you have both declared an action in the `AdapterConfiguration.xml` file and implemented the associated “Action” class, you have successfully added an action service to your UMB Adapter.

In order for the new action to be executed, the Adapter has to be started and an `ExecuteAction` action request has to be sent to the Adapter.

Any UMB Adapter can request actions to be executed by any Adapter providing action services by simply using Java code as show below:


*Adapter.java
MyAdapterProject ▸ src/main/java ▸ com.example.mypkg ▸ Adapter ▸ main(String[]) : void

```java
package com.example.mypkg;
import static org.junit.Assert.fail;

public class Adapter extends BaseAdapter implements Serializable {
    private static final long serialVersionUID = 1106236564978427884L;
    public void onInit() throws AdapterInitializationException {
    public void onExit() throws AdapterShutdownException {

    public static void main(String[] args) throws Exception {
        final Adapter adapter = new Adapter();

        try {
            adapter.start();

            // Execute an action on another Adapter
            List<Parameter> parameters = new ArrayList<Parameter>();
            parameters.add(new Parameter("InputParameter1",
                    "InputParameter1Value"));
            parameters.add(new Parameter("InputParameter2",
                    "InputParameter2Value"));
            ActionQuery actionQuery = new ActionQuery(adapter,
                    "targetAdapterName", "targetActionName", parameters);
            actionQuery.setActionId("123456789");
            ActionReply actionReply = null;
            try {
                actionReply = actionQuery.executeSyncAction();
            } catch (AdapterNotActiveException | IllegalActionStateException e) {
                fail(String
                        .format("Unexpected exception thrown executing action (id = %s): %s",
                                actionQuery.getActionId(),
                                e.getLocalizedMessage()));
            }

        } catch (Exception e) {
            log.error("Failed to start Adapter", e);
            System.exit(-1);
        }
    }
}
```

Figure 27 - Java code to send action requests

As seen in the example Java code above, you need to first create an `ActionQuery` object, specifying the source Adapter, target Adapter Name (in case you want to load balance action execution across multiple adapter, you can specify a target Adapter Group name instead: the action will be executed on a randomly selected adapter that's part of the group), target Action Name and parameters.

Then you can execute the action by calling the `executeSyncAction()` method on the `ActionQuery` object. This will execute the action synchronously (i.e. the method call is blocking until the action is complete). When the action is complete, an `ActionReply` object is returned.

Alternatively it is also possible to execute the action asynchronously:

- Synchronous execution methods (from the ActionQuery Java class):
    - public ActionReply **executeSyncAction**() throws IllegalActionStateException, AdapterNotFoundException, AdapterNotActiveException
    - public ActionReply **executeSyncAction**(long timeout) throws IllegalActionStateException, AdapterNotFoundException, AdapterNotActiveException
- Asynchronous execution methods (from the `ActionQuery` Java class):

    - public void **executeAsyncAction**() throws IllegalActionStateException, AdapterNotFoundException, AdapterNotActiveException
    - public void **executeAsyncAction**(ActionCallback callback) throws IllegalActionStateException, AdapterNotFoundException, AdapterNotActiveException
- Methods for retrieving the result of an asynchronous action (from the `ActionQuery` Java class):
    - public ActionReply **getAsyncActionReply**() throws IllegalActionStateException
    - public ActionReply **getAsyncActionReply**(long timeout) throws TimeoutException, IllegalActionStateException
- Cancellation method (from the `ActionQuery` Java class):
    - public ActionReply **cancelAction**() throws IllegalActionStateException

NOTE: For more information, please refer to the UMB Development Toolkit Javadoc available either in Eclipse IDE or directly from the UMB Development Toolkit installation directory:

- %UMB_DEV_HOME%\apidoc on Windows systems
- ${UMB_DEV_HOME}/apidoc on Linux systems

3.2.4 Adding consumer flows

In order to add consumer flows to your Adapter, you have 2 options:

- Either define automatic consumer flows (started automatically by the UMB Framework when the Adapter starts)
- Or create your own consumer flows and start them from the Java code of your Adapter

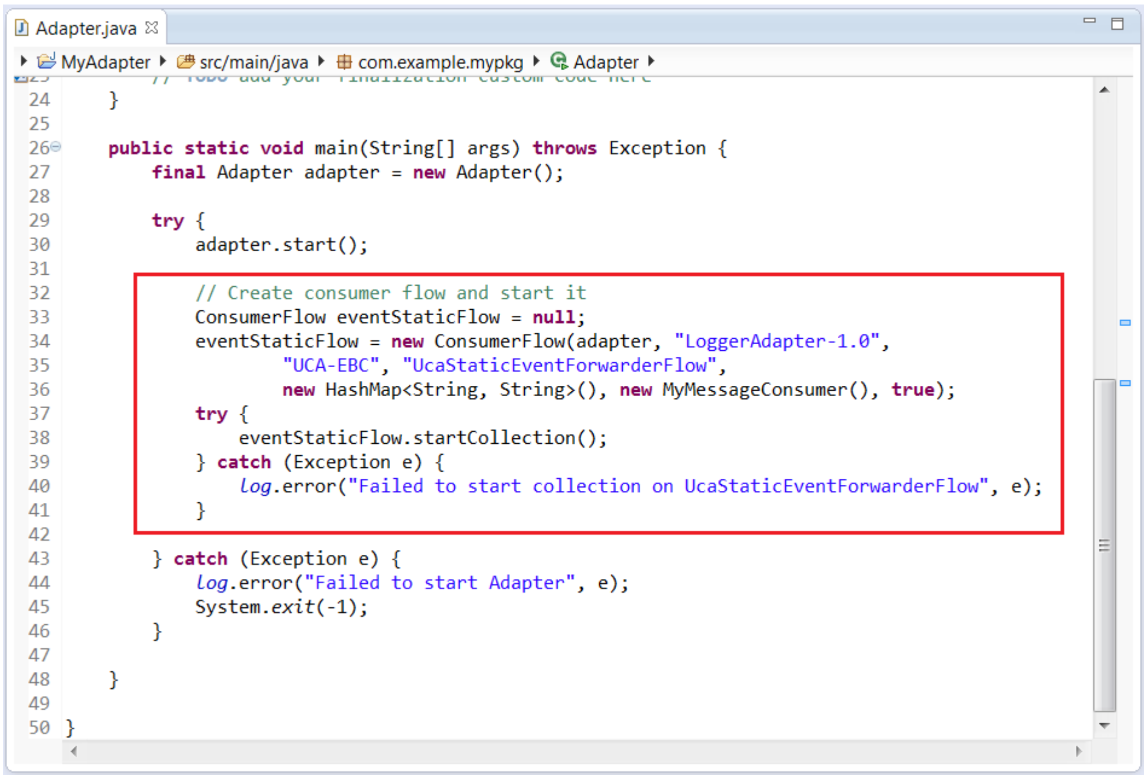
3.2.4.1 Automatically started Consumer flows

Defining consumer flows that start automatically in the `AdapterConfiguration.xml` file is straightforward.

Please refer to section 1.2.3.2 “Defining Flow Service Consumers by configuration” for full description on how to define automatic consumers.

3.2.4.2 Consumer flows started from java code

Alternatively to defining consumer flows that are created and started by the Framework, it is also possible to create consumer flows and start them directly from the Java code of your Adapter:



```java
    }

    public static void main(String[] args) throws Exception {
        final Adapter adapter = new Adapter();

        try {
            adapter.start();

            // Create consumer flow and start it
            ConsumerFlow eventStaticFlow = null;
            eventStaticFlow = new ConsumerFlow(adapter, "LoggerAdapter-1.0",
                    "UCA-EBC", "UcaStaticEventForwarderFlow",
                    new HashMap<String, String>(), new MyMessageConsumer(), true);
            try {
                eventStaticFlow.startCollection();
            } catch (Exception e) {
                log.error("Failed to start collection on UcaStaticEventForwarderFlow", e);
            }

        } catch (Exception e) {
            log.error("Failed to start Adapter", e);
            System.exit(-1);
        }

    }
}
```

Figure 28 - Java code to create consumer flows

The flow is created using the `ConsumerFlow()` constructor (giving all the necessary parameters. These are the same parameters as the ones described for automatically started Consumer Flows.)

The flow collection is started by calling the `startCollection()` method . And the collection can be stopped by calling the `stopCollection()` method.

The Consumer flows created from an Adapter's custom Java code also allow the flow to be resynchronized (in case the Flow Producer side supports this capability). In such a case, the re-synchronization of the flow can be requested by calling the `resyncCollection()` method.

3.2.4.3 Defining the flow message consumer class

Both the automatic consumer flows and consumer flow created from Java code require a "MessageConsumer" Java class.

A "MessageConsumer" class must:

1. Extend the `com.hp.umb.adapter.consumer.BaseConsumerMessageHandler` class.
2. Implement one of the following java interfaces:
    - `com.hp.umb.adapter.consumer.ConsumerMessagesHandlerInterface< K extends Event>`

      With this interface the messages are returned by sets of messages.

      The maximum message set size is specified by calling the `setConsumerMaxSetSize()` method on the consumer Flow:

      For example:

      ```
      myFlow.setConsumerMaxSetSize(100);
      ```

      which sets the maximum set size to 100.

The UMB framework reads the Kafka topic until no message arrives during a given time period (the timeout period). This timeout is a configuration parameter of the ConsumerFlow. It is specified by calling the `setConsumerTimeout()` method on the consumer Flow:

Example:

```
myFlow.setConsumerTimeout (500);
```

which sets the read timeout to 500 milliseconds. The default timeout is set to one second.

The UMB framework calls the `onNewMessageSet()` method of the "MessageConsumer" class when either the number of event in the set reaches the maximum set size, or when the specified timeout elapses.

You can create a "MessageConsumer" Java class by using the context menus in your Adapter project:

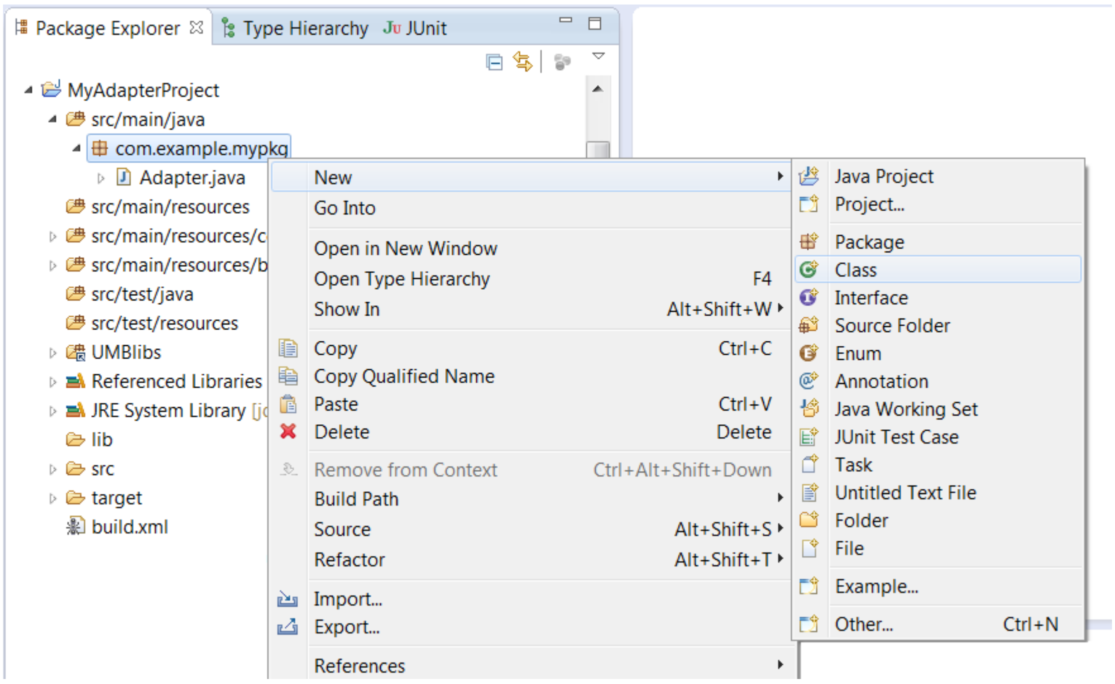
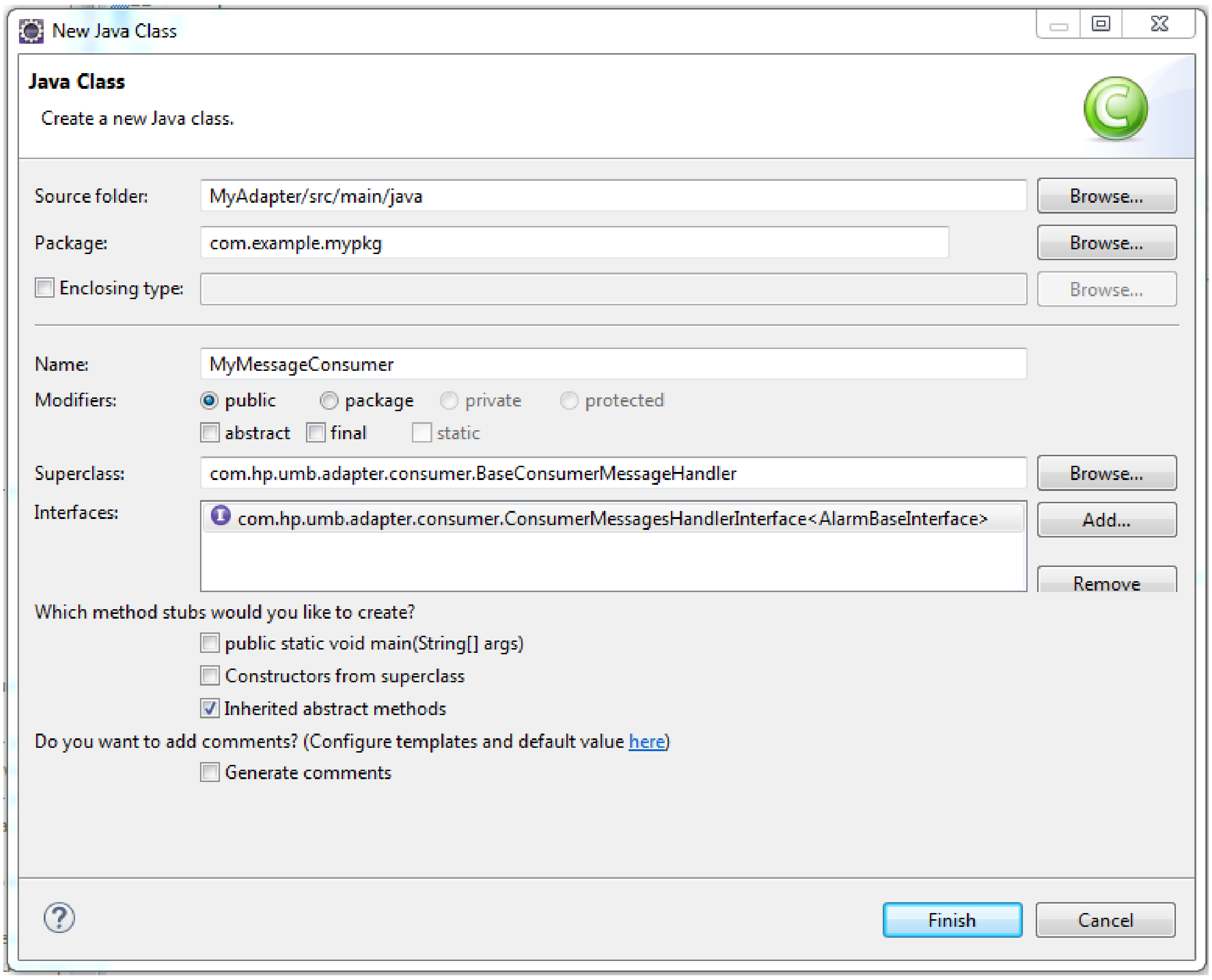


Figure 29 - Creating a "MessageConsumer" Java class – Step 1

This opens the New Class window.


New Java Class

Java Class

Create a new Java class.

Source folder: MyAdapter/src/main/java

Package: com.example.mypkg

Enclosing type: [ ]

Name: MyMessageConsumer

Modifiers: [✓ public] package private protected

[ ] abstract [ ] final [ ] static

Superclass: com.hp.umb.adapter.consumer.BaseConsumerMessageHandler

Interfaces: com.hp.umb.adapter.consumer.ConsumerMessagesHandlerInterface<AlarmBaseInterface>

Which method stubs would you like to create?

[ ] public static void main(String[] args)

[ ] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

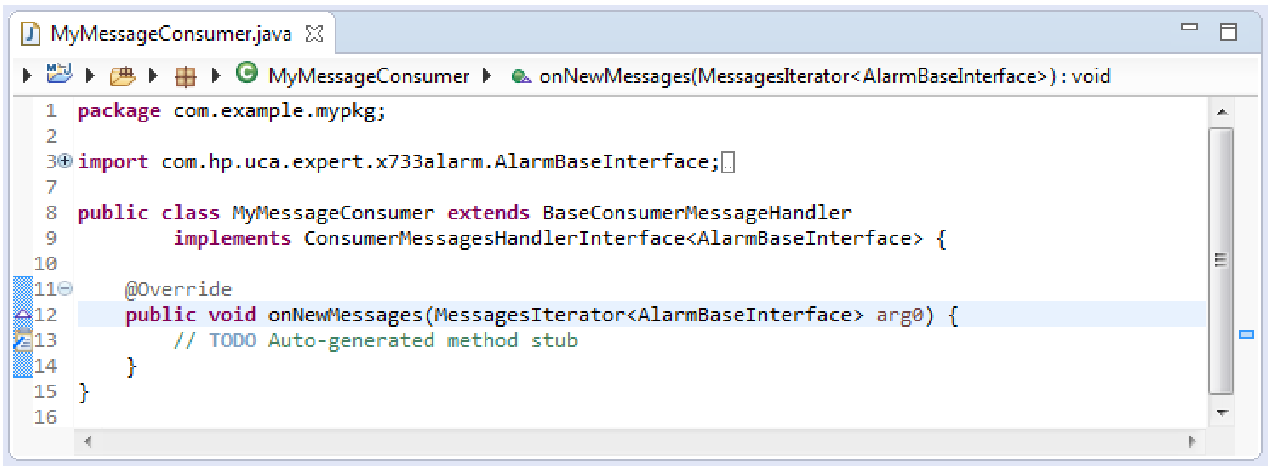
[ ] Generate comments

Figure 30 - Creating a "MessageConsumer" Java class – Step 2

Please make sure that the new "MyMessageConsumer" Java class that you're creating extends the `com.hp.umb.adapter.consumer.BaseConsumerMessageHandler` class and implements `com.hp.umb.adapter.consumer.ConsumerMessagesHandlerInterface<K extends BasicEvent>` Java interface.

Note also that you have to specify the interface's formal type K (in the example: `com.hp.uca.alarm.AlarmBaseInterface`)

Once you click on the `Finish` button, the new "MyMessageConsumer" Java class is created as shown below:


MyMessageConsumer.java
MyMessageConsumer ▸ onNewMessages(MessagesIterator<AlarmBaseInterface>) : void

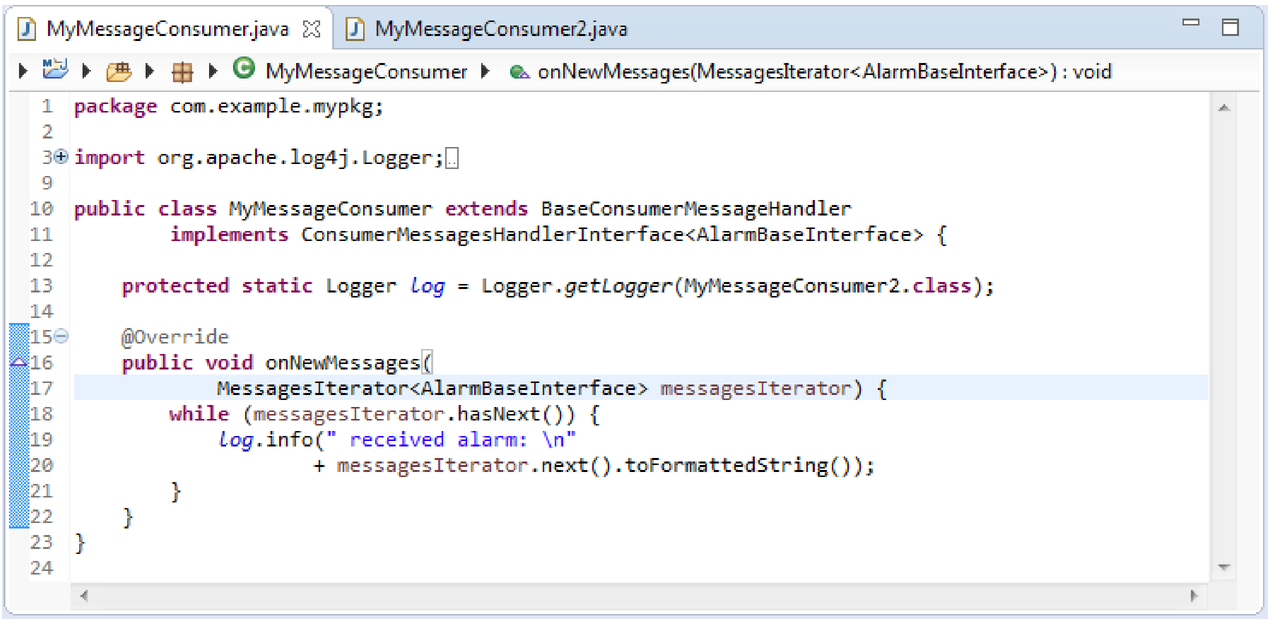
```java
package com.example.mypkg;

import com.hp.uca.expert.x733alarm.AlarmBaseInterface;

public class MyMessageConsumer extends BaseConsumerMessageHandler
        implements ConsumerMessagesHandlerInterface<AlarmBaseInterface> {

    @Override
    public void onNewMessages(MessagesIterator<AlarmBaseInterface> arg0) {
        // TODO Auto-generated method stub
    }
}
```

Figure 31 - Creating a "MessageConsumer" Java class – Step 3

In order to finalize the new "MessageConsumer" Java class, only one method needs to be implemented:

- `onNewMessage(…)`: This method is called by the UMB framework whenever an event/alarm is consumed from the flow.

  To consume the messages, the `onNewMessage()` method must use the iterator given as argument. A typical implementation should be as follows:


MyMessageConsumer.java MyMessageConsumer2.java
MyMessageConsumer ▸ onNewMessages(MessagesIterator<AlarmBaseInterface>) : void

```java
package com.example.mypkg;

import org.apache.log4j.Logger;

public class MyMessageConsumer extends BaseConsumerMessageHandler
        implements ConsumerMessagesHandlerInterface<AlarmBaseInterface> {

    protected static Logger log = Logger.getLogger(MyMessageConsumer2.class);

    @Override
    public void onNewMessages(
            MessagesIterator<AlarmBaseInterface> messagesIterator) {
        while (messagesIterator.hasNext()) {
            log.info(" received alarm: \n"
                    + messagesIterator.next().toFormattedString());
        }
    }
}
```

Once you have both declared an "auto" consumer flow in the `AdapterConfiguration.xml` file and implemented the associated "MessageConsumer" class, you have successfully added an "auto" consumer flow service to your UMB Adapter. The "auto" consumer flow will be started automatically when the Adapter starts.

3.2.4.4 Consumer flows state diagram

Consumer flows have a status that reflects the state of the collection they're consuming. This status is returned by calling the consumer flow's `getStatus()` method.

The consumer flow status diagram is as follows:

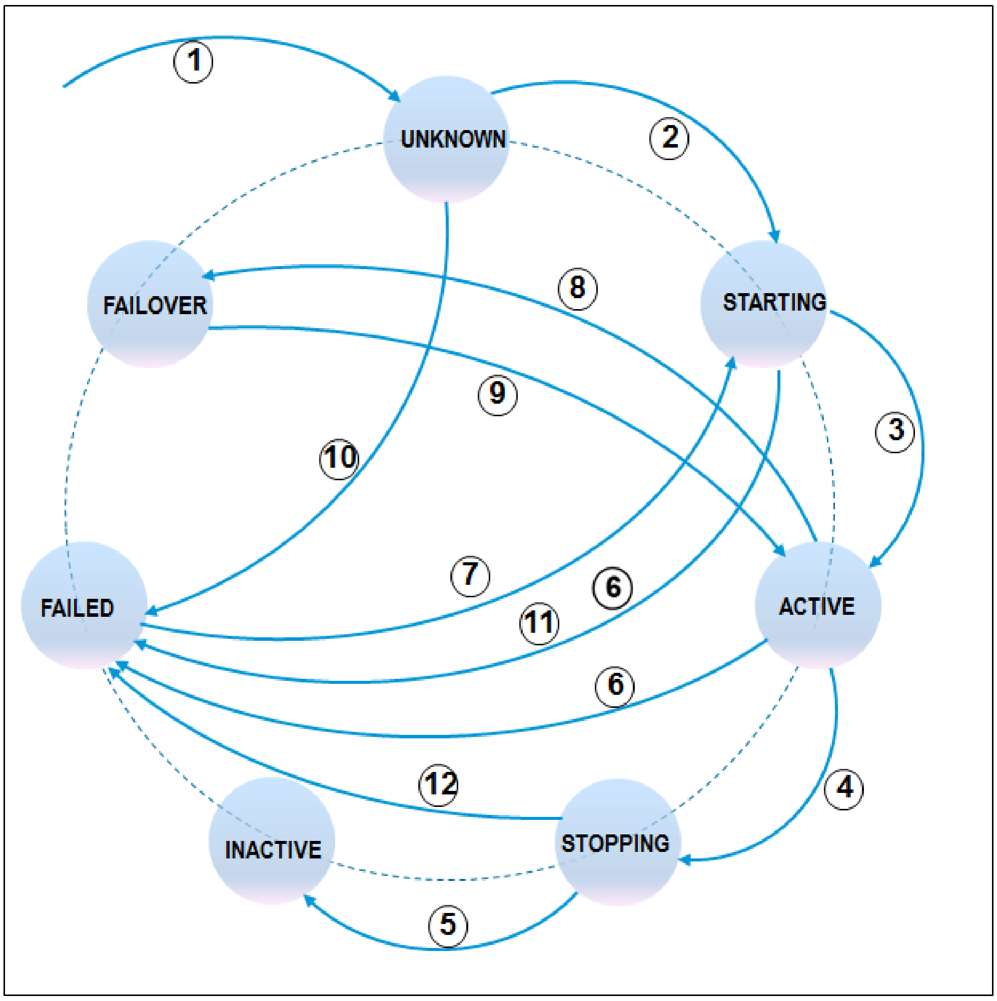


Figure 32 - Consumer Flow status diagram

At Consumer Flow creation the Status is set to UNKNOWN. The other states are the results of the following transitions:

1. ConsumerFlow creation: sets the consumer flow status to UNKNOWN
2. startCollection requested: changes the consumer flow status from UNKNOWN to STARTING
3. startCollection completed: changes the consumer flow status from STARTING to ACTIVE
4. stopCollection requested: changes the consumer flow status from ACTIVE to STOPPING
5. Collection successfully stopped : changes the consumer flow status from STOPPING to INACTIVE
6. Production flow Failed or Production Flow's Adapter stopped or
   Kafka server connection lost: changes the consumer flow status from STARTING or ACTIVE to FAILED
7. ConsumerFlow Restarted by Monitoring: changes the consumer flow status from FAILED to STARTING
8. Producer collector recovering: changes the consumer flow status from ACTIVE to FAILOVER
9. Producer collector recovered: changes the consumer flow status from FAILOVER to ACTIVE
10. Adapter not started: changes the consumer flow status from UNKNOWN to FAILED
11. Flow service does not exist: changes the consumer flow status from STARTING to FAILED
12. producer error while stopping flow: changes the consumer flow status from STOPPING to FAILED

The Adapter can react to consumer flow status changes by defining a Flow status change listener. This is done by calling the `addFlowStatusChangeListener()` adapter's method. This method requires a flow status listener instance to be passed as parameter.

The flow status listener instance must implement the `com.hp.umb.adapter.consumer.ConsumerFlowStatusListenerInterface` interface.

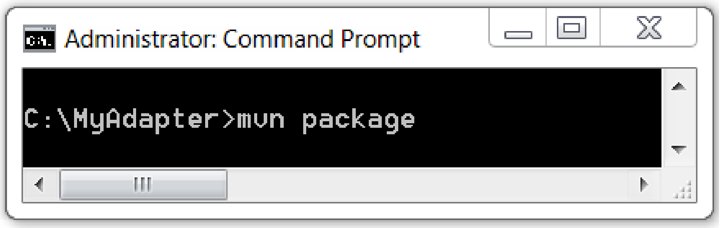
3.3 Generating the UMB Adapter kit

Once your Eclipse project has been updated, it is necessary to generate the kit associated with the Adapter so that it can be deployed, usually on the same system that runs the application that the adapter interfaces with.

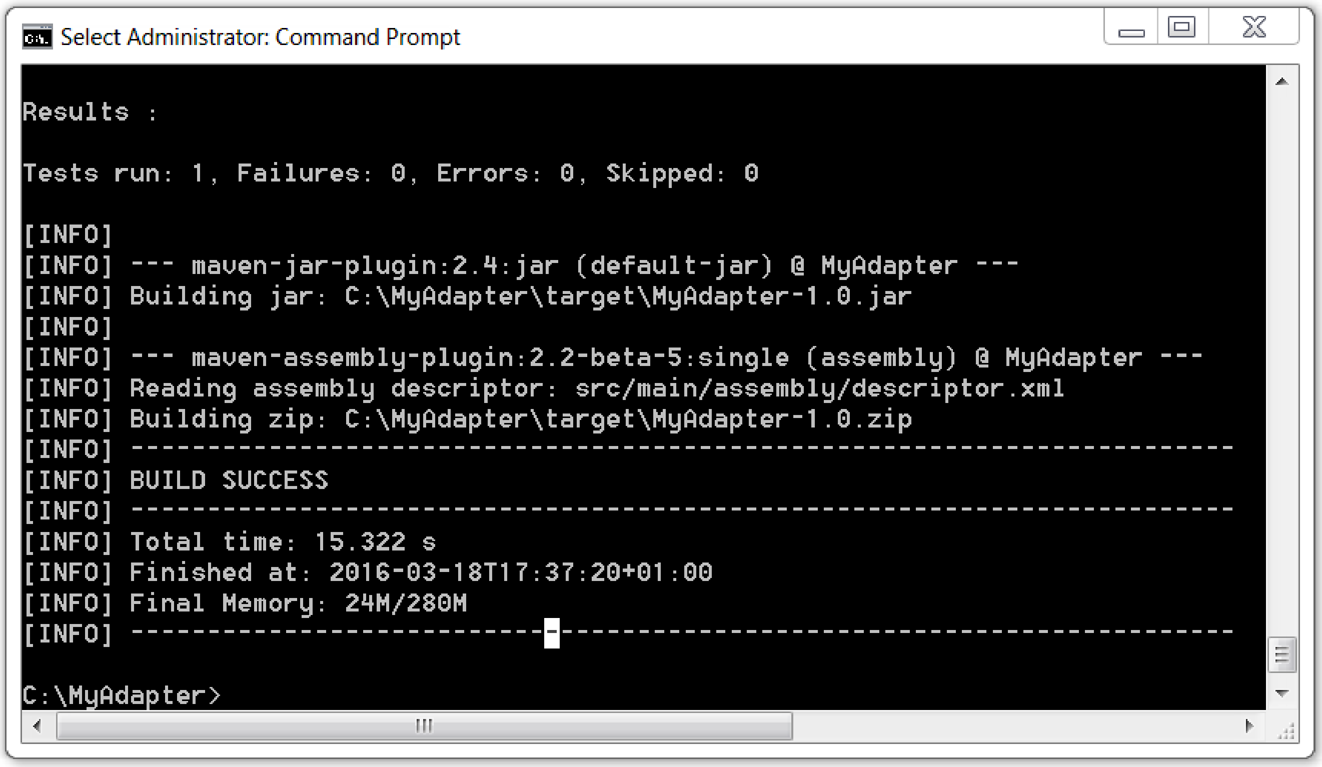
To do this, you just need to execute the following commands:

```
C:\> cd <Project Base>
C:\> mvn package
```

<Project Base> refers to the root directory of the Adapter Eclipse project.



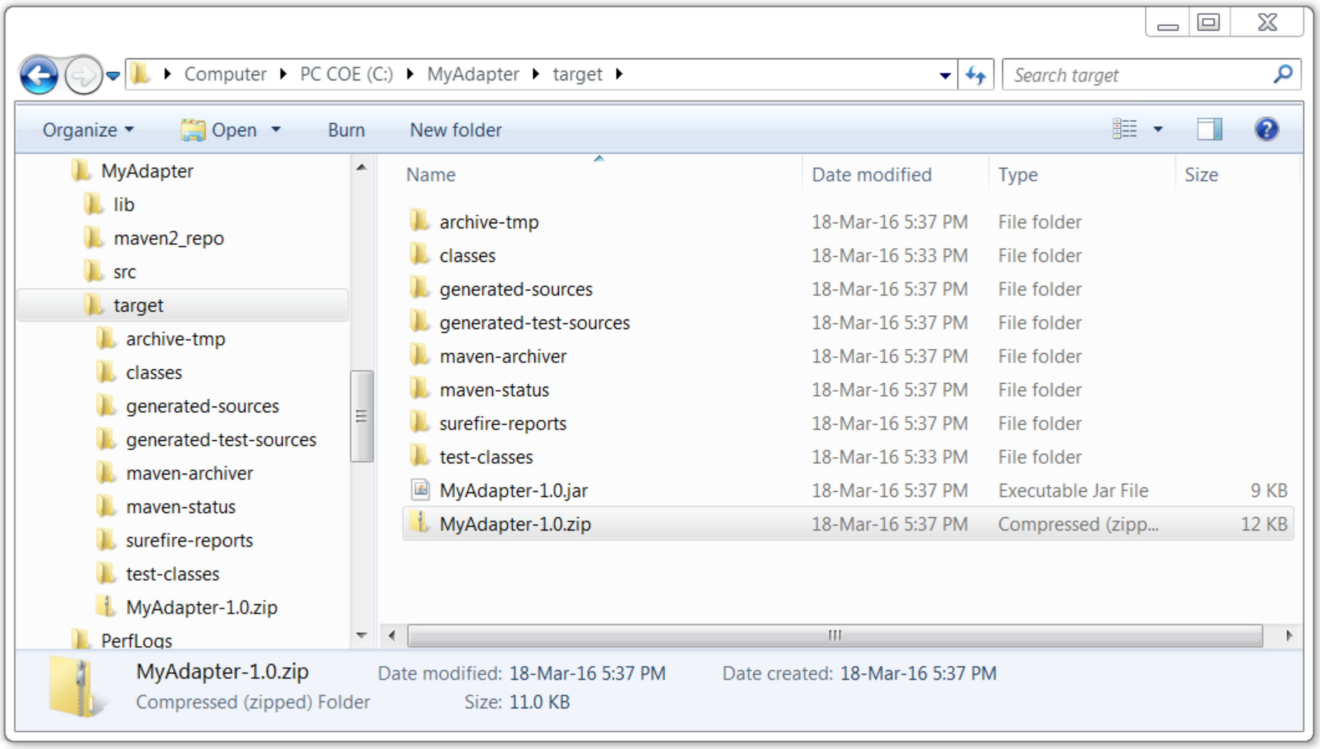
```
C:\MyAdapter>mvn package
```



```
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ MyAdapter ---
[INFO] Building jar: C:\MyAdapter\target\MyAdapter-1.0.jar
[INFO]
[INFO] --- maven-assembly-plugin:2.2-beta-5:single (assembly) @ MyAdapter ---
[INFO] Reading assembly descriptor: src/main/assembly/descriptor.xml
[INFO] Building zip: C:\MyAdapter\target\MyAdapter-1.0.zip
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 15.322 s
[INFO] Finished at: 2016-03-18T17:37:20+01:00
[INFO] Final Memory: 24M/280M
[INFO] ------------------------------------------------------------------------

C:\MyAdapter>
```

Figure 33 - Building the kit of your Adapter

The kit of the Adapter is then generated in the `target` directory of the *<Project Base>* directory as a Zip file called *<Adapter name>-<Adapter version>*`.zip` (for example `MyAdapterProject-1.0.zip`):



Name	Date modified	Type	Size
archive-tmp	18-Mar-16 5:37 PM	File folder	
classes	18-Mar-16 5:33 PM	File folder	
generated-sources	18-Mar-16 5:37 PM	File folder	
generated-test-sources	18-Mar-16 5:37 PM	File folder	
maven-archiver	18-Mar-16 5:37 PM	File folder	
maven-status	18-Mar-16 5:37 PM	File folder	
surefire-reports	18-Mar-16 5:37 PM	File folder	
test-classes	18-Mar-16 5:33 PM	File folder	
MyAdapter-1.0.jar	18-Mar-16 5:37 PM	Executable Jar File	9 KB
MyAdapter-1.0.zip	18-Mar-16 5:37 PM	Compressed (zipp...	12 KB

MyAdapter-1.0.zip Date modified: 18-Mar-16 5:37 PM Date created: 18-Mar-16 5:37 PM
Compressed (zipped) Folder Size: 11.0 KB

Figure 34 – Location of the kit of your Adapter

The ZIP file of your Adapter contains a root folder named `<Adapter name>` (for example `MyAdapter`) that contains the following sub-folders:

- `conf/` sub-folder which contains the Adapter's configuration files:
    - `adapter.properties`: defines the properties for the Adapter including connection information for the UMB Kafka/ZooKeeper instance(s)
    - `AdapterConfiguration.xml`: defines the flow and action services provided by the adapter as well as "automatic" consumer flows
    - `hazelcast.xml`: defines how to connect to the UMB Hazelcast instance(s)
    - `log4j.xml`: defines the Adapter's Log4j configuration
    - `startup.conf`: that contains a set of configurations used by the '`umb`' administration command-line tool.
- `lib/` sub-folder that contains the library files (jar files) that the Adapter depends on.

3.4 Installing and starting the UMB Adapter

Since UMB V1.1, you must use the '`umb`' administration tool to install, run and manage UMB Adapters.

To install a UMB Adapter from a Zip file, you should enter the following command:

```
$ umb install <Adapter name>-<Adapter version>.zip
```

To deploy the installed adapter in the '`default`' cluster:

```
$ umb deploy <Adapter name>
```

To start the adapter:

```
$ umb start <Adapter name>
```

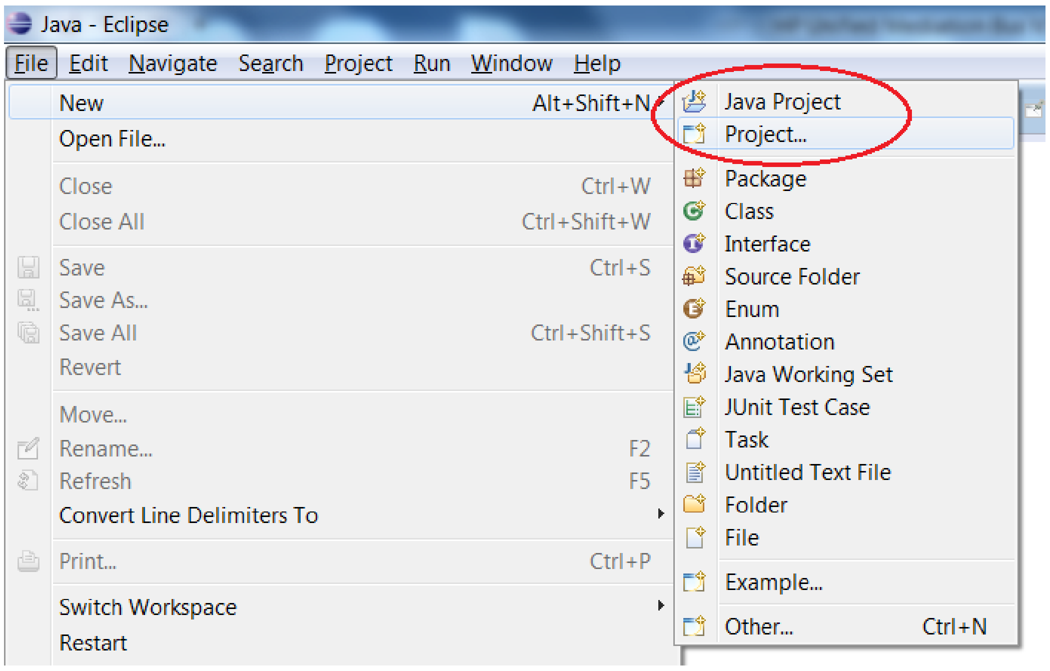
Please refer to the [R1] `HP Unified Mediation Bus– Installation and Configuration Guide` for a full description of the '`umb`' command

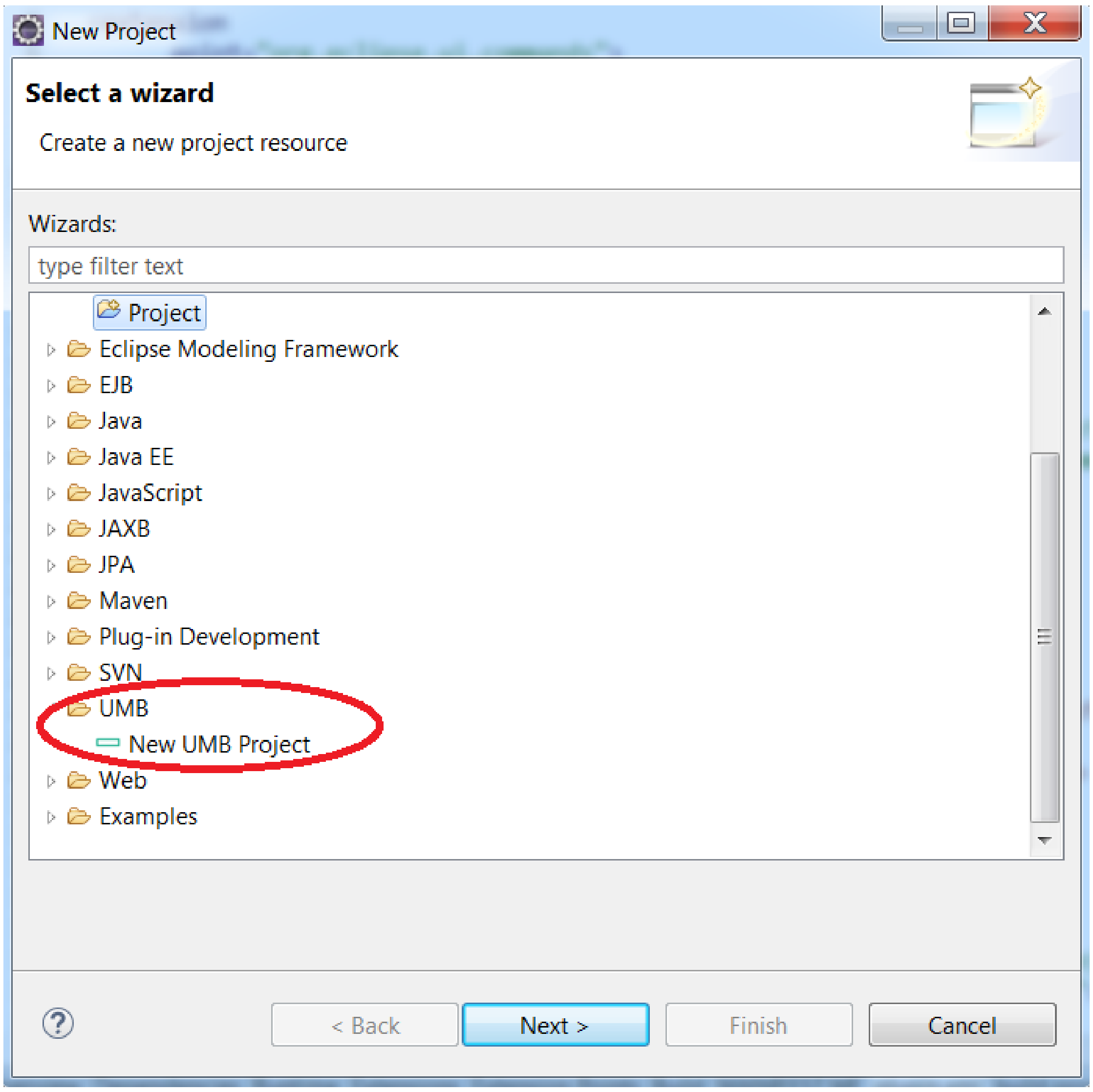
Chapter 4
Advanced development topics

4.1 Creating a new UMB Event Definition Project

UMB Adapters may require the implementation of some specific Event Classes that are passed from an adapter to another. Such classes must comply with some very specific requirements. The simplest way to create an Event class project is to use the UMB eclipse plug-in that allows for the creation of new UMB Event Definition projects in just a few clicks.

This plug-in can be launched from the Eclipse menus by selecting File -> New Project:




New Project

Select a wizard

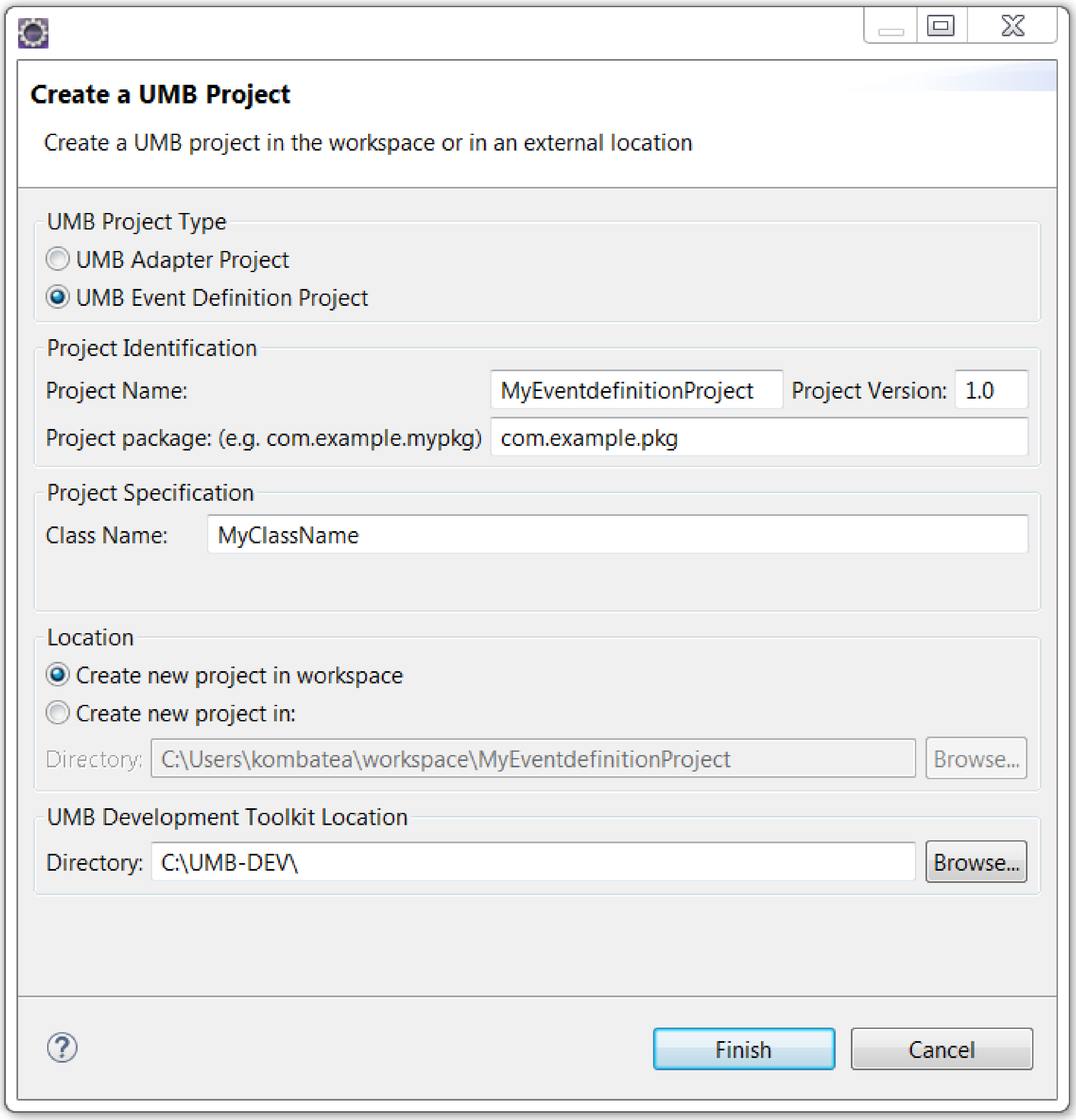
Create a new project resource

Wizards:

type filter text

- Project
- Eclipse Modeling Framework
- EJB
- Java
- Java EE
- JavaScript
- JAXB
- JPA
- Maven
- Plug-in Development
- SVN
- UMB
  - New UMB Project
- Web
- Examples

This launches the UMB Project wizard:


Create a UMB Project

Create a UMB project in the workspace or in an external location

UMB Project Type
- UMB Adapter Project
- [✓ UMB Event Definition Project]

Project Identification
Project Name: MyEventdefinitionProject Project Version: 1.0
Project package: (e.g. com.example.mypkg) com.example.pkg

Project Specification
Class Name: MyClassName

Location
- [✓ Create new project in workspace]
- Create new project in:

Directory: C:\Users\kombatea\workspace\MyEventdefinitionProject Browse...

UMB Development Toolkit Location
Directory: C:\UMB-DEV\ Browse...

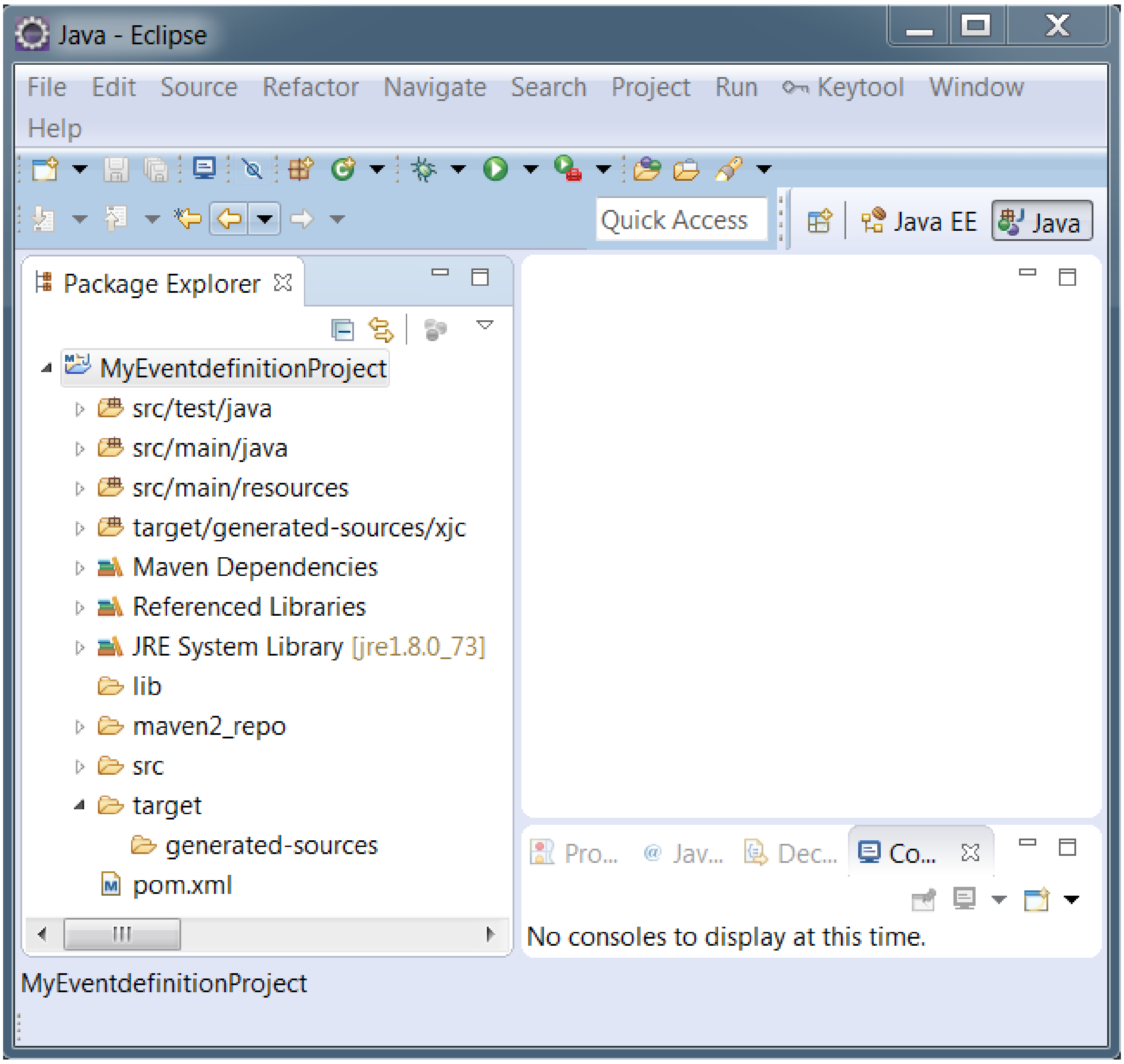
Figure 35 - UMB Event Definition project creation wizard Step1

From the UMB project wizard window, you can specify information regarding your UMB Event Definition project by specifying:

- The Project type: the type of UMB project to create
- The Project name: the name of the UMB Eclipse Event Definition to create
- The Project version: the version of the UMB project which is also the Event Definition version.
- The Project package: the name of the Java package to be used for the Java classes of the UMB Event Definition
- The Class Name: its value has to be given by the user. It is the name of the main class of the Event Definition project. The class name has to start with an upper case. No special characters are allowed.
- The Location: the location of the UMB Event Definition Eclipse project on the file system, either in the Eclipse workspace or anywhere on the file system
- The UMB Development Toolkit location: the location on the file system of the installation directory of the UMB Adapter Development Kit. The default value is the value of the **%UMB_DEV_HOME%** environment variable on

Windows systems (`${UMB_DEV_HOME}` on Linux): `C:\UMB-DEV` by default on Windows systems (`/opt/UMB-DEV` on Linux)

When you click on the Finish button, your UMB Event Definition Eclipse project (of Maven nature) is created. It has a minimum set of configuration, Java and JUnit files. It can be successfully compiled and unit tested.



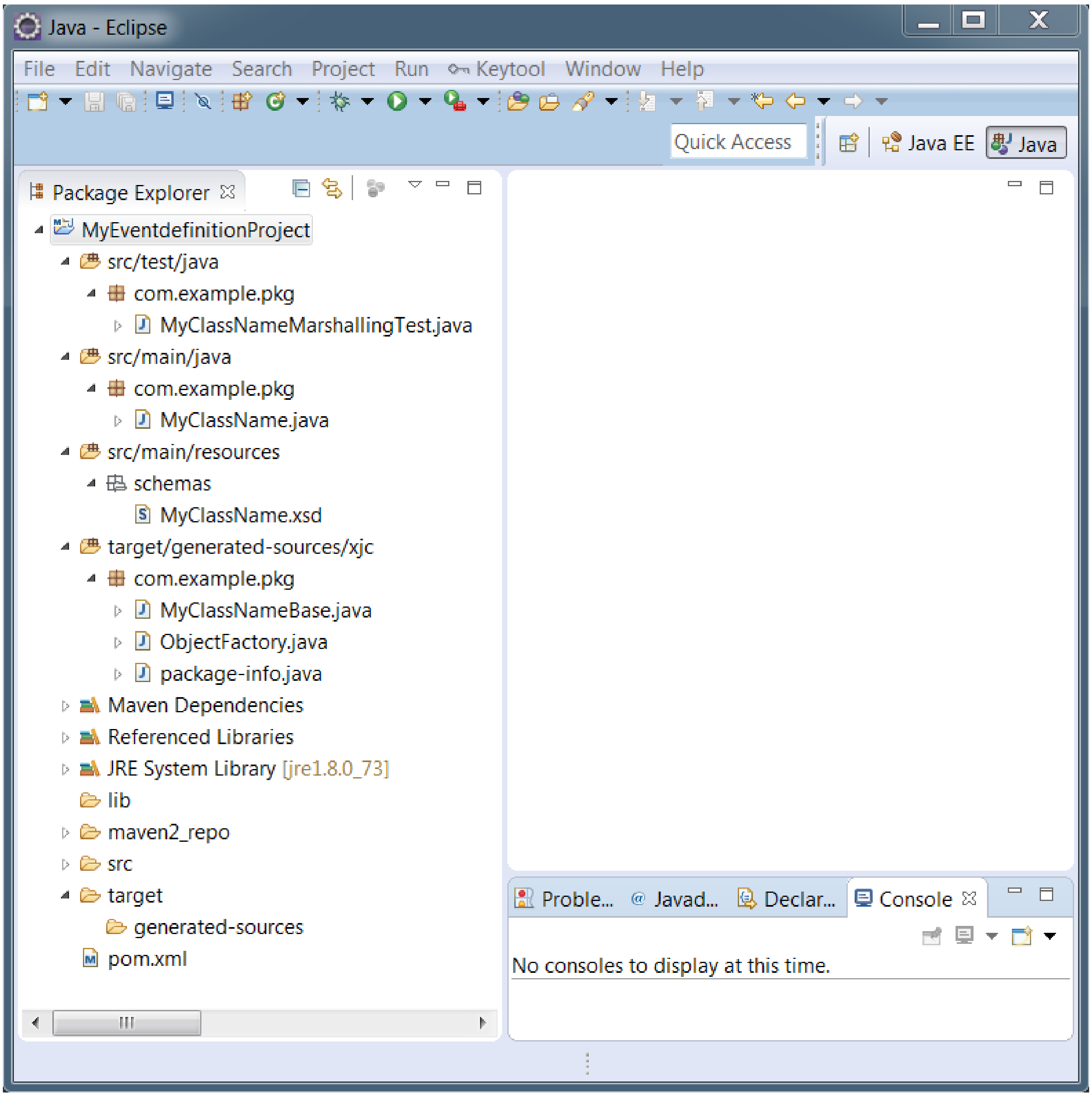
No consoles to display at this time.

Figure 36 – New UMB Event Definition project

4.1.1 Anatomy of the created project

Using Eclipse IDE, you can browse through the different directories that compose the newly created UMB Event Definition project.

Please see below for a look at the folder structure of a UMB Event Definition project:

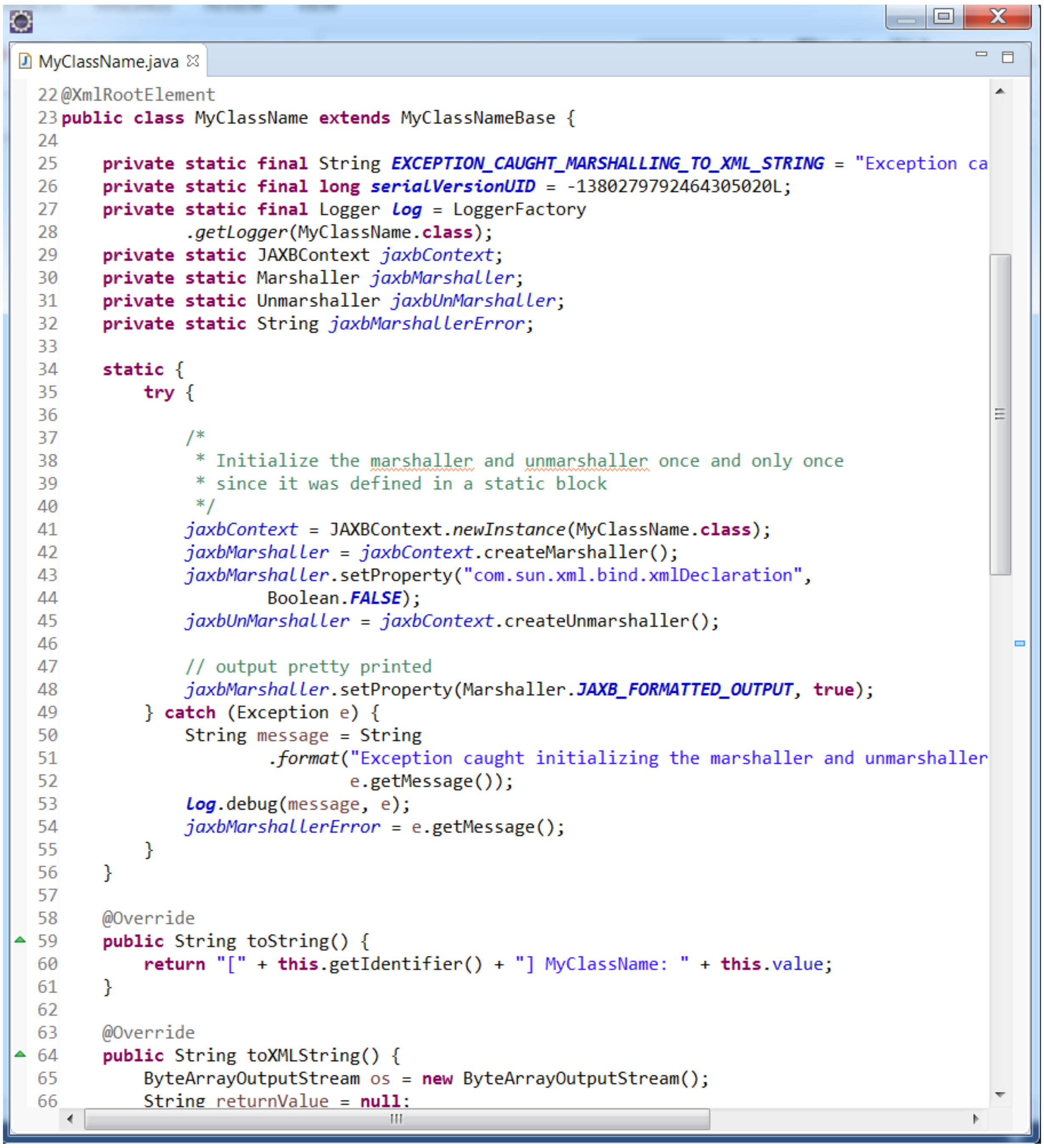


```
MyEventdefinitionProject
  src/test/java
    com.example.pkg
      MyClassNameMarshallingTest.java
  src/main/java
    com.example.pkg
      MyClassName.java
  src/main/resources
    schemas
      MyClassName.xsd
  target/generated-sources/xjc
    com.example.pkg
      MyClassNameBase.java
      ObjectFactory.java
      package-info.java
  Maven Dependencies
  Referenced Libraries
  JRE System Library [jre1.8.0_73]
  lib
  maven2_repo
  src
  target
    generated-sources
  pom.xml
```

```
No consoles to display at this time.
```

Figure 37 - Folder structure of the new UMB Event Definition project

The Event Definition Java class that defines the behavior of the Event Definition is located in the `src/main/java` folder. There is only one Java class present: `ClassName.java` class, where "`ClassName` "is the value entered by the user in the 'Class Name' field on the UMB eclipse project wizard. An object of this class represents an instance of the Event Definition. By default, this class implements a set of marshaller and unmarshaller methods that are mandatory when the Event class is used in the context of UCA-EBC.


MyClassName.java

```
@XmlRootElement
public class MyClassName extends MyClassNameBase {

    private static final String EXCEPTION_CAUGHT_MARSHALLING_TO_XML_STRING = "Exception ca
    private static final long serialVersionUID = -1380279792464305020L;
    private static final Logger log = LoggerFactory
            .getLogger(MyClassName.class);
    private static JAXBContext jaxbContext;
    private static Marshaller jaxbMarshaller;
    private static Unmarshaller jaxbUnMarshaller;
    private static String jaxbMarshallerError;

    static {
        try {

            /*
             * Initialize the marshaller and unmarshaller once and only once
             * since it was defined in a static block
             */
            jaxbContext = JAXBContext.newInstance(MyClassName.class);
            jaxbMarshaller = jaxbContext.createMarshaller();
            jaxbMarshaller.setProperty("com.sun.xml.bind.xmlDeclaration",
                    Boolean.FALSE);
            jaxbUnMarshaller = jaxbContext.createUnmarshaller();

            // output pretty printed
            jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        } catch (Exception e) {
            String message = String
                    .format("Exception caught initializing the marshaller and unmarshaller
                            e.getMessage());
            log.debug(message, e);
            jaxbMarshallerError = e.getMessage();
        }
    }

    @Override
    public String toString() {
        return "[" + this.getIdentifier() + "] MyClassName: " + this.value;
    }

    @Override
    public String toXMLString() {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        String returnValue = null;
```

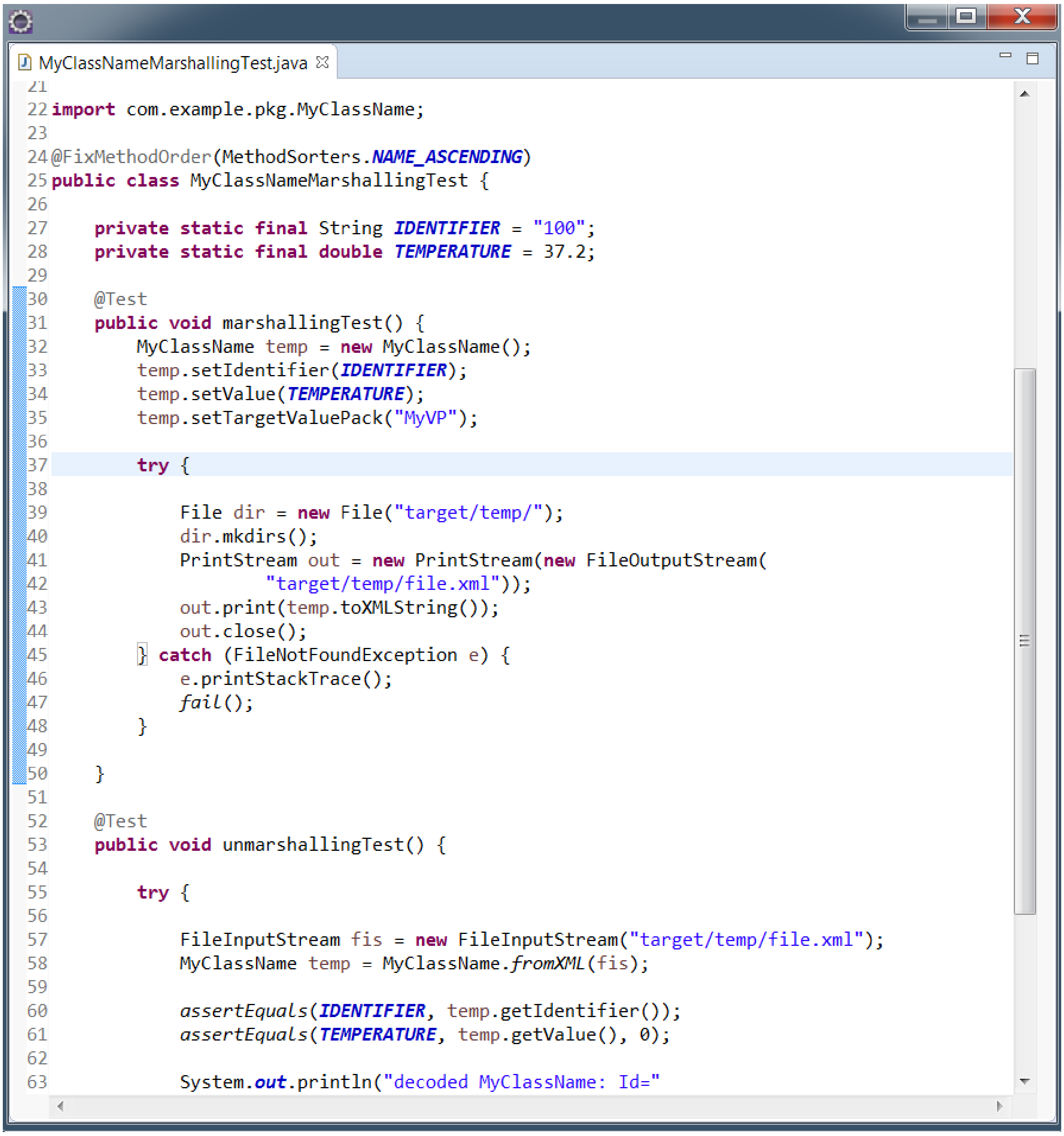
Figure 38 – MyClassName.java Java class of the UMB Event Definition project created in Figure 35

The created UMB Event Definition project also comes with a `pom.xml` file that is used for building and packaging the UMB Event Definition outside of the Eclipse IDE, using Apache Maven.

4.1.2 Validation of the created project

The Event Definition's `src/test` folder contains a Junit test Class `ClassNameMarshallingTest.java`. This is a test skeleton that marshals a "ClassName" object to an XML file and unmarshals the XML file back into a "ClassName" Java object.

This test class is a template that can be extended to test the Event Definition's capabilities.



MyClassNameMarshallingTest.java

```
import com.example.pkg.MyClassName;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class MyClassNameMarshallingTest {

    private static final String IDENTIFIER = "100";
    private static final double TEMPERATURE = 37.2;

    @Test
    public void marshallingTest() {
        MyClassName temp = new MyClassName();
        temp.setIdentifier(IDENTIFIER);
        temp.setValue(TEMPERATURE);
        temp.setTargetValuePack("MyVP");

        try {

            File dir = new File("target/temp/");
            dir.mkdirs();
            PrintStream out = new PrintStream(new FileOutputStream(
                    "target/temp/file.xml"));
            out.print(temp.toXMLString());
            out.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            fail();
        }

    }

    @Test
    public void unmarshallingTest() {

        try {

            FileInputStream fis = new FileInputStream("target/temp/file.xml");
            MyClassName temp = MyClassName.fromXML(fis);

            assertEquals(IDENTIFIER, temp.getIdentifier());
            assertEquals(TEMPERATURE, temp.getValue(), 0);

            System.out.println("decoded MyClassName: Id="
```

Figure 39 - Event DefinitionMarshallingTest.java JUnit test class of the new UMB Event Definition project

4.2 Customizing the serialization Class

The serialization class is the class in charge of serializing (de-serializing) the Event message into (and from) a byte array. The default serialization class is provided by the UMB framework. This class is the `com.hp.umb.adapter.internal.utilities.JavaClassSerializer` class which uses the standard java class serialization mechanism.

A custom serialization class can be used instead of the default one. Such a class must implement the following interfaces:

```
org.apache.kafka.common.serialization.Serializer<Object>
org.apache.kafka.common.serialization.Deserializer<Object>
```

Here is a skeleton of a custom serialization class:

```java
package com.example.mypkg;

import java.util.Map;

import org.apache.kafka.common.serialization.Deserializer;
import org.apache.kafka.common.serialization.Serializer;

public class MySerializer implements Serializer<Object>,
Deserializer<Object> {

    @Override
    public void configure(Map<String, ?> configs, boolean iskey) {
        // TODO Auto-generated method stub
    }

    @Override
    public void close() {
        // TODO Auto-generated method stub
    }

    @Override
    public Object deserialize(String topic, byte[] bytes) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public byte[] serialize(String topic, Object data) {
        // TODO Auto-generated method stub
        return null;
    }
}
```

The use of a custom serialization class needs to be configured on both the producer and consumer sides. It is necessary that both producer and consumer use the same serialization class, otherwise de-serialization will not be performed properly and results will be unpredictable:

- On the Producer side:

  In the flow service description (in the `AdapterConfiguration.xml` file), a custom serialization class can be specified using the `serializerClass` attribute of the `<flow>` XML element.

  For example:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="CollectionTestAdapter" version="1.0"
xmlns="http://hp.com/umb/config">
```

```
    <flowServices>
        <flow name="AFlowWithCustomSerializer"
              type="Dynamic"
              collectorClass="com.hp.umb.adapter.collection.MyCollector"

serializerClass="com.hp.umb.adapter.collection.MySerializer">
            <parameters>
                <parameter key="fileName"
defaultValue="data/alarms.xml"/>
            </parameters>
        </flow>
    </flowServices>
</adapter>
```

- On the Consumer side:

  In the (auto) flow consumer description (in the `AdapterConfiguration.xml` file), a custom serialization class can be specified using the `serializerClass` attribute of the `<autoConsumer>` XML element. Alternatively, a custom serialization class can be specified in Java by using the `setSerializerClass(String)` method on the `ConsumerFlow` object (For example: `myFlow.setSerializerClass("com.hp.umb.adapter.collection.MySerializer");`)

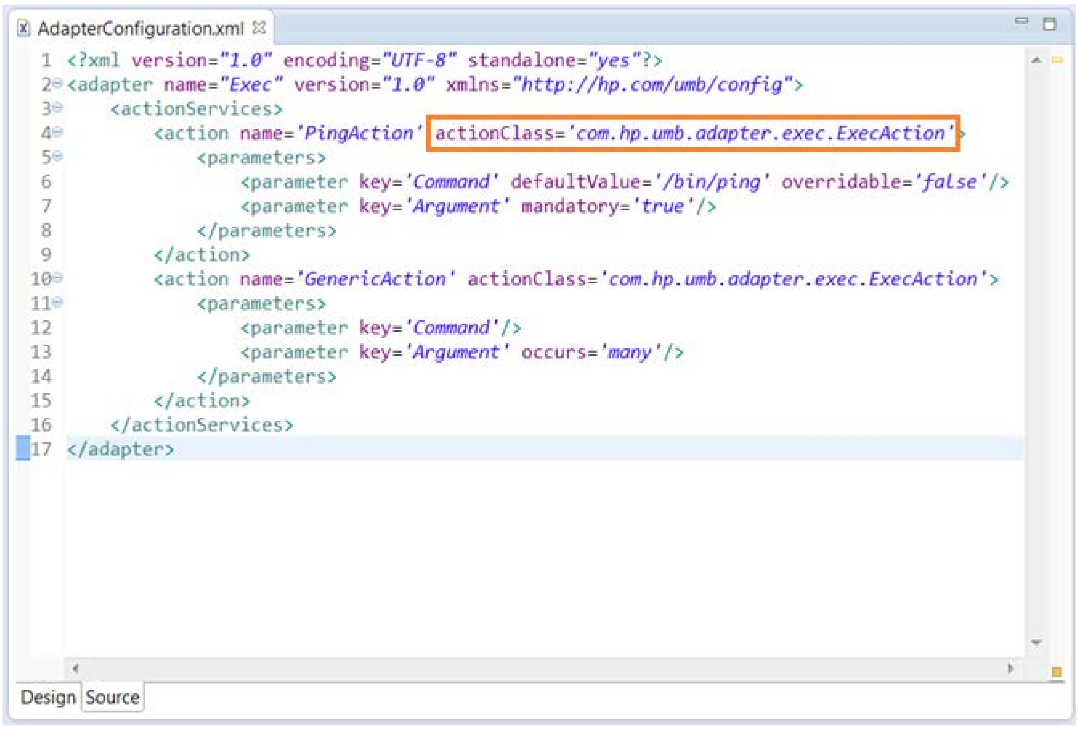
  For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="LogAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <autoConsumers>
         <autoConsumer
           consumerIdentifier="AlarmLogger"
           targetAdapterName="CollectionTestAdapter"
           targetFlowName="AFlowWithCustomSerializer"
           serializerClass="com.hp.umb.adapter.collection.MySerializer"
           messageConsumerClass="com.hp.umb.adapter.log.LogAlarmConsumer"/>
    </autoConsumers>
</adapter>
```

4.3 Lifecycle and thread-safety of user-provided classes

For most UMB Adapters, the UMB Framework requires that the user provides Java classes that give the implementation for specific features of an Adapter. This is the case for:

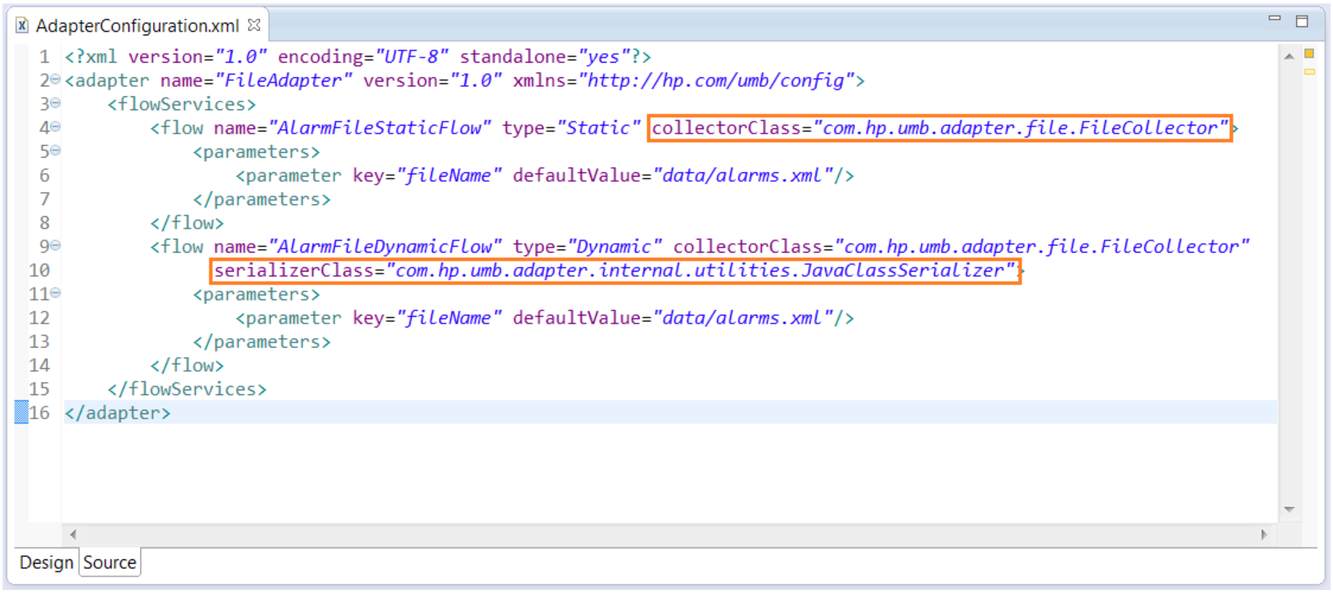
- Action services: the action service provider Adapter developer is required to provide the name of a Java class that implements the action service. The name of the class is defined in the **`actionClass`** attribute of the `action` XML element defining the action service in the `AdapterConfiguration.xml` file



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="Exec" version="1.0" xmlns="http://hp.com/umb/config">
    <actionServices>
        <action name='PingAction' actionClass='com.hp.umb.adapter.exec.ExecAction'>
            <parameters>
                <parameter key='Command' defaultValue='/bin/ping' overridable='false'/>
                <parameter key='Argument' mandatory='true'/>
            </parameters>
        </action>
        <action name='GenericAction' actionClass='com.hp.umb.adapter.exec.ExecAction'>
            <parameters>
                <parameter key='Command'/>
                <parameter key='Argument' occurs='many'/>
            </parameters>
        </action>
    </actionServices>
</adapter>
```

Figure 40 - Example of an actionClass attribute

- Flow services: the flow producer Adapter developer is required to provide the name of a Java class that provides the implementation of each flow service. The name of the class is defined in the **`collectorClass`** attribute of the `flow` XML element defining the flow service in the `AdapterConfiguration.xml` file. Optionally, it is also possible to provide the name of a Java class to serialize/de-serialize flow messages, in case the default serialization class (`com.hp.umb.adapter.internal.utilities.JavaClassSerializer`) is not wanted. The name of this class is defined in the **`serializerClass`** attribute of the `flow` XML element defining the flow service in the `AdapterConfiguration.xml` file.

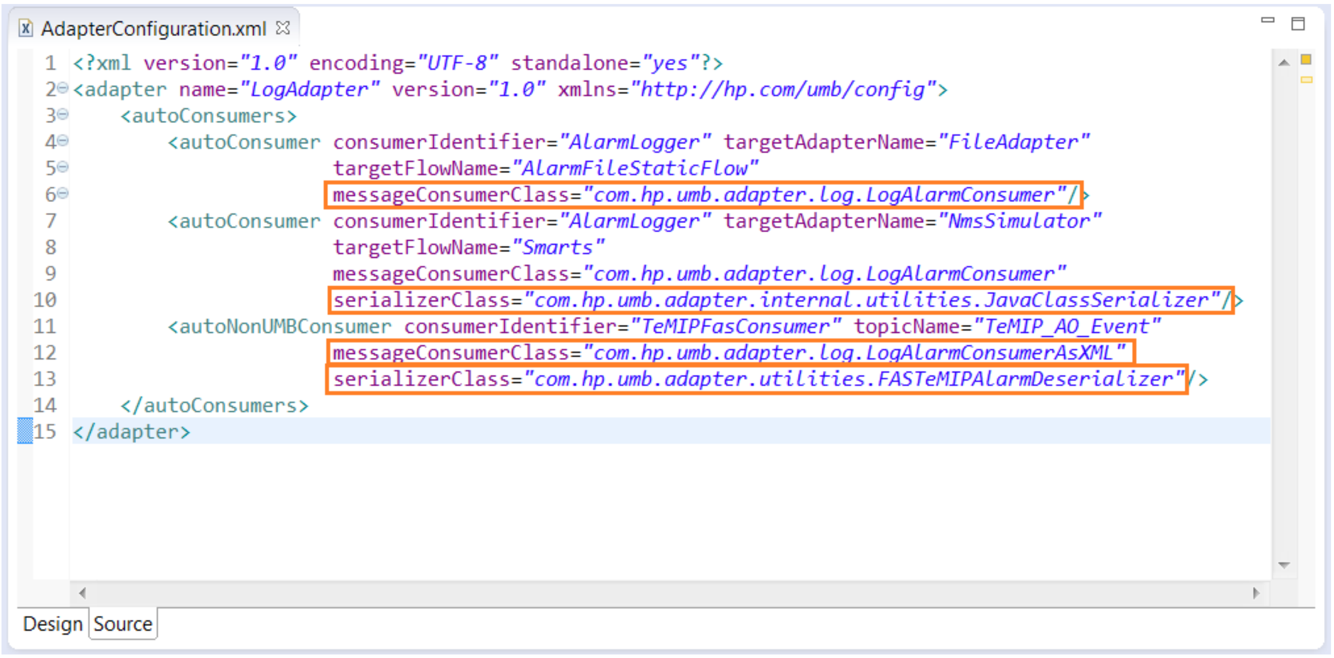


```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="FileAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <flowServices>
        <flow name="AlarmFileStaticFlow" type="Static" collectorClass="com.hp.umb.adapter.file.FileCollector">
            <parameters>
                <parameter key="fileName" defaultValue="data/alarms.xml"/>
            </parameters>
        </flow>
        <flow name="AlarmFileDynamicFlow" type="Dynamic" collectorClass="com.hp.umb.adapter.file.FileCollector"
            serializerClass="com.hp.umb.adapter.internal.utilities.JavaClassSerializer">
            <parameters>
                <parameter key="fileName" defaultValue="data/alarms.xml"/>
            </parameters>
        </flow>
    </flowServices>
</adapter>
```

Figure 41 - Example of a collectorClass and a serializerClass attribute

- Automatic flow consumers: Both standard (UMB) and non UMB flows can have automatic flow consumers. The automatic flow consumers are defined in the `AdapterConfiguration.xml` file of the flow consumer

Adapter, using either the `autoConsumer` or the `autoNonUMBConsumer` XML element. In order to define an automatic flow consumer, the developer is required to provide the name of the Java class that provides the implementation of the consumer. The name of the class is defined in the **`messageConsumerClass`** attribute of the `autoConsumer` or `autoNonUMBConsumer` XML element defining the automatic consumer in the `AdapterConfiguration.xml` file. Optionally, it is also possible to provide the name of a Java class to serialize/de-serialize flow messages, in case the default serialization class (`com.hp.umb.adapter.internal.utilities.JavaClassSerializer`) is not wanted. The name of this class is defined in the **`serializerClass`** attribute of the `autoConsumer` or `autoNonUMBConsumer` XML element defining the automatic consumer in the `AdapterConfiguration.xml` file.



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="LogAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <autoConsumers>
        <autoConsumer consumerIdentifier="AlarmLogger" targetAdapterName="FileAdapter"
                      targetFlowName="AlarmFileStaticFlow"
                      messageConsumerClass="com.hp.umb.adapter.log.LogAlarmConsumer"/>
        <autoConsumer consumerIdentifier="AlarmLogger" targetAdapterName="NmsSimulator"
                      targetFlowName="Smarts"
                      messageConsumerClass="com.hp.umb.adapter.log.LogAlarmConsumer"
                      serializerClass="com.hp.umb.adapter.internal.utilities.JavaClassSerializer"/>
        <autoNonUMBConsumer consumerIdentifier="TeMIPFasConsumer" topicName="TeMIP_AO_Event"
                      messageConsumerClass="com.hp.umb.adapter.log.LogAlarmConsumerAsXML"
                      serializerClass="com.hp.umb.adapter.utilities.FASTeMIPAlarmDeserializer"/>
    </autoConsumers>
</adapter>
```

Figure 42 - Example of a messageConsumerClass and a serializerClass attribute

Instances of Java objects from classes defined as either **`actionClass`**, **`collectorClass`**, **`messageConsumerClass`**, or **`serializerClass`** in the `AdapterConfiguration.xml` file are handled by the UMB Framework.

In the following chapters, we will detail the lifecycle of these objects as well as their concurrency/thread-safety requirements/characteristics.

4.3.1 Lifecycle and thread-safety of actionClass objects

The following schema describes the lifecycle of actionClass objects:

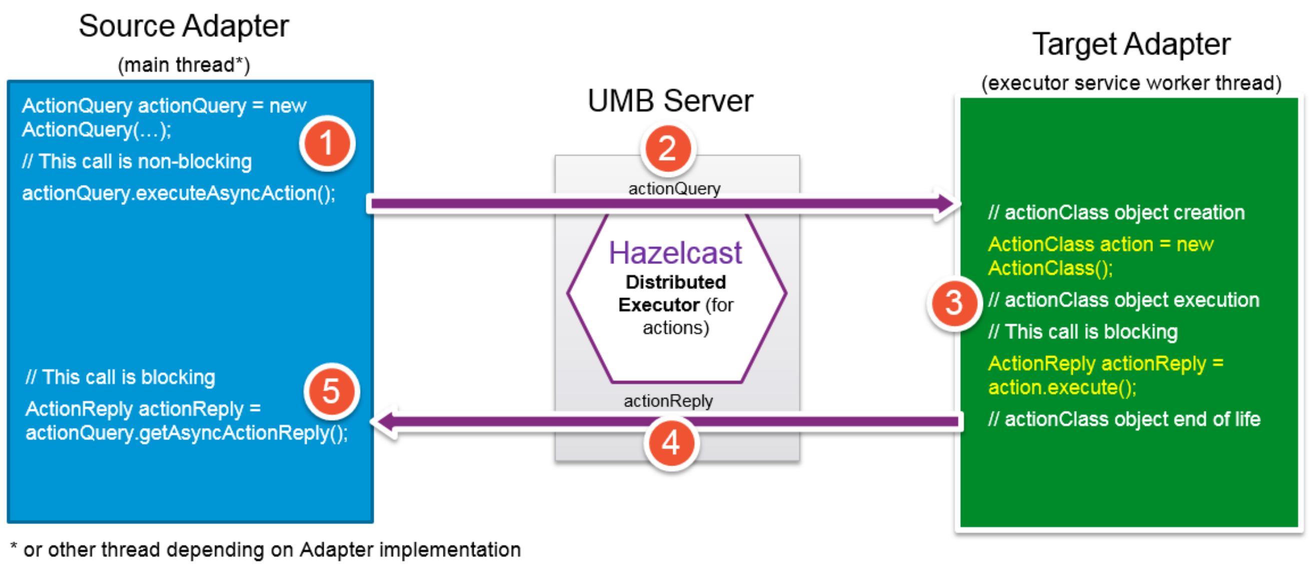

Source Adapter
(main thread*)
ActionQuery actionQuery = new ActionQuery(…);
// This call is non-blocking
actionQuery.executeAsyncAction();
1
// This call is blocking
ActionReply actionReply = actionQuery.getAsyncActionReply();
5
UMB Server
2
actionQuery
Hazelcast
Distributed Executor (for actions)
actionReply
4
Target Adapter
(executor service worker thread)
// actionClass object creation
ActionClass action = new ActionClass();
3
// actionClass object execution
// This call is blocking
ActionReply actionReply = action.execute();
// actionClass object end of life
* or other thread depending on Adapter implementation

Figure 43 - Lifecycle of actionClass objects

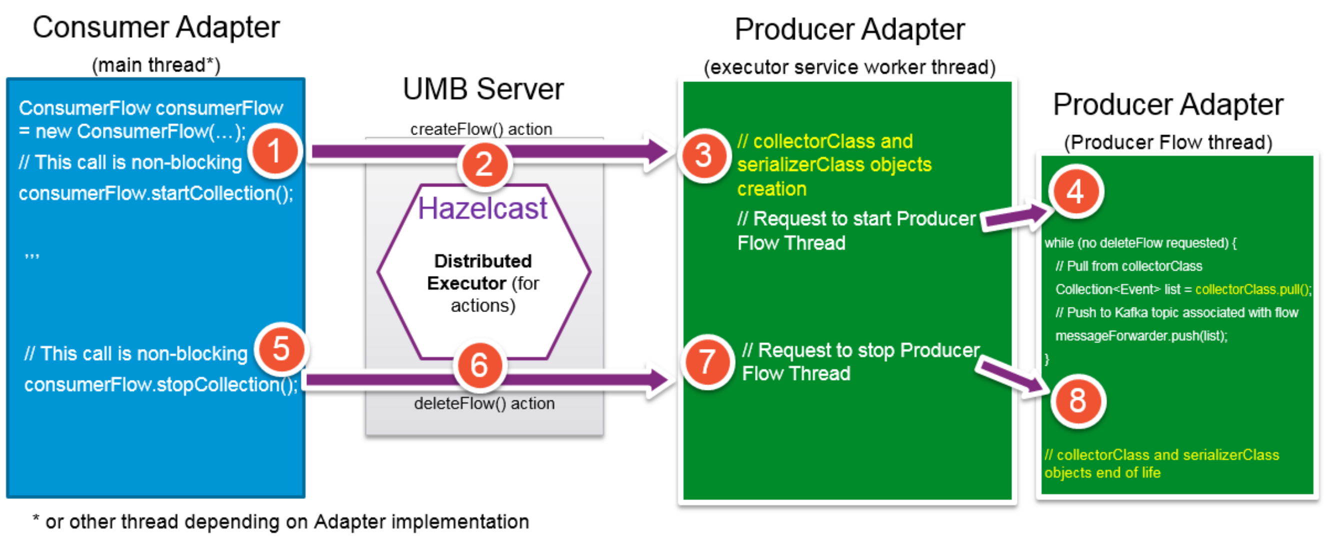
In the above schema, there are 2 adapters: 1 source Adapter that will send action execution requests, and 1 target adapter that will execute the requested actions. Executing actions in UMB is a 5 step process:

1. In step 1, on the source Adapter, an ActionQuery object is created. This is typically done in the main thread of the source Adapter but this can be done in another thread depending on the implementation of the source Adapter. The ActionQuery object contains information on the source and target adapters, the name of the action to execute on the target adapter and the action execution parameters. Then the ActionQuery is executed (in the above schema, the ActionQuery is executed asynchronously).
2. In step 2, the UMB framework and more precisely the Hazelcast distributed executor service component has taken over and transmitted the ActionQuery from the source Adapter to the target Adapter for execution.
3. In step 3, the ActionQuery is executed on the target Adapter. This is done is one of the Hazelcast Executor Service worker threads. Hazelcast defines a pool of threads to use for the Executor Service. This pool of threads is configured in the `<executor-service><pool-size>7</pool-size>…</executor-service>` section of the `hazelcast.xml` file of the target Adapter. In step 3, the actionClass object is created, then the `execute()` method is called on this object to actually execute the requested action. Execution of the `execute()` method returns an ActionReply object that represents the result of the action execution. The end of the execution of the `execute()` method marks the end of life for the actionClass object previously created that can now be garbage collected.
4. In step 4, the UMB framework transmits the ActionReply from the target Adapter to the source Adapter.
5. In step 5, the execution that was blocked on the `actionQuery.getAsyncActionReply()` call resumes and the ActionReply is retrieved.

To summarize, actionClass object are created when execution starts on the target Adapter (inside an Hazelcast Executor Service worker thread). These objects live until the execution stops and then they are ready to be garbage collected. ActionClass objects are not re-used. There's no requirement regarding thread-safety for the actionClass classes as execution will always occur in a single thread and execution occurs only once and then the actionClass object is garbage collected.

4.3.2 Lifecycle and thread-safety of collectorClass and serializerClass producer flow objects

This chapter describes the lifecycle and thread-safety of collectorClass and serializerClass producer flow objects.



```
ConsumerFlow consumerFlow
= new ConsumerFlow(…);
// This call is non-blocking
consumerFlow.startCollection();

'''

// This call is non-blocking
consumerFlow.stopCollection();
```

```
// collectorClass and
serializerClass objects
creation
// Request to start Producer
Flow Thread

// Request to stop Producer
Flow Thread
```

```
while (no deleteFlow requested) {
  // Pull from collectorClass
  Collection<Event> list = collectorClass.pull();
  // Push to Kafka topic associated with flow
  messageForwarder.push(list);
}

// collectorClass and serializerClass
objects end of life
```

Figure 44 - Lifecycle of collectorClass and serializerClass producer flow objects

In the above schema, there are 2 adapters: 1 Consumer Adapter that will send requests to create and (later on) delete a flow of alarms/events provided by the Producer Adapter, and 1 Producer Adapter that will execute the requested actions and actually create or delete the corresponding flow.

Creating a flow in UMB is a 4 step process:

1. In step 1, on the Consumer Adapter, a ConsumerFlow object is created and the `startCollection()` method is called on this object. This is typically done in the main thread of the Consumer Adapter but this can be done in another thread depending on the implementation of the Consumer Adapter. The call to the `startCollection()` method triggers a createFlow() action request to be sent to the Producer Adapter
2. In step 2, the UMB framework and more precisely the Hazelcast distributed executor service component has taken over and transmitted the createFlow() action request from the Consumer Adapter to the Producer Adapter for execution.
3. In step 3, the createFlow() action is executed on the Producer Adapter. This is done is one of the Hazelcast Executor Service worker threads. Hazelcast defines a pool of threads to use for the Executor Service. This pool of threads is configured in the `<executor-service><pool-size>7</pool-size>…</executor-service>` section of the `hazelcast.xml` file of the Producer Adapter. In step 3, the collectorClass and serializerClass producer flow objects are created and a new Producer Flow Thread is created to handle the collection of alarms/events of the flow.
4. In step 4, the Producer Flow Thread starts and executes a loop until a deleteFlow() action request is received by the Producer Adapter. At each run of the loop, the Producer Flow Thread calls the `pull()` method on the collectorClass object and then forwards the list of alarms/events thus retrieved to the Kafka topic associated with the flow.

Deleting a flow in UMB is a 4 step process too:

5. In step 5, on the Consumer Adapter, the `stopCollection()` method is called on the ConsumerFlow object. This is typically done in the main thread of the Consumer Adapter but this can be done in another thread depending on the implementation of the Consumer Adapter. The call to the `startCollection()` method triggers a deleteFlow() action request to be sent to the Producer Adapter

6. In step 6, the UMB framework and more precisely the Hazelcast distributed executor service component has taken over and transmitted the deleteFlow() action request from the Consumer Adapter to the Producer Adapter for execution.
7. In step 7, the deleteFlow() action is executed on the Producer Adapter. This is done is one of the Hazelcast Executor Service worker threads. Hazelcast defines a pool of threads to use for the Executor Service. This pool of threads is configured in the `<executor-service><pool-size>7</pool-size>…</executor-service>` section of the `hazelcast.xml` file of the Producer Adapter. In step 7, a request is sent to the Producer Flow Thread to stop the collection of alarms/events of the flow.
8. In step 4, the Producer Flow Thread stops. The end of the execution of the Producer Flow Thread marks the end of life for the collectorClass and serializerClass objects previously created that can now be garbage collected.

To summarize, collectionClass and serializerClass producer flow objects are created when flows are created on the Producer Adapter (inside an Hazelcast Executor Service worker thread) upon a createFlow() request from a Consumer Adapter. These objects live until the corresponding flows are deleted upon a deleteFlow() request from a Consumer Adapter and then they are ready to be garbage collected.

CollectorClass and serializerClass objects are not re-used. Each flow creation leads to new collectionClass and serializerClass objects. For example, if a flow is created, deleted, then re-created again, this will lead to a first set of collectionClass/serializerClass objects being created then garbage collected and then a second set of collectionClass/serializerClass objects being created.

There should be some level of thread-safety for collectorClass classes as execution may occur concurrently in multiple threads. If requests for createFlow(), deleteFlow(), auditFlow() or resynchFlow() are sent concurrently to the Producer Adapter, then multiple threads (from the Executor Service thread-pool of the Producer Adapter) may concurrently execute methods from the CollectorClass. These methods being:

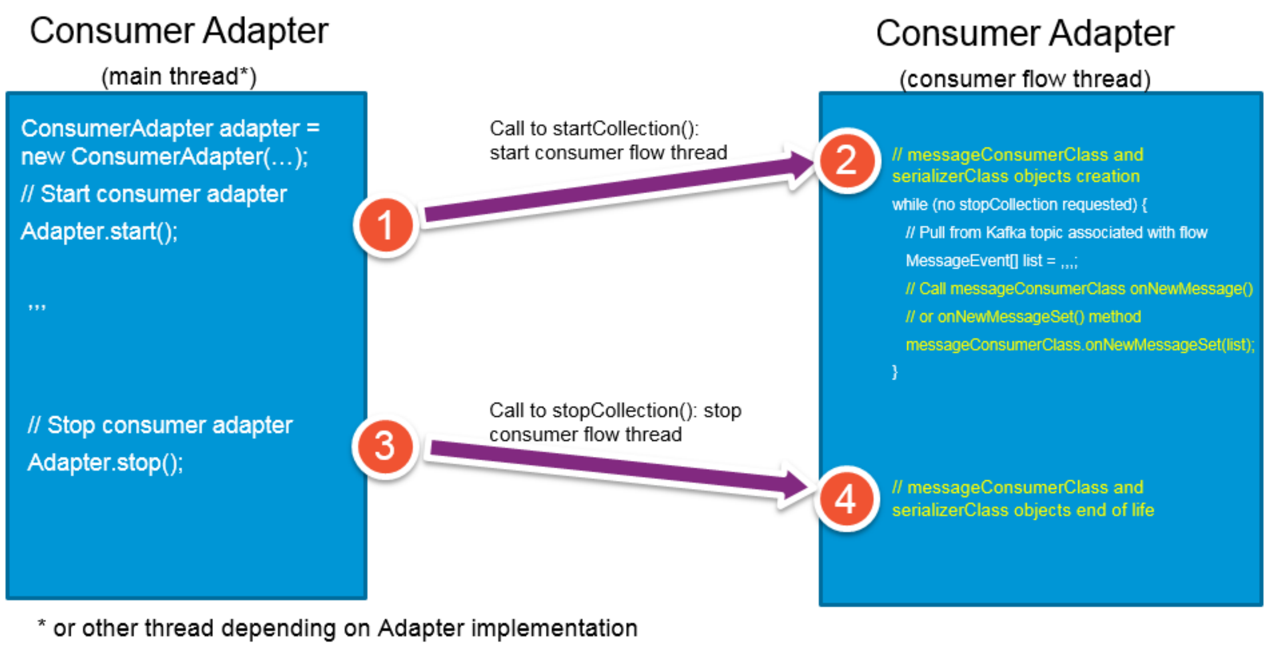
- `onInitialization()` and `startCollection()` on createFlow() requests
- `onDestruction()` and `stopCollection()` on deleteFlow() requests
- `resynchCollection()` on resynchFlow() requests
- `getCollectionAudit()` on auditFlow() requests

In addition to the Executor Service worker threads, the Producer Flow Thread also accesses the collectorClass objects as it periodically calls the `pull()` method.

On the other hand, there's no requirement regarding thread-safety for serializerClass classes as execution will always occur in a single thread, the Producer Flow Thread.

4.3.3 Lifecycle and thread-safety of messageConsumerClass and serializerClass consumer flow objects

This chapter describes the lifecycle and thread-safety of messageConsumerClass and serializerClass consumer flow objects.


Consumer Adapter
(main thread*)
Consumer Adapter
(consumer flow thread)
1
Call to startCollection():
start consumer flow thread
2
3
Call to stopCollection(): stop
consumer flow thread
4
* or other thread depending on Adapter implementation

```
ConsumerAdapter adapter =
new ConsumerAdapter(…);
// Start consumer adapter
Adapter.start();

,,,

// Stop consumer adapter
Adapter.stop();
```

```
// messageConsumerClass and
serializerClass objects creation
while (no stopCollection requested) {
   // Pull from Kafka topic associated with flow
   MessageEvent[] list = ,,,;
   // Call messageConsumerClass onNewMessage()
   // or onNewMessageSet() method
   messageConsumerClass.onNewMessageSet(list);
}

// messageConsumerClass and
serializerClass objects end of life
```

In the above schema, there is just 1 adapter, the Consumer Adapter that will create consumer flows as defined in the `AdapterConfiguration.xml` file at adapter start and stop these flows at adapter stop.

The lifecycle of a consumer flow (whether it's a UMB or non UMB consumer flow) in UMB is a 4 step process:

1. In step 1, on the Consumer Adapter, a ConsumerFlow object is created and started by calling the `startCollection()` method on all consumerFlow instance. This call triggers the creation and start of a Consumer Flow Thread for each consumer flow.
2. In step 2, a Consumer Flow Thread is started which triggers the creation of the messageConsumerClass and serializerClass objects. The Consumer Flow Thread then executes a loop until a stopCollection() call is made. At each run of the loop, the Consumer Flow Thread pulls messages from the Kafka topic associated with the flow.
3. Step 3 does not directly follow step 2. It happens later on when the consumer Flow is stopped via a call to the `stopCollection()` metho. This call triggers the request to stop the Consumer Flow .
4. In step 4, the Consumer Flow Thread is stopped, which marks the end of life for the messageConsumerClass and serializerClass objects previously created that can now be garbage collected.

To summarize, messageConsumerClass and serializerClass consumer flow objects are created when the ConsumerFlow is created and the collection starts (inside an Consumer Flow Thread) upon a createCollection() call. These objects live until the ConsumerFlow is stopped via a call to stopCollection() and then these objects are ready to be garbage collected.

MessageConsumerClass and serializerClass objects are not re-used. Each consumer flow creation leads to new messageConsumerClass and serializerClass objects

There's no requirement regarding thread-safety for the messageConsumerClass and serializerClass classes as execution will always occur in a single thread, the Consumer Flow Thread.

4.4 Retrieving information on Adapters in the solution/cluster

The UMB Framework offers the possibility for an adapter to be aware of other (remote) adapters that are part of the solution/cluster (i.e. accessible through Hazelcast).

This is done mainly by:

1. Getting the list of known remote adapters
2. Getting notifications when adapters are added/removed from the solution/cluster or when the state of the adapters in the cluster/solution changes.

4.4.1 Getting the list of known Adapters

The `com.hp.umb.adapter.BaseAdapter` class (and thus any custom Adapter class) provides the `getAdapterLimitedProxyMap()` method. This method returns a map of all adapters that are currently connected to the distributed UMB solution.

This method can be used as follows:

```
// Log all Adapters already present in the cluster
for (Map.Entry<String, ? extends AdapterProxyLimited> entry :
                    adapter.getAdapterLimitedProxyMap().entrySet()) {
    AdapterProxyLimited remoteAdapter = entry.getValue();
    log.info("Adapter : " + remoteAdapter.getName()+ " : "
                    + remoteAdapter.getState());
}
```

4.4.2 Getting notifications on Adapters

Adapter notifications are emitted in either of the following conditions:

- An adapter has started
- An adapter has stopped
- An adapter's state has changed (from STARTING to ACTIVE or from ACTIVE to STOPPING)

In order to receive Adapter notifications, you first need to create a new class that implements the `com.hp.umb.adapter.configuration.AdapterProxyListenerInterface` as shown below:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.hp.umb.adapter.configuration.AdapterProxyEvent;
import com.hp.umb.adapter.configuration.AdapterProxyListenerInterface;

public class TestAdapterProxyListener implements
AdapterProxyListenerInterface {

    private static final Logger log = LoggerFactory
            .getLogger(TestAdapterProxyListener.class);

    @Override
    public void entryAdded(AdapterProxyEvent adapterProxyEvent) {
        // TODO write custom code here
        log.info("Proxy Listener: Adapter Proxy Entry added : "
                + adapterProxyEvent.getName() + " : "
                + adapterProxyEvent.getState());
    }

    @Override
    public void entryRemoved(AdapterProxyEvent adapterProxyEvent) {
        // TODO write custom code here
        log.info("Proxy Listener: Adapter Proxy Entry removed : "
                + adapterProxyEvent.getName().toString());
    }

    @Override
    public void entryUpdated(AdapterProxyEvent adapterProxyEvent) {
        // TODO write custom code here
        log.info("Proxy Listener: Adapter Proxy Entry updated : "
                + adapterProxyEvent.getName() + " : "
                + adapterProxyEvent.getState());
    }

}
```

Then you need to associate an instance of this new class as an AdapterProxyListener for your Adapter. This is done using the `addAdapterProxyListener()` method of the `com.hp.umb.adapter.BaseAdapter` class (which any Adapter extends):

```
// Register an Adapter listener
try {
    adapter.addAdapterProxyListener(new TestAdapterProxyListener());
} catch (AdapterNotActiveException e1) {
    log.error("Failed to add Proxy Listener", e1);
 }
```

In the above code sample the `adapter` variable represents your Adapter instance.

4.5 Multiple Kafka/ZooKeeper clusters configuration

There are cases where it is necessary to not just use one Kafka/ZooKeeper cluster but several. With UMB, it is possible to design solutions where some flows are produced to one Kafka/ZooKeeper cluster and some flows are produced to another. There's no limit to the number of Kafka/ZooKeeper cluster that UMB can use.

The first step in a multi Kafka/ZooKeeper cluster UMB solution is to define all the clusters to use in the `adapter.properties` file of each UMB Adapter that needs to work with multiple Kafka/ZooKeeper clusters.

The `producer.*` and `consumer.*` Kafka/ZooKeeper properties in the `adapter.properties` file can be prefixed by a Kafka/ZooKeeper cluster specific prefix so that each Kafka/ZooKeeper cluster has its own properties. There's no pattern to follow for the name of the prefix. It can be anything. For example, the prefix can refer to the name or id of the Kafka/ZooKeeper cluster: `kafka1`, `kafka2`, etc… Any chosen prefix for a Kafka/ZooKeeper cluster will have to match the prefix used in the `AdapterConfiguration.xml` file, when we define that a flow targets a specific Kafka/ZooKeeper cluster using the `broker` attribute of the `flow` XML element (see later on for an example `AdapterConfiguration.xml` file that demonstrates this feature).

Below is an example of an `adapter.properties` file that defines the properties of two separate Kafka/ZooKeeper clusters (identified by their prefixes as `kafka1` and `kafka2`):

```
# Properties specific to "kafka1" Kafka/ZooKeeper cluster
kafka1.producer.bootstrap.servers=brokerhost1:9092
kafka1.zookeeper.connect=brokerhost1:2181

# Properties specific to "kafka2" Kafka/ZooKeeper cluster
kafka2.producer.bootstrap.servers=brokerhost2:9092
kafka2.zookeeper.connect=brokerhost2:2181
```

In order for an adapter to produce a flow to a specific Kafka/ZooKeeper cluster there are two steps to follow:

1. In the `AdapterConfiguration.xml` file, it is necessary to set the value of the optional `broker` attribute for each flow to indicate that the flow will be produced to a specific Kafka/ZooKeeper cluster.

   If the optional `broker` attribute is not set, it is understood that the flow will target the "default" Kafka/ZooKeeper cluster which properties are not be prefixed (with a Kafka/ZooKeeper id/name) in the `adapter.properties` file.

   For example:

   ```
   <adapter name="Adapter1" version="1.0" xmlns="http://hp.com/umb/config">
       <flowServices>
           <flow name="StaticFlow1" type="Static" broker="kafka1" […]>
           </flow>
           <flow name="StaticFlow2" type="Static" broker="kafka2" […]>
           </flow>
     </flowServices>
   </adapter>
   ```

2. As mentioned above, in the `adapter.properties` file, it is necessary to use the value previously set as prefix for all the properties that belong to a specific Kafka/ZooKeeper cluster.
   For example:

   ```
   kafka1.producer.bootstrap.servers= brokerhost1:9092
   kafka1.zookeeper.connect= brokerhost1:2181
   ```

In the `adapter.properties` file, it is possible to have both Kafka/ZooKeeper cluster specific properties (i.e. properties prefixed with a Kafka/ZooKeeper cluster id/name) and also general properties that will be used for all Kafka/ZooKeeper clusters (i.e. properties that are not prefixed with a Kafka/ZooKeeper cluster id/name).

4.6 Make an adapter compliant with the 'umb' command-line administration tool

UMB V1.1 brings the '`umb`' command-line administration tool. This tools enables users to start, stop and control an UMB adapter from a command-line Interface. In order for your adapter to work correctly with this administration tool, you need to understand how to configure your adapter's `MANIFEST.MF` or `startup.conf` files.

Properly configuring these files is the key to letting the '`umb`' command-line tool know how to start your adapter. At the minimum, the '`umb`' command-line tool needs to know what the main class of the Adapter is: i.e. the class that contains the `main(String[] args)` method used to start your Adapter.

There are several ways to specify the main class of your adapter:

- The easiest way is to deliver your Adapter as a single jar file with the main-class specified in the Adapter's jar's `MANIFEST.MF` file.
- Another way (if your UMB adapter brings multiple jar files for example) is to deliver a `startup.conf` configuration file

A `startup.conf` file defines the main class and JVM options to be used by the '`umb`' administration tool when starting the adapter.

An example of contents of the `startup.conf` file is (the content below is from the `startup.conf` file of the TeMIP Adapter):

```
MAIN_CLASS=com.hp.umb.adapter.temip.TemipAdapter

MAIN_JAR=umb-adapter-temip-1.1.jar

JAVA_OPTS=-Xmx1024m

REFERENCE=temip-adapter

ADAPTER_ARGUMENTS="Put your adapter arguments here"
```

The '`umb`' administration tool will use:

- MAIN_CLASS as the class to be called for starting the adapter
- MAIN_JAR (in case MAIN_CLASS is not set) as the jar to be called for starting the adapter (this jar is expected to contain a `MANIFEST.MF` file defining the main-class of the jar)
- JAVA_OPTS as optional Java options to be used when starting the adapter
- REFERENCE as the adapter's directory name (the one under $UMB_HOME/adapters) so that the 'umb' command-line tool can know where to find the adapters libraries
- ADAPTER_ARGUMENTS as the adapter program arguments (the arguments to pass to the `main(String[] args)` method of the main class). By default no arguments are required, but depending on some specific configuration needs it may be useful to pass arguments to the adapter's main class.

NOTE:
As any other configuration file, the `startup.conf` file should be delivered in the `conf/` sub-folder of the adapter.

4.7 Specifying a custom Partitioner

Starting with version 1.1, UMB is capable of managing Kafka topics that use more than 1 partition. It is therefore possible to specify a custom 'Partitioner' for an adapter or for a given flow. A 'partitioner' is a class that extends the `kafka.producer.Partitioner` interface and allows for customizing the way the partition to use is selected for any given message.

With the default 'Partitioner' class, **the message Object itself** is used as the key for partitioning. The partition to use is computed using the key `hashcode()`. This method is akin to a pseudo-random association of messages to partitions.

The Partioner can be customized if there is the need to target specific partitions for specific messages.

For example:

We have the following *Event* Object and we have a topic with a partition for every possible cityCode value:

```
public class MyEvent extends Event {

    int cityCode = 0;

    public int getCityCode(){ return cityCode;}

 }
```

And we want to send events with different cityCodes to different partitions. In order to do so, you need to perform the following steps:

1) Create a custom Partitioner:

A custom Partitioner is a class that implements the `kafka.producer.Partitioner` interface:

```
public class MyPartitioner implements Partitioner {

    @Override
    public int partition(Object key, int numPartitions) {

        // get the key as a MyEvent object and retrieve the
       // partition corresponding to the cityCode
        if (key instanceof MyEvent) {
             targetPartition = ((MyEvent)key).getCityCode()
       }

      // the partition where the message will be sent
        return targetPartition % numPartitions;
    }
 }
```

The Partitioner class computes the target partition based on the key object (in our case the Event itself) and the number of available partitions. With UMB, the key is the message object itself, thus any field of the message can be used to compute the target partition.

2) Specify the custom Partitioner to use in the `adapter.properties` file of the flow producer Adapter:

This can be done globally (for all flows defined for the producer Adapter)

```
producer.partitioner.class=com.hp.mypackage.MyPartitioner
```

or on a per-broker basis by using the 'broker' feature (in which case, the Partitioner class will be used for all flows that use a specific broker, 'broker1' in our case)

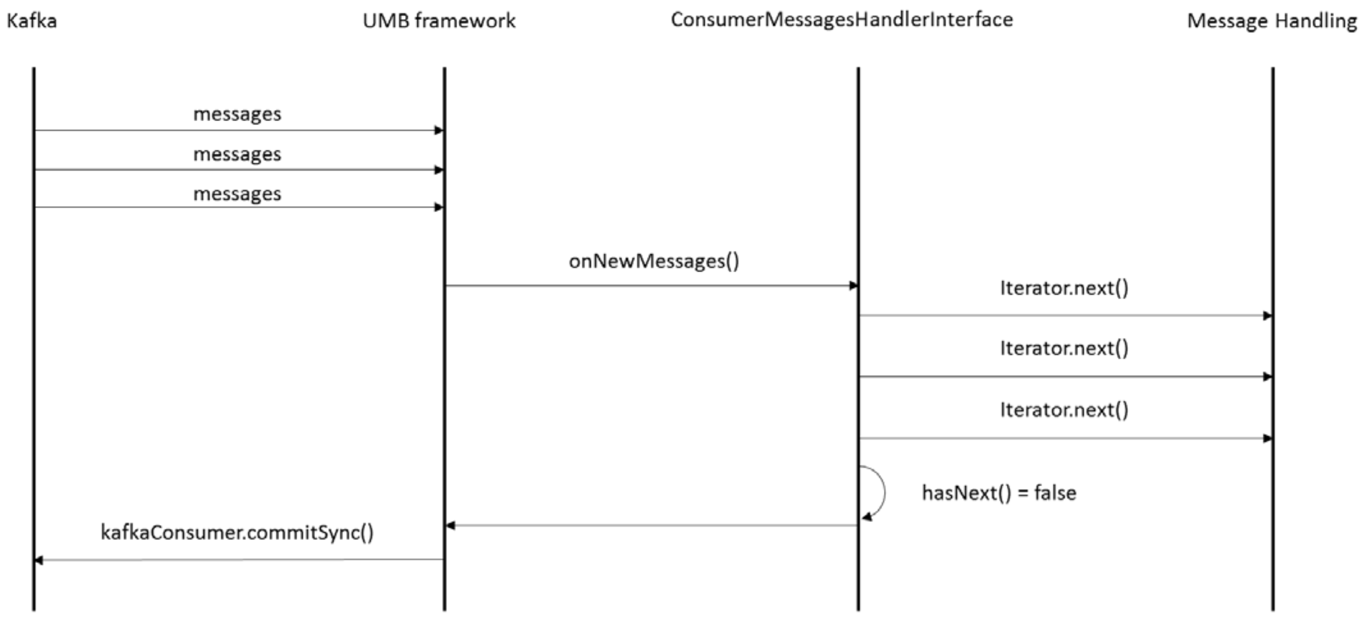
```
broker1.producer.partitioner.class=com.hp.mypackage.MyPartitioner
```

4.8 Using the manual Commit feature

The committed position is the last offset that has been securely consumed by a flow consumer. Should the consumer process fail and restart, this is the offset that it will restart from when the consumer '`readOffsetValue`' property is configured to '`earliest`'.

When automatic commit is configured (the commit feature is automatic by default), the UMB framework commits all messages once they have been consumed by the `onNewMessages()` method of the class extending the `BaseConsumerMessageHandler` class.

The call diagram of the automatic commit feature is the following:



This default behavior can be de-activated in order to give the class extending the `BaseConsumerMessageHandler` class a better control on message commitment.

The manual commit feature can be activated either on the autoConsumer definition (in the `AdapterConfiguration.xml` file on the consumer Adapter) or via a consumer flow API:

- Setting the `manualCommit` attribute to `true` in the `autoConsumer` definition:

```
<autoConsumer consumerIdentifier="AlarmLogger" targetAdapterName="FileAdapter"
```

```
               targetFlowName="AlarmFileStaticFlow" manualCommit="true"
               messageConsumerClass="com.hp.umb.adapter.log.LogAlarmConsumer"/>
```

- Calling the `setManualCommit(true)` method from the consumer flow API:

```
ConsumerFlowInterface consumerFlow = new ConsumerFlow(
                    adapterConsumer, "MyConsumer", "FileAdapter",
                    "AlarmFileStaticFlow ", new ArrayList<Parameter>(),
                    new ConsumerSetCommitAllPartitions(), true);
consumerFlow.setManualCommit(true);
```

When the automatic commit feature is activated, the UMB framework does not make any commits. It is therefore the responsibility of the message consumer's custom Java code to commit the messages.

There are two methods on the `BaseConsumerFlow` class that allows for committing messages:

- `commit(BasicEvent message)`

When this method is used, only the partition that the message was consumed from is commited. The method commits all the messages read from this partition up to the message given as parameter.

A typical use of this API is as follows:

```
   @Override
    public void onNewMessages(MessagesIterator<BasicEvent> iterator) {
        while (iterator.hasNext()) {
            BasicEvent message = iterator.next();

            log.info("reading message: partition=" + message.getPartition()
                    + "  offset=" + message.getOffset());

            PERFORM SOME CUSTOME CODE HERE

            if ( SOME CONDITION ) {
                log.info("committing message: partition="
                        + message.getPartition() + "  offset="
                        + message.getOffset());
                getFlow().commit(message);
            }
        }
    }
```

Note: for flows with a single partition, all consumed messages are committed.

- `commitAllPartitions()`

This method makes sense for flows configured with several partitions. It allows for committing all consumed events regardless of their partitions. An event is considered as "consumed" as soon as it has been retrieved using the iterator `next()` method.

For example:

```
   @Override
    public void onNewMessages(MessagesIterator<BasicEvent> iterator) {
        while (iterator.hasNext()) {
            BasicEvent message = iterator.next();

            log.info("reading message: partition=" + message.getPartition()
                    + "  offset=" + message.getOffset());

            PERFORM SOME CUSTOME CODE HERE
            if ( SOME CONDITION ) {
                log.info("committing all consumed messages in all partitions’’);
                getFlow().commitAllPartitions();
            }
        }
    }
```

Note: for a flow with a single partition, calling either the `commit(message)` or the `commitAllPartitions()` method is equivalent.

Chapter 5
Unified Mediation Bus sample Adapters

The UMB Adapter Development Kit provides sample Adapters that can be used as examples to create your own Adapters. These sample Adapters are located in the **`${UMB_DEV_HOME}`**`/adapter-examples` folder on Linux, **`%UMB_DEV_HOME%`**`\adapter-examples` folder on Windows.

5.1 Camel Adapter

The Camel Adapter is an example adapter that demonstrates how a UMB Adapter can be integrated with Camel[3] in order to benefit from the power and versatility of Camel inside an Adapter. This adapter acts both as a Flow and Action service provider.

As a Flow provider, the adapter will:

- respond to collection flow actions: `CreateFlow`, `DeleteFlow`, `ResynchFlow`, `StatusFlow`
- as a consequence of these collection flow actions, the adapter will create/delete/resynchronize or get the status of collections of alarms/events

As an Action provider, the adapter will:

- respond to action requests

The Camel Adapter is composed of:

- Configuration files:
    - o The Adapter properties file: `adapter.properties` that defines properties for the adapter including connection information for Kafka/ZooKeeper
    - o The Adapter's Hazelcast configuration file: `hazelcast.xml` that defines how to connect to the UMB Hazelcast Central Repository
    - o The Adapter's Log4j configuration file: `log4j.xml`
    - o The Adapter configuration file: `AdapterConfiguration.xml` that defines the flows and actions provided by the adapter
    - o A Camel Spring file: `camel-context.xml` that defines routes to be used for processing actions, collection flow actions and collections
- Java files that define the Adapter's behavior
- A JUnit test file that tests the Adapter's behaviour: `CamelAdapterTest.java`

The Adapter uses the Camel Spring API[4] instead of the Camel Java API[5] because it provides the ability to modify the Camel routes in the `camel-context.xml` file without having to recompile the Adapter. It is possible to use the Camel Java API instead inside a UMB Adapter however this is not part of this example.

---

[3] Please see http://camel.apache.org/ for more information on Camel

[4] Please see: http://camel.apache.org/spring.html for more information on the Camel Spring DSL

[5] Please see http://camel.apache.org/java-dsl.html for more information on the Camel Java DSL

The following figure explains the overall architecture of the Camel Adapter.

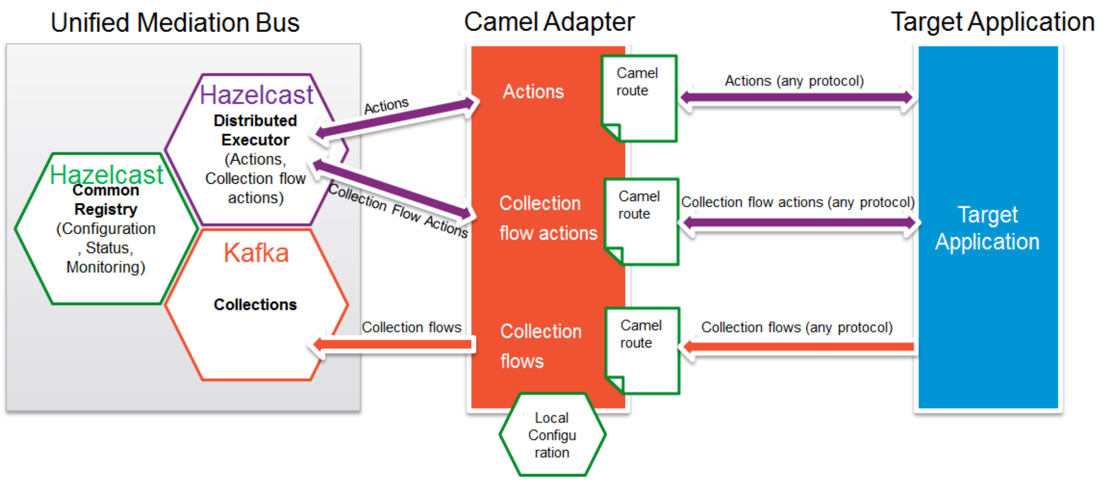


Figure 45 - Camel adapter overview

In the above figure, the Camel Adapter is used to connect to a Target Application to the Unified Mediation Bus. The Camel routes defined in the `camel-context.xml` file are used to connect to the Target Application for processing actions, collection flow actions and collections of alarms/events.

As Camel is used for connection to the Target Application any protocol can be used to interact with the Target Application: web services (SOAP, REST), JMS, JDBC, …[6]

The following sections will explain in detail how the Camel Adapter works.

5.1.1 Configuration

The configuration files of the Camel Adapter are located in the `src/main/resources` and `src/test/resources` folders. Each of the configuration files is explained in detail below.

5.1.1.1 The adapter.properties file

The Adapter properties file: `adapter.properties` defines properties for the adapter including connection information for the UMB Kafka/ZooKeeper instance(s).

The following properties are defined by default in this file:

- producer.bootstrap.servers: a list of Kafka broker `<host>:<port>` information
- producer.acks: set to `1` by default, indicating that Kafka is in a mode where messages are acknowledged

Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `adapter.properties` file.

5.1.1.2 The hazelcast.xml file

The Adapter's Hazelcast configuration file: `hazelcast.xml` defines how to connect to the UMB Hazelcast instance(s).

[6] Please see http://camel.apache.org/components.html for a list of available Camel components

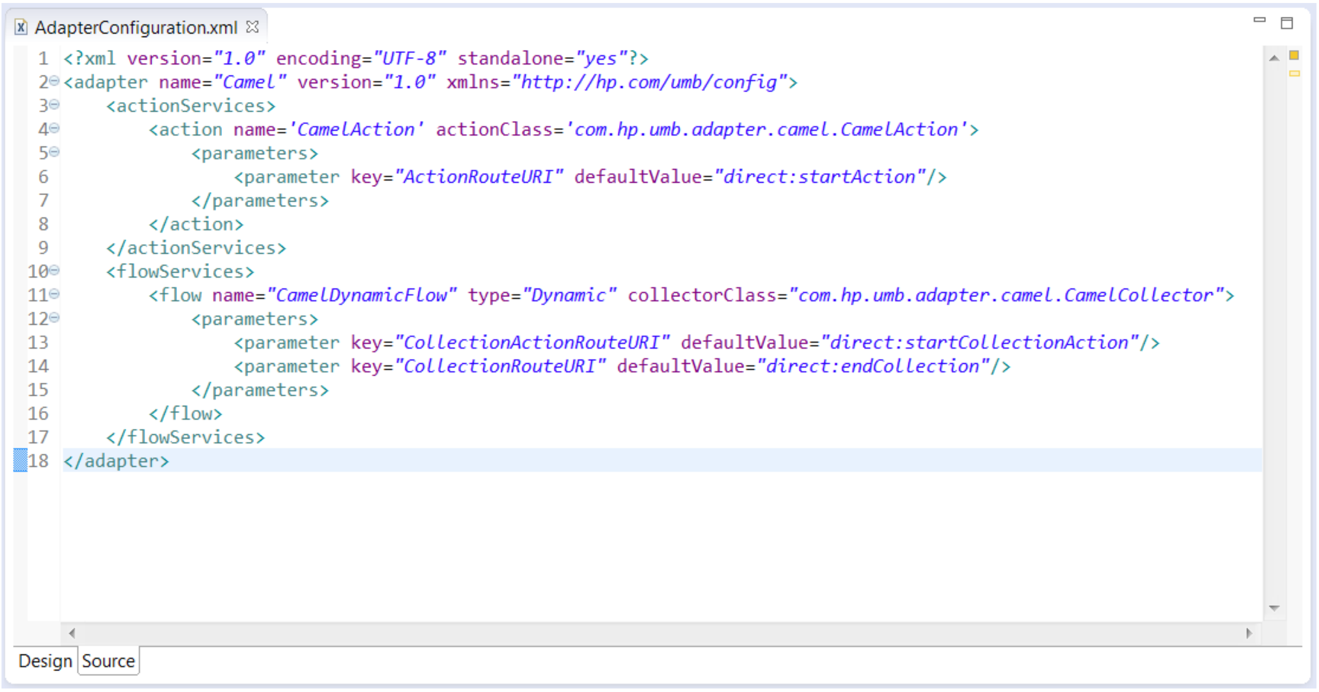
Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `hazelcast.xml` file.

5.1.1.3 The log4j.xml file

The Adapter's Log4j configuration file: `log4j.xml`

5.1.1.4 The AdapterConfiguration.xml file

The Adapter configuration file: `AdapterConfiguration.xml` defines the flows and actions provided by the adapter.



AdapterConfiguration.xml

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="Camel" version="1.0" xmlns="http://hp.com/umb/config">
    <actionServices>
        <action name='CamelAction' actionClass='com.hp.umb.adapter.camel.CamelAction'>
            <parameters>
                <parameter key="ActionRouteURI" defaultValue="direct:startAction"/>
            </parameters>
        </action>
    </actionServices>
    <flowServices>
        <flow name="CamelDynamicFlow" type="Dynamic" collectorClass="com.hp.umb.adapter.camel.CamelCollector">
            <parameters>
                <parameter key="CollectionActionRouteURI" defaultValue="direct:startCollectionAction"/>
                <parameter key="CollectionRouteURI" defaultValue="direct:endCollection"/>
            </parameters>
        </flow>
    </flowServices>
</adapter>
```

Figure 46 - The Camel Adapter's AdapterConfiguration.xml file

By default, one action and one flow are defined.

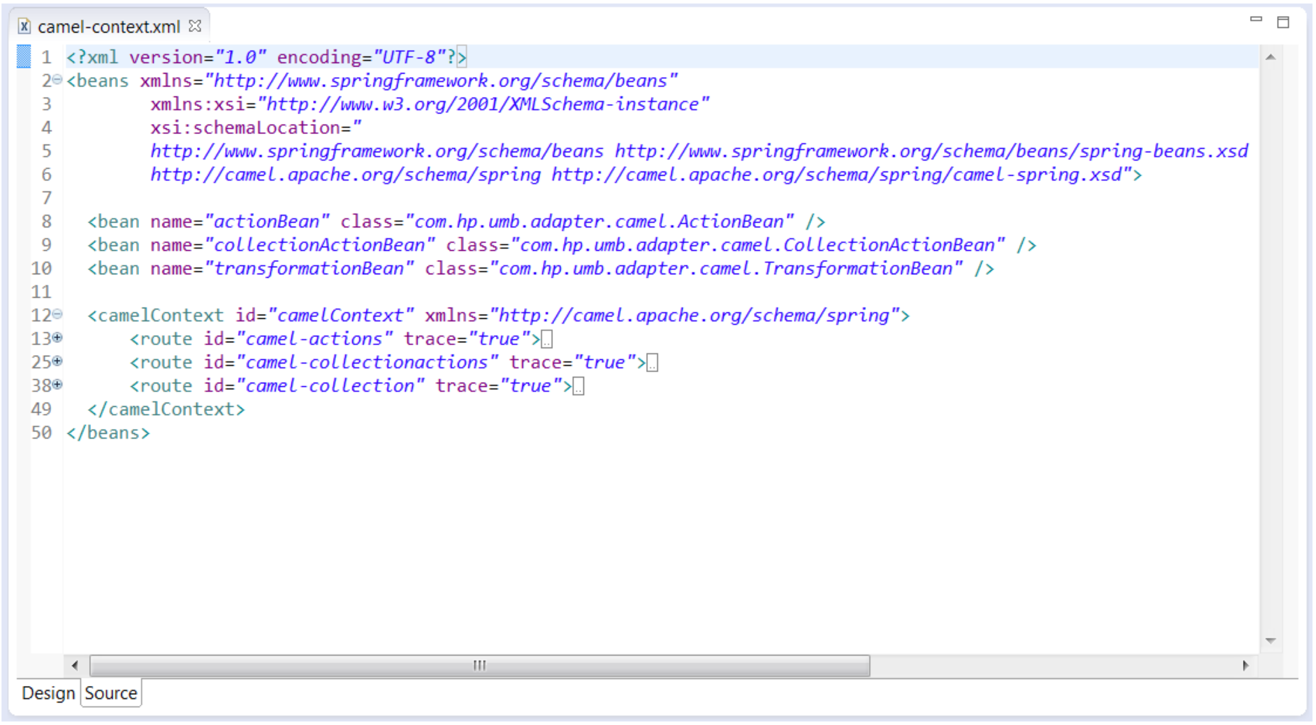
The action named "`CamelAction`" defines its implementing class as well as the Camel route start endpoint URI associated with the action. This URI is a reference to the URI of the start endpoint of the Camel route named "`camel-actions`" in the `camel-context.xml` file.

The flow named "`CamelDynamicFlow`" defines its implementing class as well as both the Camel route start endpoint URI for collection flow actions and the Camel route end endpoint URI for the collection associated with the flow. The "`CollectionActionRouteURI`" URI is a reference to the URI of the start endpoint of the Camel route named "`camel-collectionactions`" in the `camel-context.xml` file. The "`CollectionRouteURI`" URI is a reference to the URI of the end endpoint of the Camel route named "`camel-collection`" in the `camel-context.xml` file.

Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `AdapterConfiguration.xml` file.

5.1.1.5 The camel-context.xml file

A Camel Spring file: `camel-context.xml` defines routes to be used for processing actions, collection flow actions and collections.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">

  <bean name="actionBean" class="com.hp.umb.adapter.camel.ActionBean" />
  <bean name="collectionActionBean" class="com.hp.umb.adapter.camel.CollectionActionBean" />
  <bean name="transformationBean" class="com.hp.umb.adapter.camel.TransformationBean" />

  <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
      <route id="camel-actions" trace="true">
      <route id="camel-collectionactions" trace="true">
      <route id="camel-collection" trace="true">
  </camelContext>
</beans>
```

Figure 47 - The Camel Adapter's camel-context.xml file

Note: In the above screen capture of the `camel-context.xml` file, the Camel routes are collapsed, so the detail of these routes is not shown. These routes will be presented and explained in detail in the remainder of this section.

The `camel-context.xml` file is a Spring XML file that defines a Camel Context which in turn defines Camel routes[7].

The Camel Adapter defines 3 routes:

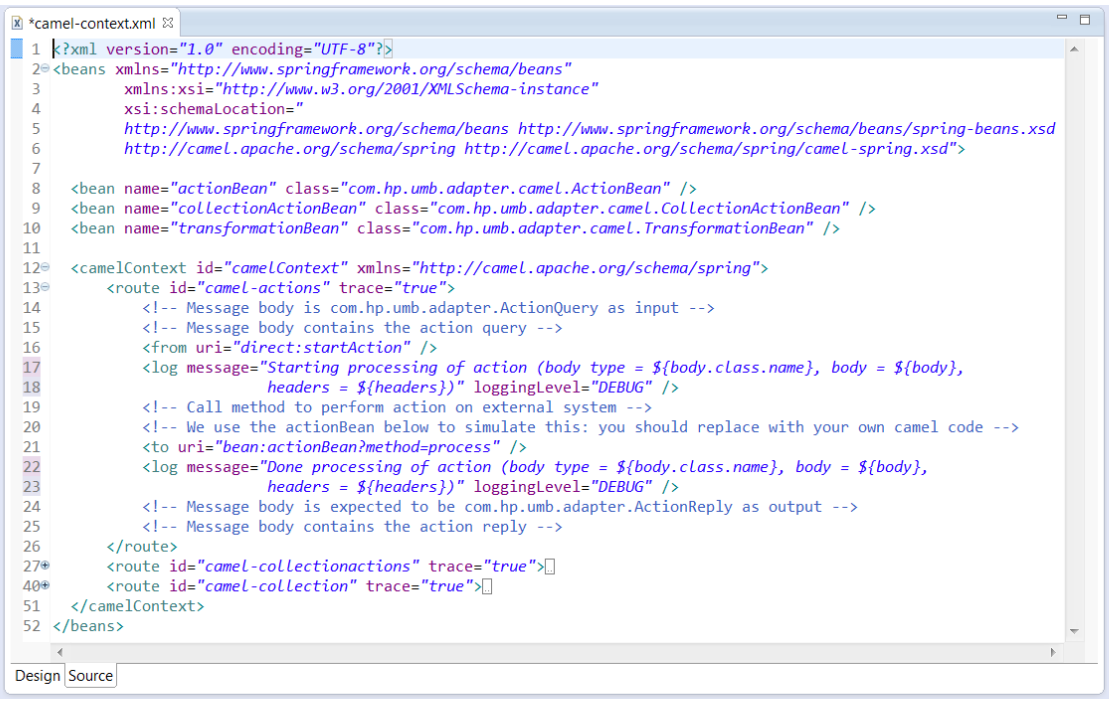
- The "`camel-actions`" route: this route processes action requests for actions named "`CamelAction`"
- The "`camel-collectionactions`" route: this route processes collection flow action requests, i.e. `CreateFlow`/`DeleteFlow`/`ResynchFlow`/`StatusFlow` for the flow named "`CamelDynamicFlow`"
- The "`camel-collection`" route: this route processes collection of alarms/events for the flow named "`CamelDynamicFlow`"

These routes (or more accurately the start or end endpoint URIs of these routes) are referenced in the `AdapterConfiguration.xml` file as shown in the previous section: 5.1.1.4 "The AdapterConfiguration.xml file".

5.1.1.5.1 The "camel-actions" route

As mentioned above, this route processes action requests for actions named "`CamelAction`" as per the configuration of the "`CamelAction`" action in the `AdapterConfiguration.xml` file.

[7] Please see: http://camel.apache.org/spring.html for more information on Camel Spring



```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">

  <bean name="actionBean" class="com.hp.umb.adapter.camel.ActionBean" />
  <bean name="collectionActionBean" class="com.hp.umb.adapter.camel.CollectionActionBean" />
  <bean name="transformationBean" class="com.hp.umb.adapter.camel.TransformationBean" />

  <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
      <route id="camel-actions" trace="true">
          <!-- Message body is com.hp.umb.adapter.ActionQuery as input -->
          <!-- Message body contains the action query -->
          <from uri="direct:startAction" />
          <log message="Starting processing of action (body type = ${body.class.name}, body = ${body},
                        headers = ${headers})" loggingLevel="DEBUG" />
          <!-- Call method to perform action on external system -->
          <!-- We use the actionBean below to simulate this: you should replace with your own camel code -->
          <to uri="bean:actionBean?method=process" />
          <log message="Done processing of action (body type = ${body.class.name}, body = ${body},
                        headers = ${headers})" loggingLevel="DEBUG" />
          <!-- Message body is expected to be com.hp.umb.adapter.ActionReply as output -->
          <!-- Message body contains the action reply -->
      </route>
      <route id="camel-collectionactions" trace="true">
      <route id="camel-collection" trace="true">
  </camelContext>
</beans>
```

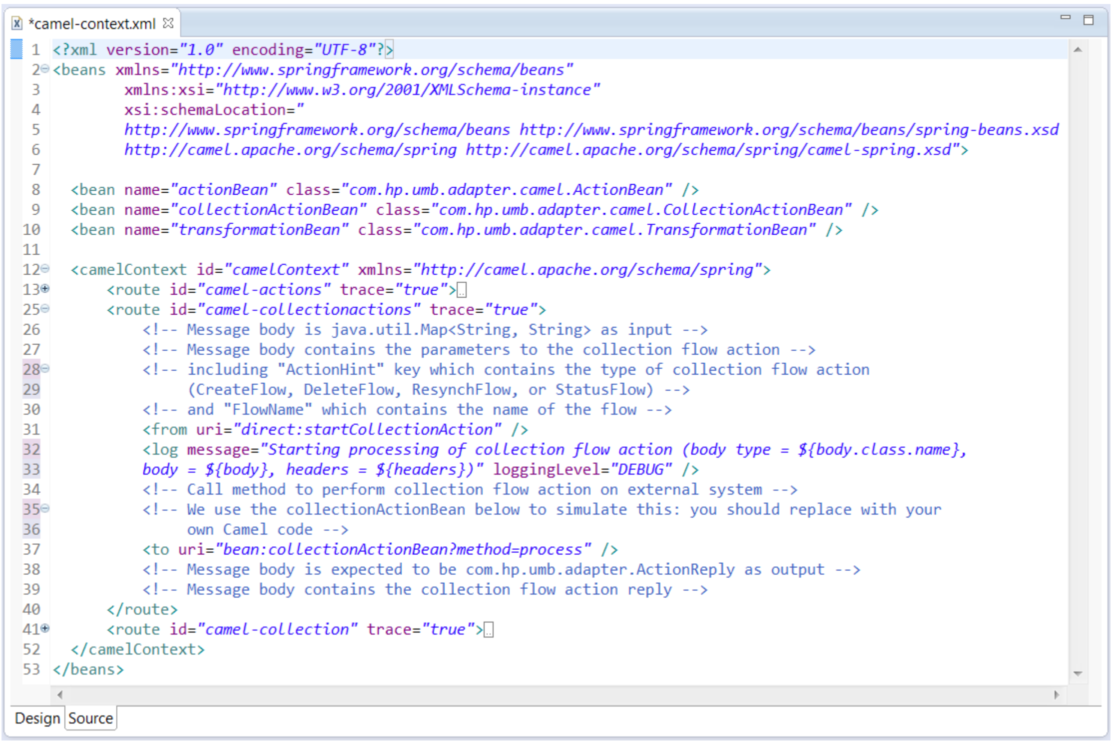
Figure 48 - “camel-actions” route in the camel-context.xml file

This route works by requesting an action to be performed on the Target Application and processing the action response.

This route is explained in detail in the 5.1.2.1 “Actions” chapter.

5.1.1.5.2 The “camel-collectionactions” route

This route processes collection flow action requests, i.e. `CreateFlow`/`DeleteFlow`/`ResynchFlow`/`StatusFlow` for the flow named “`CamelDynamicFlow`” as per the configuration of the “`CamelDynamicFlow`” flow in the `AdapterConfiguration.xml` file.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">

  <bean name="actionBean" class="com.hp.umb.adapter.camel.ActionBean" />
  <bean name="collectionActionBean" class="com.hp.umb.adapter.camel.CollectionActionBean" />
  <bean name="transformationBean" class="com.hp.umb.adapter.camel.TransformationBean" />

  <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
      <route id="camel-actions" trace="true">
      <route id="camel-collectionactions" trace="true">
          <!-- Message body is java.util.Map<String, String> as input -->
          <!-- Message body contains the parameters to the collection flow action -->
          <!-- including "ActionHint" key which contains the type of collection flow action
               (CreateFlow, DeleteFlow, ResynchFlow, or StatusFlow) -->
          <!-- and "FlowName" which contains the name of the flow -->
          <from uri="direct:startCollectionAction" />
          <log message="Starting processing of collection flow action (body type = ${body.class.name},
          body = ${body}, headers = ${headers})" loggingLevel="DEBUG" />
          <!-- Call method to perform collection flow action on external system -->
          <!-- We use the collectionActionBean below to simulate this: you should replace with your
               own Camel code -->
          <to uri="bean:collectionActionBean?method=process" />
          <!-- Message body is expected to be com.hp.umb.adapter.ActionReply as output -->
          <!-- Message body contains the collection flow action reply -->
      </route>
      <route id="camel-collection" trace="true">
  </camelContext>
</beans>
```

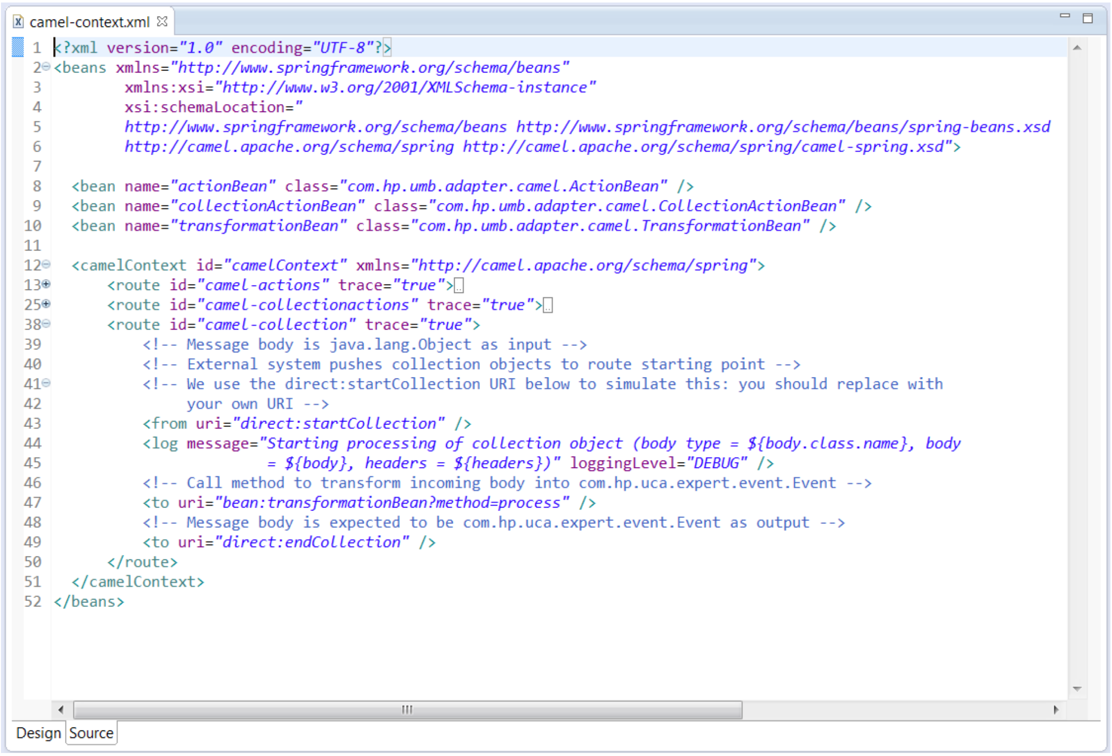
Figure 49 - “camel-collectionactions” route in the camel-context.xml file

This route works by requesting a collection flow action to be performed on the Target Application and processing the action response.

This route is explained in detail in the 5.1.2.2 “Collections” chapter.

5.1.1.5.3 The “camel-collection” route

This route processes collection of alarms/events for the flow named “`CamelDynamicFlow`” as per the configuration of the “`CamelDynamicFlow`” flow in the `AdapterConfiguration.xml` file.



camel-context.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">

  <bean name="actionBean" class="com.hp.umb.adapter.camel.ActionBean" />
  <bean name="collectionActionBean" class="com.hp.umb.adapter.camel.CollectionActionBean" />
  <bean name="transformationBean" class="com.hp.umb.adapter.camel.TransformationBean" />

  <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
      <route id="camel-actions" trace="true">
      <route id="camel-collectionactions" trace="true">
      <route id="camel-collection" trace="true">
          <!-- Message body is java.lang.Object as input -->
          <!-- External system pushes collection objects to route starting point -->
          <!-- We use the direct:startCollection URI below to simulate this: you should replace with
               your own URI -->
          <from uri="direct:startCollection" />
          <log message="Starting processing of collection object (body type = ${body.class.name}, body
                        = ${body}, headers = ${headers})" loggingLevel="DEBUG" />
          <!-- Call method to transform incoming body into com.hp.uca.expert.event.Event -->
          <to uri="bean:transformationBean?method=process" />
          <!-- Message body is expected to be com.hp.uca.expert.event.Event as output -->
          <to uri="direct:endCollection" />
      </route>
  </camelContext>
</beans>
```

Figure 50 - “camel-collection” route in the camel-context.xml file

This route works by collecting alarms/events from a Target Application and forwarding them to the Collection service of the UMB framework, implemented by Kafka/ZooKeeper.

This route is explained in detail in the 5.1.2.2 “Collections” chapter.

5.1.2 How does it work?

5.1.2.1 Actions

When a “`CamelAction`” is requested to be executed by the Camel Adapter, the request will be handed over to the “`CamelAction`” implementing class (the `com.hp.umb.adapter.camel.CamelAction` Java class) by the UMB framework. The `CamelAction` class will push the request to the “`camel-actions`” route defined in the `camel-context.xml` file. Inside the “`camel-actions`” route, the request will be processed by being sent to the Target Application. We use the `com.hp.umb.adapter.camel.ActionBean` Java class inside the “`camel-actions`” route to simulate the request being sent to the Target Application. The response to the request is picked up by the `com.hp.umb.adapter.camel.CamelAction` Java class at the end of the "`camel-actions`" route. The response is then forwarded to the original requester by the UMB framework.

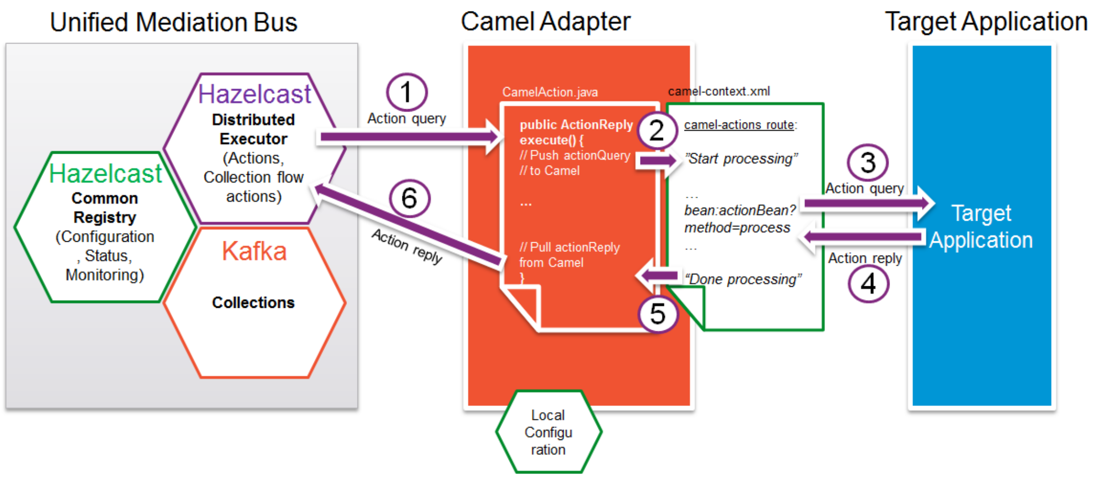


Figure 51 - Processing Actions in the Camel Adapter

Executing an action on the Camel Adapter entails the following steps:

1. An action request is forwarded by the Distributed Executor service of the UMB framework to the Camel Adapter. This action request comes from another Adapter connected to the UMB. If this action is named "`CamelAction`" (let's assume this is the case), then the action request is to be processed by the `com.hp.umb.adapter.camel.CamelAction` class, as per the Camel Adapter's `AdapterConfiguration.xml` configuration file. An action request in the UMB framework takes the form of a `com.hp.umb.adapter.ActionQuery` object.
2. The action request is processed by the `public ActionReply execute()` method in the `CamelAction` class. The `ActionQuery` object that represents the action request is pushed to the start endpoint of the "`camel-actions`" route.
3. The `ActionQuery` object follows the "`camel-actions`" route step by step. Along this route, the action request is sent to the "`actionBean`" for processing[8]. This step simulates the action request being sent to a Target Application for processing. Should you wish to actually connect to a Target Application, you should consider replacing this step by your own Camel code.
4. The action request is processed by the "`actionBean`" which returns an action response in the form of a `com.hp.umb.adapter.ActionReply` object which is pushed back along the Camel route.
5. The Camel route ends and the action response is returned to the `public ActionReply execute()` method in the `CamelAction` class.
6. The `public ActionReply execute()` method returns the action response to the Distributed Executor service of the UMB framework, which in turn sends it to whichever Adapter requested the action to be processed initially.

The "`camel-actions`" route in the Camel Adapter is an example route for processing action requests using Camel. You can modify this route to do your own processing using the full extent of the Camel Spring DSL. The only constraint is that the messages processed by the "`camel-actions`" route have to be of type

[8] The "`actionBean`" is implemented by the `com.hp.umb.adapter.camel.ActionBean` class as the bean declaration for the "`actionBean`" indicates, at the beginning of the `camel-context.xml` file

`com.hp.umb.adapter.ActionQuery` as input of the route and
`com.hp.umb.adapter.ActionReply` as output of the route.

5.1.2.2 Collections

When a collection flow action (`CreateFlow`/`DeleteFlow`/`ResynchFlow`/`StatusFlow`) is requested to be executed by the Camel Adapter for the "`CamelDynamicFlow`" flow, the request will be handed over to the "`CamelDynamicFlow`" implementing class (the `com.hp.umb.adapter.camel.CamelCollector` Java class) by the UMB framework.

The `CamelCollector` class will push the request to the "`camel-collectionactions`" route defined in the `camel-context.xml` file. Inside the "`camel-collectionactions`" route, the request will be processed by being sent to the Target Application. We use the `com.hp.umb.adapter.camel.CollectionActionBean` Java class inside the "`camel-collectionactions`" route to simulate the request being sent to the Target Application. The response to the request is picked up by the `com.hp.umb.adapter.camel.CamelCollector` Java class at the end of the "`camel-collectionactions`" route. The response is then forwarded to the original requester by the UMB framework.

Once a collection has been created, the Target Application will push collection alarms/events to the start of the "`camel-collection`" route. These alarms/events will be transformed by the `com.hp.umb.adapter.camel.TransformationBean` Java class so that they can be mapped into event (or alarms) compatible with the UMB framework. The alarms/events will then be picked by the `com.hp.umb.adapter.camel.CamelCollector` Java class at the end of the "`camel-collection`" route. They will then be forwarded to the proper Topic on the Kafka instance(s) part of the UMB framework.

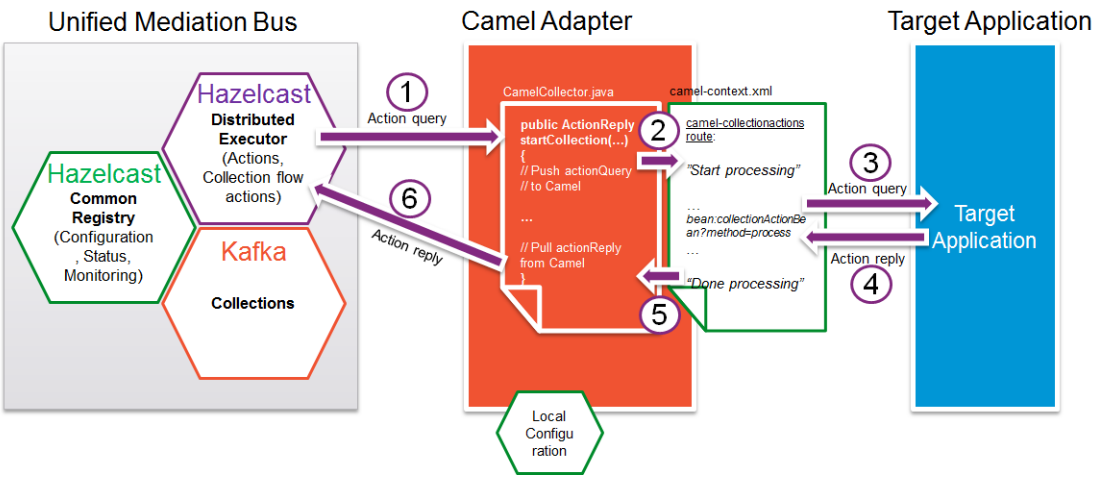


Figure 52 - Processing Collection Flow Actions in the Camel Adapter

Processing collection flow actions is very similar to processing actions in the Camel Adapter as described in the 5.1.2.1 "Actions" chapter.

The only differences are that:

- collection flow actions are processed inside the `com.hp.umb.adapter.camel.CamelCollector` Java class (instead of the `com.hp.umb.adapter.camel.CamelAction` Java class for actions) by either of the following methods (instead of the `public ActionReply execute()` method for actions):
  - public ActionReply startCollection()

  - public ActionReply stopCollection()
  - public ActionReply resynchCollection()
  - public ActionReply getCollectionStatus()
- collection flow actions are processed by the "`camel-collectionactions`" route (instead of the "`camel-actions`" route for actions)
- to simulation the collection flow actions being processed by a target application the `collectionActionBean` bean is used (instead of the `actionBean` bean for actions)

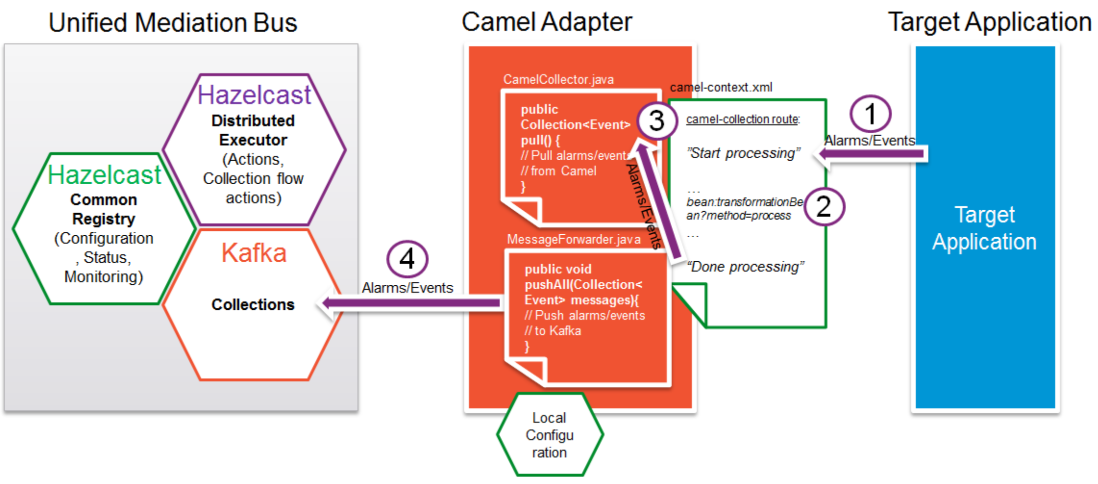


Figure 53 - Processing Collections in the Camel Adapter

Collecting alarms/events in the Camel Adapter entails the following steps:

1. Thanks to a previous `CreateFlow` action, a collection has been created on the Target Application which pushes alarms/events to the "`camel-collection`" route start endpoint: `direct:startCollection`. As the Camel Adapter is an example Adapter we use a `direct:` endpoint (for simplicity's sake) as the start endpoint of the route. In a real use case, we could imagine that the Target Application pushes alarms/events to a JMS queue/topic and we would use this JMS queue/topic as the start endpoint of the route.
2. Alarms/Events are processed along the "`camel-collection`" route by the `transformationBean` bean. This bean provides a means to transform Alarm/Event objects initially in the Target Application format into Alarms/Events in UCA EBC format (objects that implement the `com.hp.uca.expert.event.Event` Java interface)
3. Alarms/Event are picked up at the end of the "`camel-collection`" route by the `public Collection<Event> pull()` method of the `com.hp.umb.adapter.camel.CamelCollector` Java class.
4. These alarms/events are then pushed to the Collection service component of the UMB framework implemented by Kafka/Zookeeper on the topic associated with the collection flow. The alarm/event collection is thus made available for consumption by the Adapter that requested the collection flow to be created in the first place (since this is a dynamic flow as per the `AdapterConfiguration.xml` file).

5.1.3 JUnit tests

A JUnit test is present in the `src/test/java` folder. The name of the JUnit test class is `com.hp.umb.adapter.camel.CamelAdapterTest`. This class contains 2 test methods:

- A method that tests action executions named: `testExecuteAction()`
- A method that tests collection flows named: `testFlowAction()`

The `testExecuteAction()` test works by requesting an action to be executed on the Camel Adapter and verifying that the action response is correct.

The `testFlowAction()` test works by creating a collection flow on the Camel Adapter, resynchronizing it, retrieving its status and then deleting it.

5.2 File Adapter

The File Adapter is a sample adapter that demonstrates how a UMB Adapter can easily provide flow collection services based on files. This adapter acts as a Flow service provider.

As a Flow provider, the adapter will:

- respond to collection flow actions: `CreateFlow`, `DeleteFlow`, `ResynchFlow`, `StatusFlow`
- as a consequence of these collection flow actions, the adapter will create/delete/resynchronize or get the status of collections of alarms or events

There are 2 distinct parts in the File Adapter:

- One that can produce flows of alarms based on alarms stored in an XML file
- One that can produce flows of events (temperatures in our case) based on data stored in a comma-separated values (CSV) file

The File Adapter is composed of:

- Configuration files:
  - The Adapter properties file: `adapter.properties` that defines properties for the adapter including connection information for Kafka/ZooKeeper
  - The Adapter's Hazelcast configuration file: `hazelcast.xml` that defines how to connect to the UMB Hazelcast Central Repository
  - The Adapter's Log4j configuration file: `log4j.xml`
  - The Adapter configuration file: `AdapterConfiguration.xml` that defines the flows and actions (in our case just flows, no actions) provided by the adapter
- Data files:
  - An XML alarms file: `alarms.xml` that contains alarms in XML format to be used to create alarm flows
  - An comma-separated values (CSV) file: `temperatures.csv` that contains temperature data in CSV format to be used to create temperature flows
- Java files that define the Adapter's behavior

The following figure explains the overall architecture of the File Adapter.

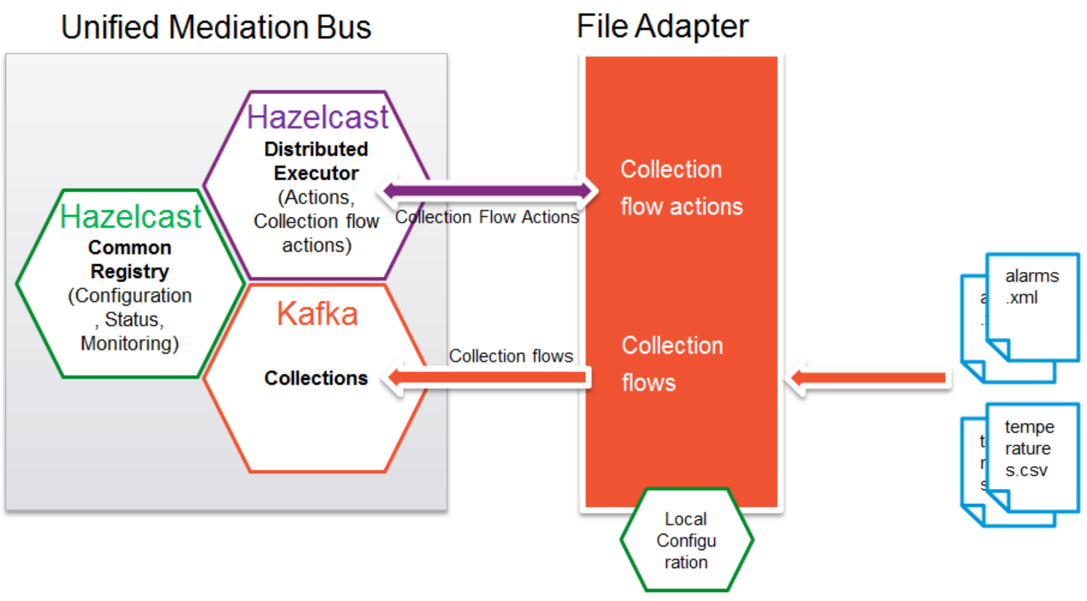


Figure 54 - File adapter overview

In the above figure, the File Adapter is used to provide alarm and event (temperatures) collection flows to the Unified Mediation Bus based on data files in XML format for alarms and CSV format for events (temperatures). The flows are defined in the `AdapterConfiguration.xml` file. A data file is associated with each flow. Both static and dynamic flows are supported.

The following sections will explain in detail how the File Adapter works.

Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `AdapterConfiguration.xml` file.

5.2.1 Configuration

The configuration files of the File Adapter are located in the `src/main/resources` and `src/test/resources` folders (data files are located in the `src/test/resources/data` folder). Each of the configuration files is explained in detail below.

5.2.1.1 The adapter.properties file

The Adapter properties file: `adapter.properties` defines properties for the adapter including connection information for the UMB Kafka/ZooKeeper instance(s).

The following properties are defined by default in this file:

- producer.bootstrap.servers: a list of Kafka brokers in the form of `<host>:<port>`
- producer.acks: set to `1` by default, indicating that Kafka is in a mode where messages are acknowledged
- zookeeper.connect: a list of Zookeeper instances in the form of `<host>:<port>`

Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `adapter.properties` file.

5.2.1.2 The hazelcast.xml file

The Adapter's Hazelcast configuration file: `hazelcast.xml` defines how to connect to the UMB Hazelcast instance(s).

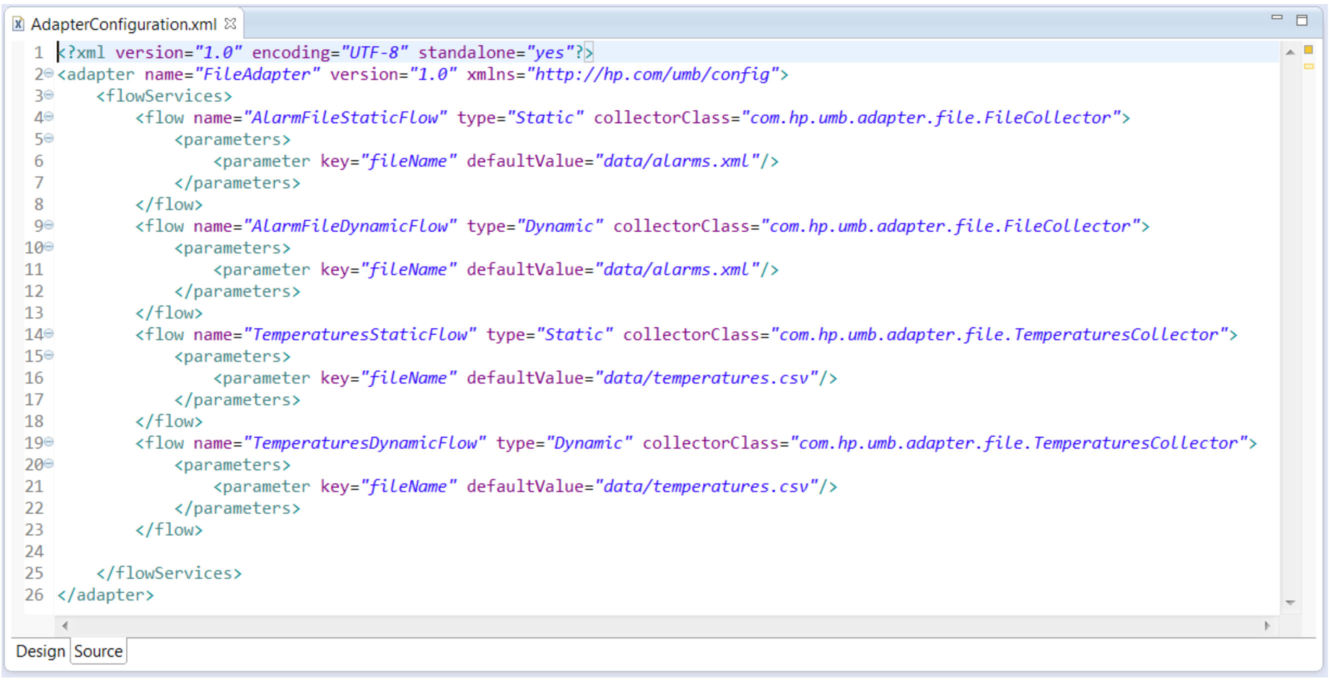
Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `hazelcast.xml` file.

5.2.1.3 The log4j.xml file

The Adapter's Log4j configuration file: `log4j.xml`

5.2.1.4 The AdapterConfiguration.xml file

The Adapter configuration file: `AdapterConfiguration.xml` defines the flows and actions provided by the adapter.



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="FileAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <flowServices>
        <flow name="AlarmFileStaticFlow" type="Static" collectorClass="com.hp.umb.adapter.file.FileCollector">
            <parameters>
                <parameter key="fileName" defaultValue="data/alarms.xml"/>
            </parameters>
        </flow>
        <flow name="AlarmFileDynamicFlow" type="Dynamic" collectorClass="com.hp.umb.adapter.file.FileCollector">
            <parameters>
                <parameter key="fileName" defaultValue="data/alarms.xml"/>
            </parameters>
        </flow>
        <flow name="TemperaturesStaticFlow" type="Static" collectorClass="com.hp.umb.adapter.file.TemperaturesCollector">
            <parameters>
                <parameter key="fileName" defaultValue="data/temperatures.csv"/>
            </parameters>
        </flow>
        <flow name="TemperaturesDynamicFlow" type="Dynamic" collectorClass="com.hp.umb.adapter.file.TemperaturesCollector">
            <parameters>
                <parameter key="fileName" defaultValue="data/temperatures.csv"/>
            </parameters>
        </flow>

    </flowServices>
</adapter>
```

Figure 55 - The File Adapter's AdapterConfiguration.xml file

By default, 4 flows are defined:

- 2 alarm flows: one static and one dynamic[9]
- 2 temperatures flows: one static and one dynamic

The flow named "`AlarmFileStaticFlow`" defines its implementing class (`com.hp.umb.adapter.file.FileCollector`) as well as the data file to use (`data/alarms.xml`). Its type is declared to be static.

[9] Static flow are automatically started when the Adapter is started while dynamic flows are not. For dynamic flows a `CreateFlow` collection flow action needs to be sent to the adapter for the flow to be created and started. This is done automatically by the flow consumer when the `startCollection()` method is called.

The flow named "`AlarmFileDynamicFlow`" is identical to the "`AlarmFileStaticFlow`" except that it is declared to be dynamic.

The flow named "`TemperaturesStaticFlow`" defines its implementing class (`com.hp.umb.adapter.file.TemperaturesCollector`) as well as the data file to use (`data/temperatures.csv`). Its type is declared to be static.

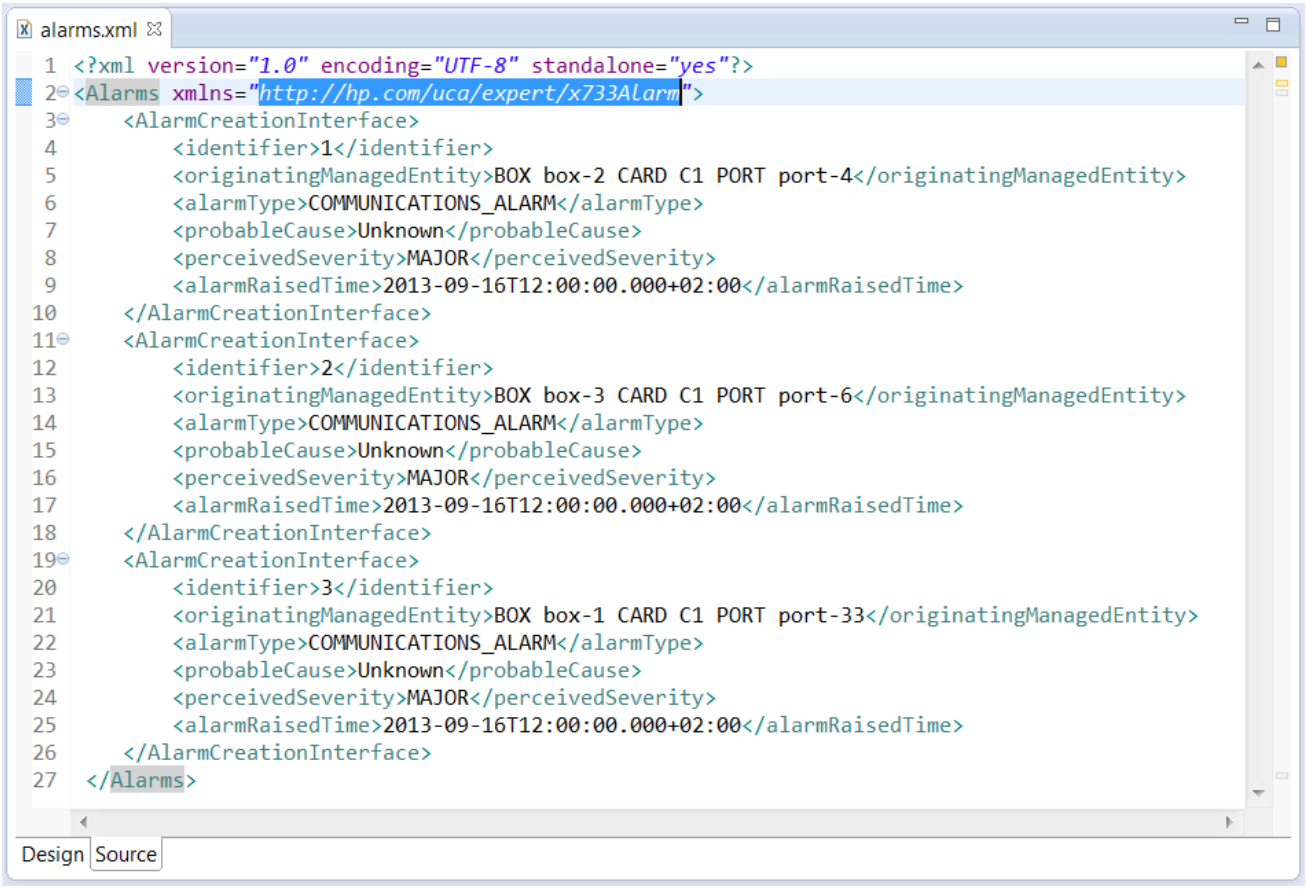
The flow named "`TemperaturesDynamicFlow`" is identical to the "`TemperaturesStaticFlow`" except that it is declared to be dynamic.

Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `AdapterConfiguration.xml` file.

5.2.1.4.1 The "alarms.xml" data file

This file contains alarms in XML format to be used for both the "`AlarmFileStaticFlow`" and "`AlarmFileDynamicFlow`" flows.

This file uses the same format as alarm files in the UCA EBC application (the XML namespace used is: `http://hp.com/uca/expert/x733Alarm`).

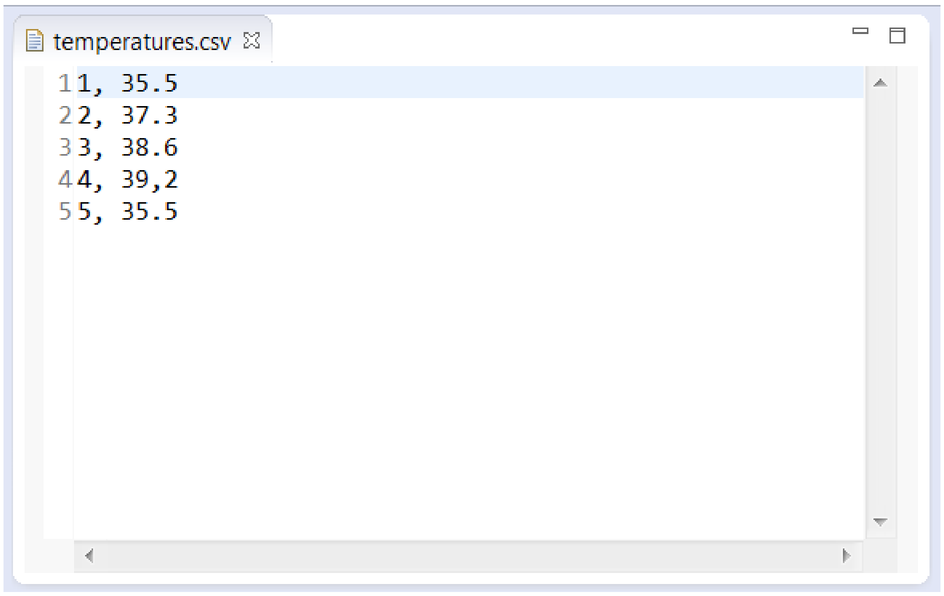


```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Alarms xmlns="http://hp.com/uca/expert/x733Alarm">
    <AlarmCreationInterface>
        <identifier>1</identifier>
        <originatingManagedEntity>BOX box-2 CARD C1 PORT port-4</originatingManagedEntity>
        <alarmType>COMMUNICATIONS_ALARM</alarmType>
        <probableCause>Unknown</probableCause>
        <perceivedSeverity>MAJOR</perceivedSeverity>
        <alarmRaisedTime>2013-09-16T12:00:00.000+02:00</alarmRaisedTime>
    </AlarmCreationInterface>
    <AlarmCreationInterface>
        <identifier>2</identifier>
        <originatingManagedEntity>BOX box-3 CARD C1 PORT port-6</originatingManagedEntity>
        <alarmType>COMMUNICATIONS_ALARM</alarmType>
        <probableCause>Unknown</probableCause>
        <perceivedSeverity>MAJOR</perceivedSeverity>
        <alarmRaisedTime>2013-09-16T12:00:00.000+02:00</alarmRaisedTime>
    </AlarmCreationInterface>
    <AlarmCreationInterface>
        <identifier>3</identifier>
        <originatingManagedEntity>BOX box-1 CARD C1 PORT port-33</originatingManagedEntity>
        <alarmType>COMMUNICATIONS_ALARM</alarmType>
        <probableCause>Unknown</probableCause>
        <perceivedSeverity>MAJOR</perceivedSeverity>
        <alarmRaisedTime>2013-09-16T12:00:00.000+02:00</alarmRaisedTime>
    </AlarmCreationInterface>
 </Alarms>
```

Figure 56 - File Adapter's "alarms.xml" data file

5.2.1.4.2 The “temperatures.csv” data file

This file contains temperatures in CSV format to be used for both the “`TemperaturesStaticFlow`” and “`TemperaturesDynamicFlow`” flows.



```
1, 35.5
2, 37.3
3, 38.6
4, 39,2
5, 35.5
```

Figure 57 - File Adapter’s “temperatures.csv” data file

The first column is the temperature identifier, and the second column is the temperature value.

5.2.2 How does it work?

5.2.2.1 Collections

5.2.2.1.1 Alarms collections

Alarm collections are implemented with the `com.hp.umb.adapter.file.FileCollector` Java class, as apparent by the “`AlarmFileStaticFlow`” and “`AlarmFileDynamicFlow`” flow definitions in the `AdapterConfiguration.xml` file.

The `FileCollector` Java class will respond to collection flow action requests (`CreateFlow`/`DeleteFlow`/`ResynchFlow`/`StatusFlow`) from the UMB framework and also handle the actual collection of alarms from the alarms file (specified in the `AdapterConfiguration.xml` file) to the collections service of the UMB framework (implemented by Kafka) and from there to potential consumers.

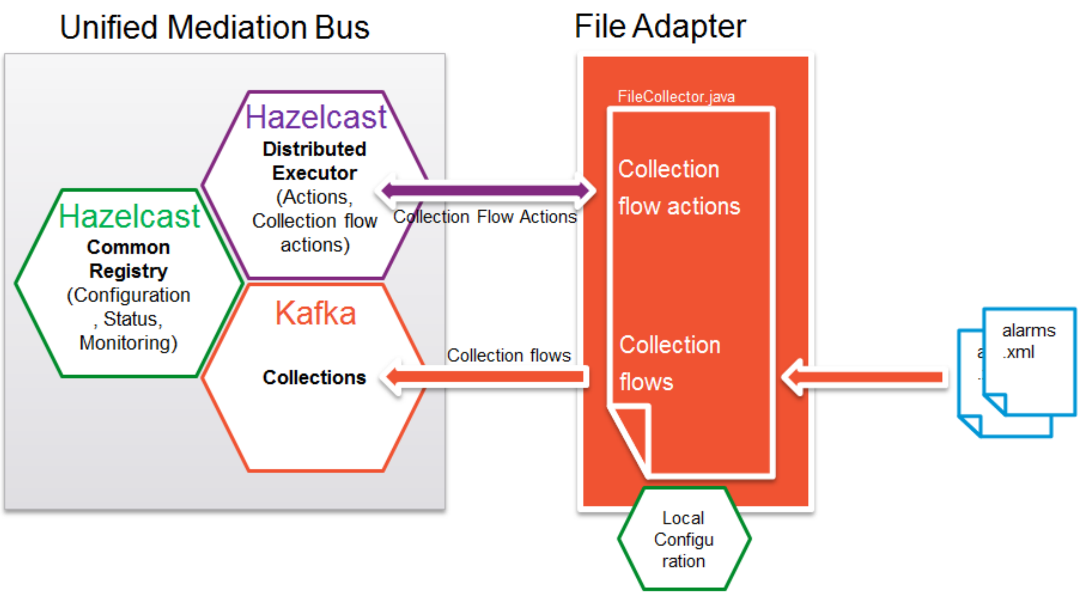


Figure 58 - File Adapter's alarms collections

5.2.2.1.2 Temperatures collections

Temperature collections are implemented with the `com.hp.umb.adapter.file.TemperaturesCollector` Java class, as apparent by the "`TemperaturesStaticFlow`" and "`TemperaturesDynamicFlow`" flow definitions in the `AdapterConfiguration.xml` file.

The `TemperaturesCollector` Java class will respond to collection flow action requests (`CreateFlow`/`DeleteFlow`/`ResynchFlow`/`StatusFlow`) from the UMB framework and also handle the actual collection of temperatures from the temperatures CSV file (specified in the `AdapterConfiguration.xml` file) to the collections service of the UMB framework (implemented by Kafka) and from there to potential consumers.

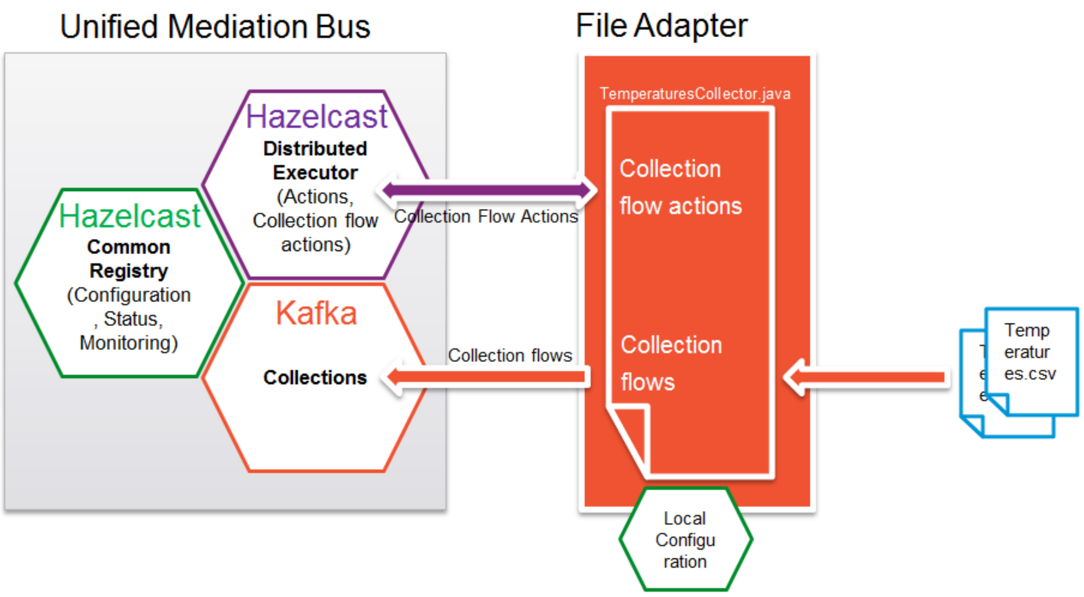


Figure 59 - File Adapter's temperatures collections

5.2.3 JUnit tests

A JUnit test is present in the `src/test/java` folder. The name of the JUnit test class is `com.hp.umb.adapter.file.FileAdapterTest`. This class contains 1 test method:

- A method that tests collection flows named: `testFlowAction()`

The `testFlowAction()` test works by creating a collection flow on the File Adapter, resynchronizing it, retrieving its status and then deleting it.

5.3 Log Adapter

The Log Adapter is a sample adapter that demonstrates how a UMB Adapter can be easily set up as a flow consumer in order to log alarms or events from existing UMB collection flows.

As a Flow consumer, the adapter will:

- send collection flow actions: CreateFlow, DeleteFlow, ResynchFlow to target UMB Adapters acting as flow producers
- consume (and log) alarms or events from these collection flows

The Log Adapter defines 2 flow consumers by default:

- One that can consume flows of alarms[10]: `com.hp.umb.adapter.log.LogAlarmConsumer`
- One that can consume flows of events[11]: `com.hp.umb.adapter.log.LogEventConsumer`

The Log Adapter is composed of:

- Configuration files:
    - The Adapter properties file: `adapter.properties` that defines properties for the adapter including connection information for Kafka/ZooKeeper
    - The Adapter's Hazelcast configuration file: `hazelcast.xml` that defines how to connect to the UMB Hazelcast Central Repository
    - The Adapter's Log4j configuration file: `log4j.xml`
    - The Adapter configuration file: `AdapterConfiguration.xml` that defines the flows and actions (in our case no flows or actions are defined) provided by the adapter
- Java files that define the Adapter's behavior

The following figure explains the overall architecture of the Log Adapter.

[10] Alarms are objects that implement the `com.hp.uca.expert.alarm.AlarmCommon` Java interface

[11] Events are objects that implement the `com.hp.uca.expert.event.Event` Java interface

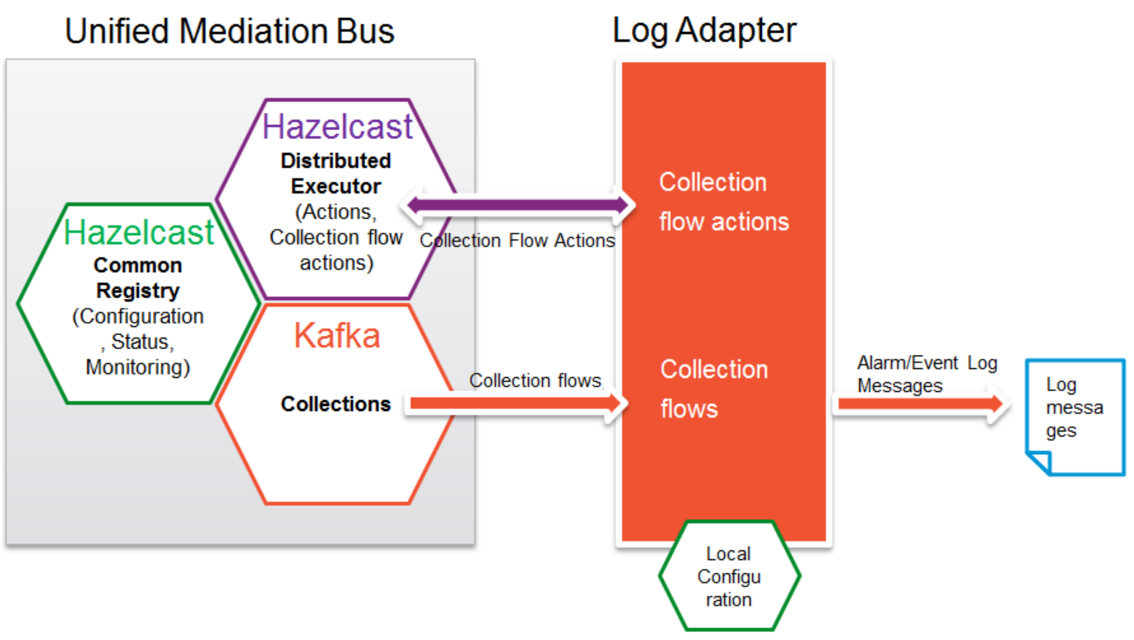


Figure 60 - Log adapter overview

In the above figure, the Log Adapter is used to consume alarm and event collection flows from the Unified Mediation Bus and log these alarms and events as log messages. Both static and dynamic flows are supported. The Log Adapter can also send collection flow actions to UMB in order to create/delete/resynchronize collection flows.

The following sections will explain in detail how the Log Adapter works.

Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `AdapterConfiguration.xml` file.

5.3.1 Configuration

The configuration files of the Log Adapter are located in the `src/main/resources` and `src/test/resources` folders. Each of the configuration files is explained in detail below.

5.3.1.1 The adapter.properties file

The Adapter properties file: `adapter.properties` defines properties for the adapter including connection information for the UMB Kafka/ZooKeeper instance(s).

The following properties are defined by default in this file:

```
Producer properties
producer.bootstrap.servers=localhost:9091
producer.acks=1
producer.batch.size=1000000
producer.linger.ms=500

Consumer specific properties
consumer.bootstrap.servers=localhost:9091

Adapter general properties
zookeeper.connect=localhost:2181, localhost:2182
hazelcast.backpressure.workaround.delay.ms=50
```

Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `adapter.properties` file.

5.3.1.2 The hazelcast.xml file

The Adapter's Hazelcast configuration file: `hazelcast.xml` defines how to connect to the UMB Hazelcast instance(s).

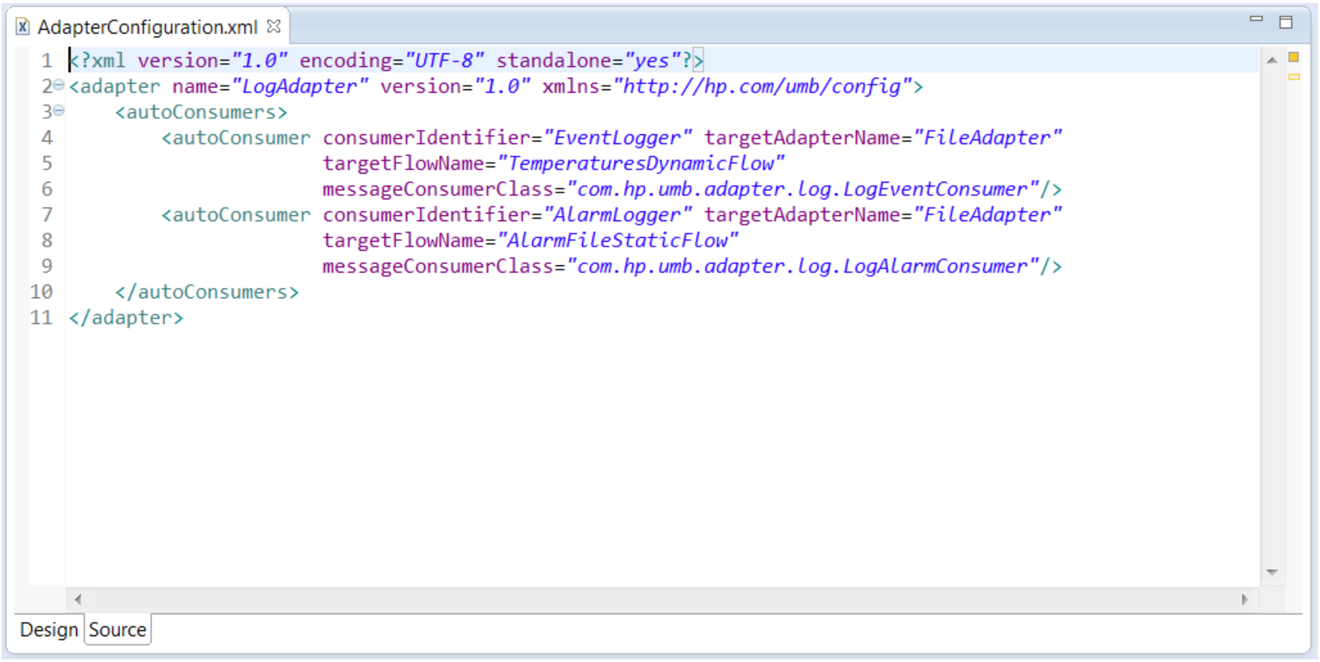
Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `hazelcast.xml` file.

5.3.1.3 The log4j.xml file

The Adapter's Log4j configuration file: `log4j.xml`

5.3.1.4 The AdapterConfiguration.xml file

The Adapter configuration file: `AdapterConfiguration.xml` defines the flows and actions provided by the adapter.



```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<adapter name="LogAdapter" version="1.0" xmlns="http://hp.com/umb/config">
    <autoConsumers>
        <autoConsumer consumerIdentifier="EventLogger" targetAdapterName="FileAdapter"
                      targetFlowName="TemperaturesDynamicFlow"
                      messageConsumerClass="com.hp.umb.adapter.log.LogEventConsumer"/>
        <autoConsumer consumerIdentifier="AlarmLogger" targetAdapterName="FileAdapter"
                      targetFlowName="AlarmFileStaticFlow"
                      messageConsumerClass="com.hp.umb.adapter.log.LogAlarmConsumer"/>
    </autoConsumers>
</adapter>
```

Figure 61 - The Log Adapter's AdapterConfiguration.xml file

As the Log adapter is only a flow consumer, it does not define any producer flow services in the `AdapterConfiguration.xml` file. However it does declare "autoConsumers" for each consumer flow that needs to be automatically created and started. Declaring consumer flows in the `AdapterConfiguration.xml` file removes the need to add Java code in the Adapter's main Java class for the specific purpose of creating and starting consumer flows.

In the provided configuration example, two consumer flows that start automatically are defined:

- Alarm Logger
- Event Logger
- Each of them is consuming events from the File Adapter flows: the `TemperatureStaticFlow` flow for the `Even Logger` and the `AlarmFileStaticFlow` flow for the `Alarm Logger.`

The Log Adapter can be easily enhanced (even after the Adapter has been installed) by adding new consumer flows in the "autoConsumer" section of the configuration file and providing the associated consumer message handler class as a .jar file in the Log Adapter's `lib` directory.

Please refer to the [R1] Unified Mediation Bus installation and configuration Guide for details on how to configure the `AdapterConfiguration.xml` file.

5.3.2 How does it work?

5.3.2.1 Collections

5.3.2.1.1 Alarms/Events collections

The `com.hp.umb.adapter.log.LogAdapter` Java class is the class that implements the Log Adapter. It contains a `main(String[] args)` method that starts the Adapter based on configuration settings stored in the `AdapterConfiguration.xml` file.

The `AdapterConfiguration.xml` file defines 2 consumer flows that are set to automatically start when the Adapter starts:

- UcaStaticForwarderFlow
- UcaStaticEventForwarderFlow

Each of these consumer flows is associated with a consumer message handler[12] class that defines what to do with each message consumed from the collection flow. Messages take the form of alarms[13] in the case of the `UcaStaticForwarderFlow` flow and events[14] in the case of the `UcaStaticEventForwarderFlow` flow.

The following consumer message handler classes are defined in the Log Adapter:

- com.hp.umb.adapter.log.LogAlarmConsumer: this class is associated with the UcaStaticForwarderFlow flow
- com.hp.umb.adapter.log.LogEventConsumer: this class is associated with the UcaStaticEventForwarderFlow flow

These classes are implemented in such a way that each message consumed from the collection flow (alarm or event) is logged in a log file.

However, any other treatment could be implemented by some other customized message handlers.

[12] Consumer message handler classes must implement the `com.hp.umb.adapter.consumer.ConsumerMessageHandlerInterface` Java interface

[13] Alarms are objects that implement the `com.hp.uca.expert.alarm.AlarmCommon` Java interface

[14] Events are objects that implement the `com.hp.uca.expert.event.Event` Java interface

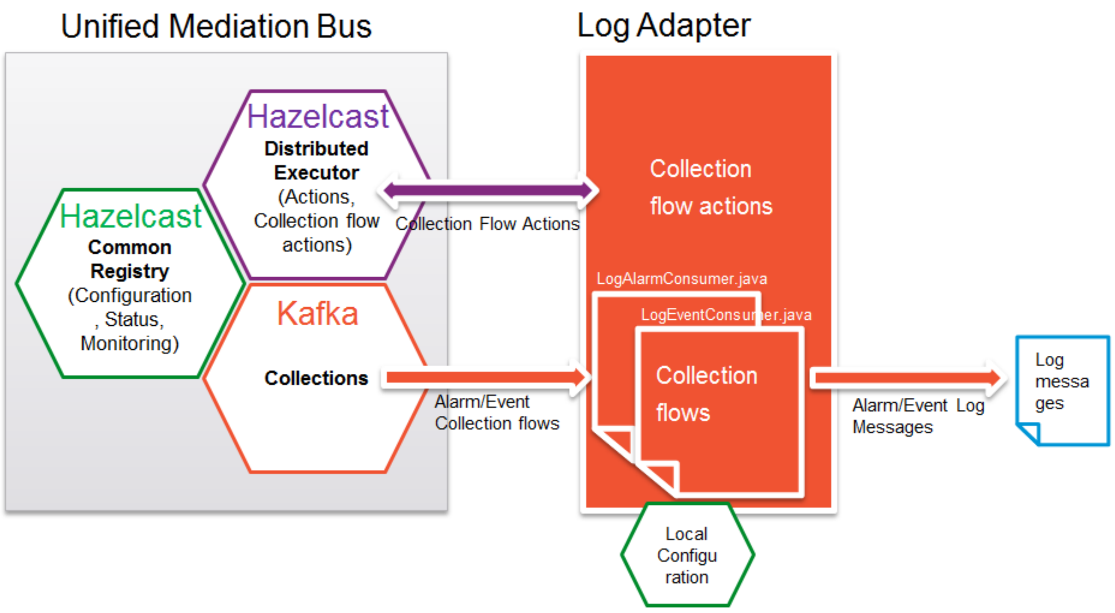


Figure 62 - Log Adapter consuming alarms/events collections

Appendix A

5.4 Maven *`pom.xml`*

The Adapter examples provided with UMB come with a Maven `pom.xml` file that can build and package the project as described in this document.

Following is the list of Apache Maven targets that can be executed from the command line using the **mvn** tool:

```
# mvn clean
```

Removes all files created during the build from the build directory.

```
# mvn compile
```

Compiles all Java files of the project.

```
# mvn test
```

Runs the JUnit tests defined in the project.

```
# mvn package
```

Build the final, “ready to deploy” Adapter ZIP file without cleaning the folder 'target'

```
# mvn clean package
```

Is equivalent to executing the following targets: “clean”, “compile”, “test” and “package”.

Chapter 6 Glossary

UCA: Unified Correlation Analyzer

EBC: Event Based Correlation

IDE: Integrated Development Environment

JMS: Java Messaging Service

JMX: Java Management Extension, used to access or process action on the UMB product.

JNDI: Java Naming and Directory Interface

Inference engine: Process that uses a Rete algorithm for expert behavior

DRL: Drools Rule file

XML: Extensible Markup Language

XSD: Schema of an XML file, describing its structure

X.733: Standard describing the structure of an Alarm used in telecommunication environment.

EVP: UMB Value Pack

DSL: Domain Specific Language

API: Application Programming Interface

URI: Uniform Resource Identifier

CSV: comma-separated values