
HP Unified Mediation Bus



Unified Mediation Bus Version 1.0

Adapter Development Guide

Edition: 1.0

For Windows® and Linux (RHEL 6.5) Operating Systems

September 2015

© Copyright 2015 Hewlett-Packard Development Company, L.P.

Legal Notices

Warranty

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

License Requirement and U.S. Government Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notices

© Copyright 2015 Hewlett-Packard Development Company, L.P.

Trademark Notices

HP-UX Release 10.20 and later and HP-UX Release 11.00 and later (in both 32 and 64-bit configurations) on all HP 9000 computers are Open Group UNIX 95 branded products.

Java™ is trademark of Oracle and/or its affiliates.

Microsoft®, Internet Explorer, Windows®, Windows Server 2012®, Windows XP®, and Windows 7® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX® is a registered trademark of The Open Group.

X/Open® is a registered trademark, and the X device is a trademark of X/Open Company Ltd. in the UK and other countries.

Red Hat® is a registered trademark of the Red Hat Company.

Linux® is a registered trademark of Linus Torvalds in the U.S. and other countries.

Eclipse™ is a trade mark of The Eclipse Foundation.

Smarts® is a registered trademark of EMC Corporation.

Netcool® is a registered trademark of Micromuse Ltd.

Hazelcast® is a registered trademark of Hazelcast, Inc

Zookeeper™ is a trademark of The Apache Software Foundation (ASF).

Kafka™ is a trademark of The Apache Software Foundation (ASF).

Apache Camel, Camel, Apache are trademarks of The Apache Software Foundation.

Apache Bigtop™ is a trademark of the Apache Software Foundation.

Contents

Preface	9
Chapter 1.....	11
Introduction	11
Overview	11
1.1 The Mediation Common Registry.....	13
1.1.1 Using Hazelcast® for Actions implementation	13
1.2 The Message Broker.....	14
Unified Mediation Bus principles	15
1.3 The Unified Mediation Bus Server (broker).....	15
1.4 The Unified Mediation Bus Adapters.....	15
1.5 The Unified Mediation Bus Adapter Services	18
1.5.1 Action Services	18
1.5.2 Flow Services	20
1.5.3 Defining Flow Service Consumers from Configuration	23
1.6 The Unified Mediation Bus Messages	26
1.6.1 Logging and testing considerations (UCA-EBC)	26
Chapter 2.....	29
Getting started with Unified Mediation Bus Development Kit	29
2.1 Installing the Unified Mediation Bus Adapter Development Kit	29
2.2 Adapter Development Pre-requisites.....	29
2.2.1 Eclipse IDE	29
2.2.2 Post-install Environment Setup.....	30
2.2.3 Unified Mediation Bus Eclipse plug-in installation instructions	31
Chapter 3.....	35
Unified Mediation Bus Adapters development	35
3.1 Creating a new UMB Adapter	35
3.1.1 Creating a UMB Adapter project within Eclipse	35
3.1.2 Anatomy of the created project	39
3.1.3 Validation of the created project	40
3.2 Customizing the created UMB Adapter project.....	41
3.2.1 Customizing the Adapter Name	41
3.2.2 Adding producer collection flow services	42
3.2.3 Adding action services.....	47
3.2.4 Adding consumer flows.....	52
3.3 Generating the UMB Adapter kit	58
3.4 Installing the UMB Adapter kit	60
3.5 Starting the UMB Adapter kit	60
Chapter 4.....	62
Advance development topics	62

4.1	Extending the DefaultEvent class.....	62
4.1.1	Defining the new metric schema.....	62
4.1.2	Generating the new metric java class	64
4.1.3	Adding marshaller and unmarshaller for the new metric.....	66
4.1.4	Generating the new metric jar file.....	67
4.2	Customizing the serialization Class	68
4.3	Discovering the solution's adapter topology and states	70
4.3.1	Getting the list of known Adapters	70
4.3.2	Getting Adapter's Notifications.....	70
Chapter 5.....		72
Unified Mediation Bus sample Adapters		72
5.1	Camel Adapter.....	72
5.1.1	Configuration.....	73
5.1.2	How does it work?	78
5.1.3	JUnit tests.....	81
5.2	File Adapter	82
5.2.1	Configuration.....	83
5.2.2	How does it work?	86
5.2.3	JUnit tests.....	88
5.3	Log Adapter.....	88
5.3.1	Configuration.....	89
5.3.2	How does it work?	91
Appendix A		93
A.	Ant <i>build.xml</i> targets	93
Glossary.....		94

Figures

Figure 1 Unified Mediation Bus architecture overview	12
Figure 2: Mediation Common Registry overview	13
Figure 3: Unified Mediation Bus Action Mechanism overview	14
Figure 4: Unified Mediation Bus flows overview	15
Figure 5 - Action execution without load balancing	17
Figure 6 - Action execution with load balancing	17
Figure 7: Example of an Action Service Definition	19
Figure 8 - Example of “auto” consumer flows in the AdapterConfiguration.xml file	25
Figure 9- Example of autoNonUmbConsumer flow definition in the AdapterConfiguration.xml file	26
Figure 10 - Unified Mediation Bus plug-in: Installation step 1	31
Figure 11 - Unified Mediation Bus Eclipse plug-in: Installation step 2	32
Figure 12 - Unified Mediation Bus Eclipse plug-in: Installation step 3	33
Figure 13 - UMB Adapter project creation wizard Step1	37
Figure 14 – New UMB Adapter project	38
Figure 15 - Folder structure of the new UMB Adapter project	39
Figure 16 - Adapter.java Java class of the new UMB Adapter project	40
Figure 17 - AdapterTest.java JUnit test class of the new UMB Adapter project	41
Figure 18 - Customizing the Adapter Name	42
Figure 19 - Example of a flow in the AdapterConfiguration.xml file	43
Figure 20 - Creating a “Collector” Java class – Step 1	44
Figure 21 - Creating a “Collector” Java Class Step 2	44
Figure 22 - Creating a “Collector” Java class – Step 3	45
Figure 23 - Example of an action in the AdapterConfiguration.xml file	48
Figure 24 - Creating an “Action” Java class – Step 1	49
Figure 25 - Creating an “Action” Java class – Step 2	49
Figure 26 - Creating an “Action” Java class – Step 3	50
Figure 27 - Java code to send action requests	51
Figure 28 - Java code to create consumer flows	53
Figure 29 - Creating a “MessageConsumer” Java class – Step 1	54
Figure 30 - Creating a “MessageConsumer” Java class – Step 2	55
Figure 31 - Creating a “MessageConsumer” Java class – Step 3	56
Figure 32 - Consumer Flow status diagram	57
Figure 33 - Building the kit of your Adapter	58
Figure 34 – Location of the kit of your Adapter	59
Figure 35 - Contents of the kit of your Adapter	60
Figure 36 - JAXB Diagram	63
Figure 37 - Camel adapter overview	73
Figure 38 - The Camel Adapter’s AdapterConfiguration.xml file	74
Figure 39 - The Camel Adapter’s camel-context.xml file	75
Figure 40 - “camel-actions” route in the camel-context.xml file	76
Figure 41 - “camel-collectionactions” route in the camel-context.xml file	77
Figure 42 - “camel-collection” route in the camel-context.xml file	78
Figure 43 - Processing Actions in the Camel Adapter	79
Figure 44 - Processing Collection Flow Actions in the Camel Adapter	80
Figure 45 - Processing Collections in the Camel Adapter	81
Figure 46 - File adapter overview	83
Figure 47 - The File Adapter’s AdapterConfiguration.xml file	84
Figure 48 - File Adapter’s “alarms.xml” data file	85
Figure 49 - File Adapter’s “temperatures.csv” data file	86
Figure 50 - File Adapter’s alarms collections	87
Figure 51 - File Adapter’s temperatures collections	87
Figure 52 - Log adapter overview	89
Figure 53 - The Log Adapter’s AdapterConfiguration.xml file	90
Figure 54 - Log Adapter consuming alarms/events collections	92

Tables

Table 1 - Software versions	9
Table 2 - Eclipse IDE Prerequisites for UMB Adapter Development Kit	29

Preface

This guide provides an overview of the Unified Correlated Analyzer Mediation 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: V1.0

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 V1.0	<ul style="list-style-type: none">• Windows XP / Vista• Windows Server 2007• Windows 7• Red Hat Enterprise Linux Server release 6.5

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:

References

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

Support

Please visit our HP Software Support Online Web site at www.hp.com/go/hpssoftwaresupport 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.

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.

Overview

Unified Mediation Bus allows several applications to exchange Events (and by extension Alarms) with each other. It also provides facilities 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:
 - a common registry feature that centralizes configuration, status and monitoring information on all UMB Adapters that are part of the overall UMB solution
 - 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 1 or more dedicated UMB Server host(s), that contains Apache Kafka / Apache ZooKeeper

- Several UMB Adapter¹ 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.

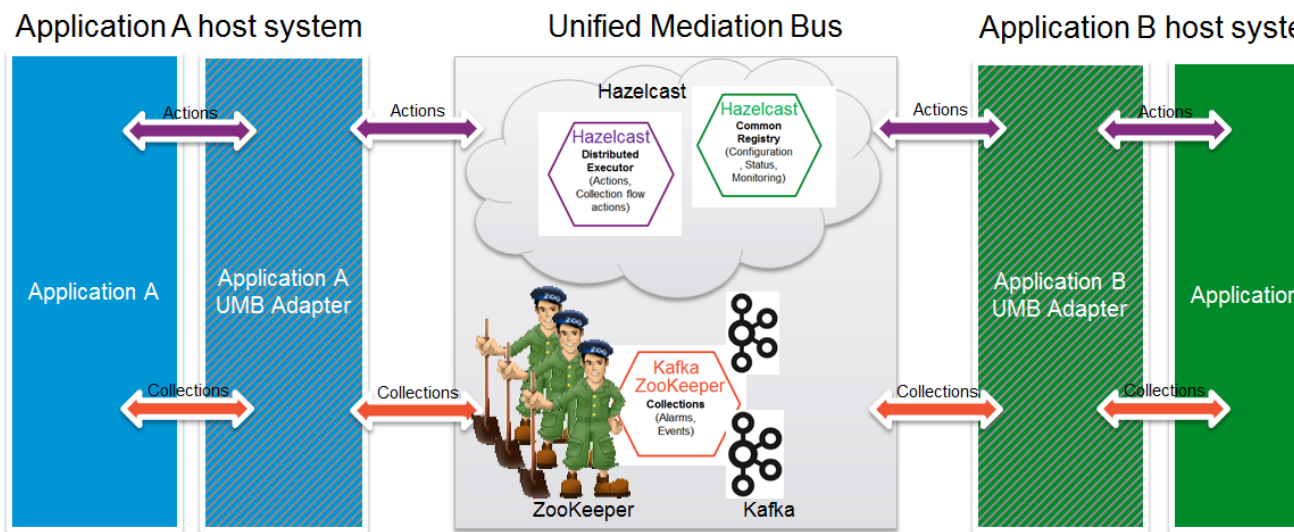


Figure 1 Unified Mediation Bus architecture overview

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 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*

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 communication with another one.

The Common Registry can be schematically represented as follows:

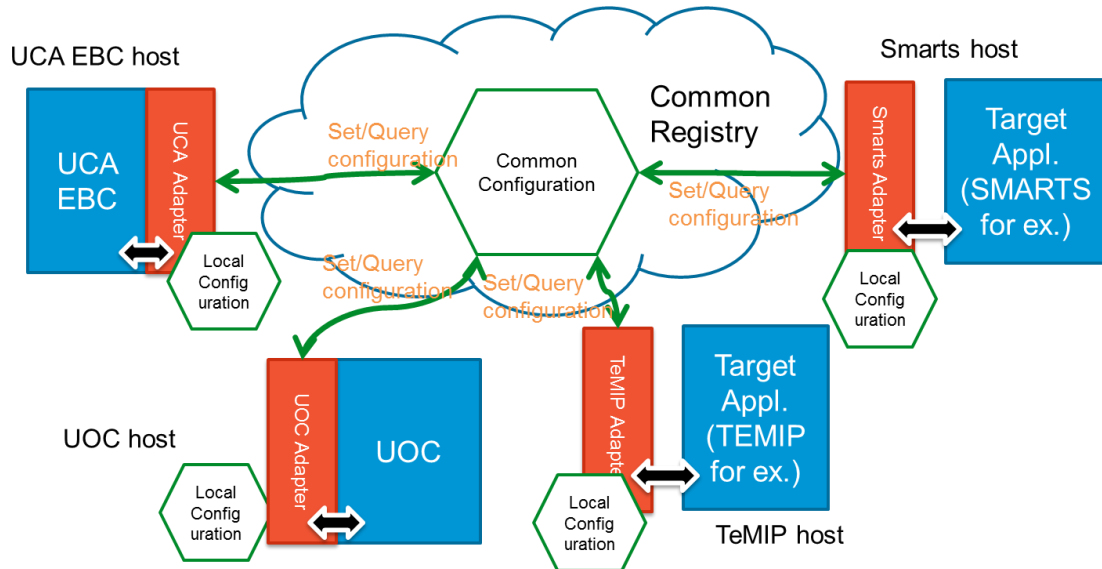


Figure 2: Mediation Common Registry overview

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 facility 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.

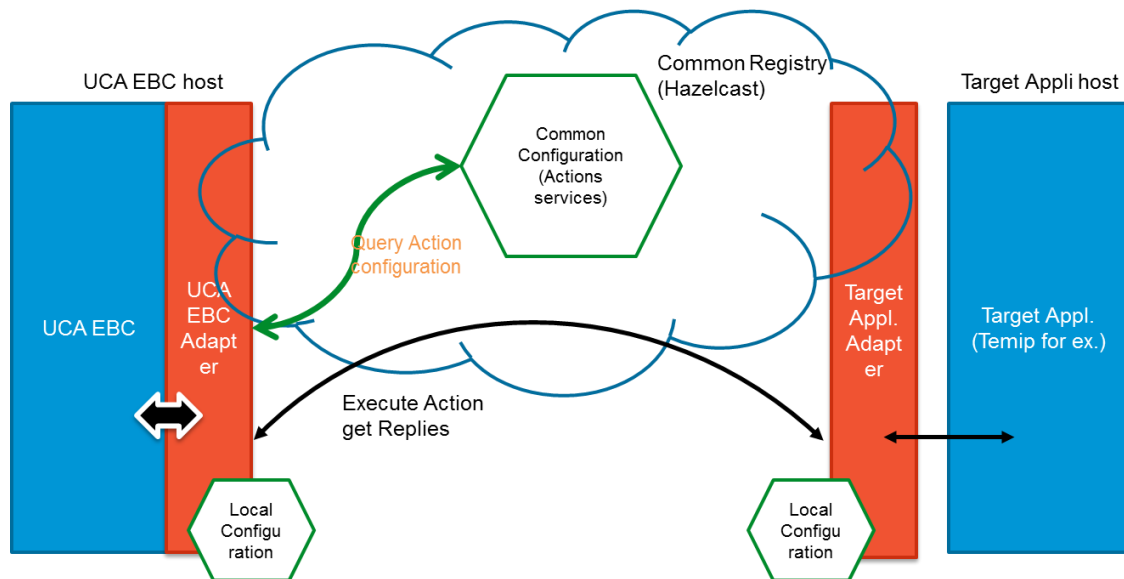


Figure 3: Unified Mediation Bus Action Mechanism overview

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 though the use of ZooKeeper).

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

The collection flows are of two types:

- **Static flows**
 - ✓ One Producer for several possible Consumers. Each of the consumers 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.
 - ✓ 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

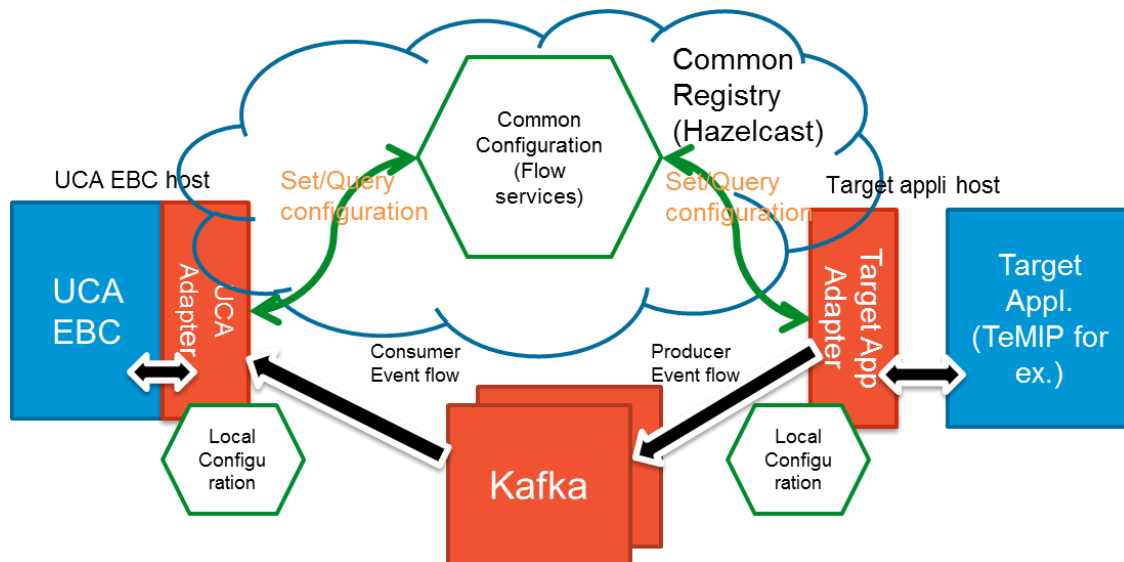


Figure 4: Unified Mediation Bus flows overview

Unified Mediation Bus principles

1.3 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.

1.4 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.

Embedded adapters are preferable to Standalone Adapters and recommended whenever possible.

- **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 is 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 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:

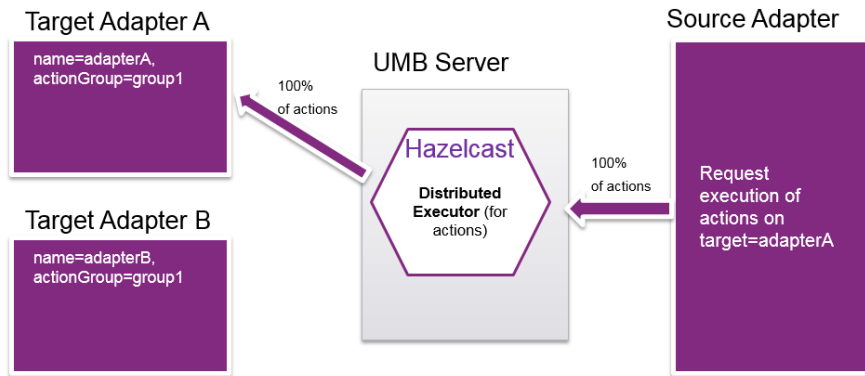
- **name**

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. Action groups exist so that they can be used as targets (instead of adapters) for executing actions. For example, if you have several adapters in your UMB solutions that share the same action group, you can request execution of actions on the action group itself. This is done by specifying an action group (instead of an adapter) as the target of the action. The action itself 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):

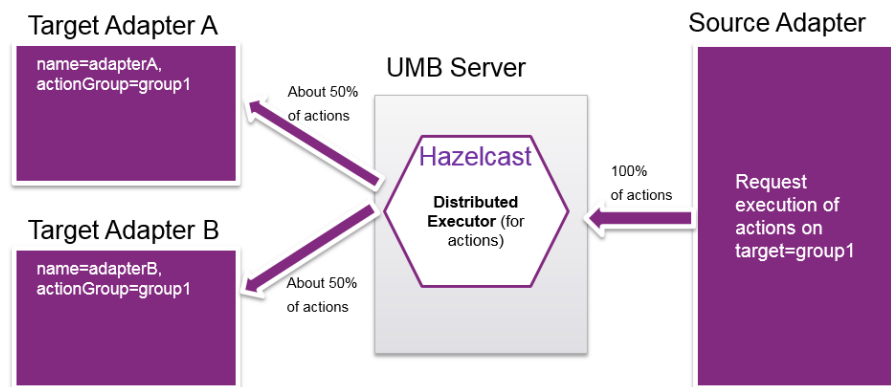


100% of actions targeted to Adapter A will be executed on Target Adapter A

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):



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

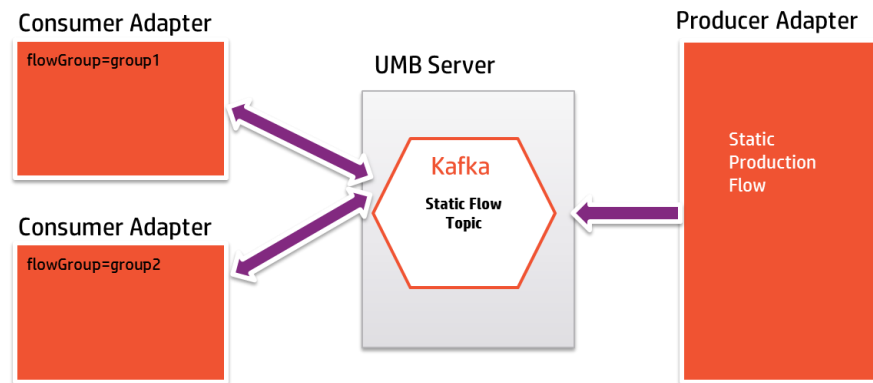
In the figure above, load balancing of actions is used and on average 50% of actions are executed on each adapter part of the targeted action group.

- **flowGroup**

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

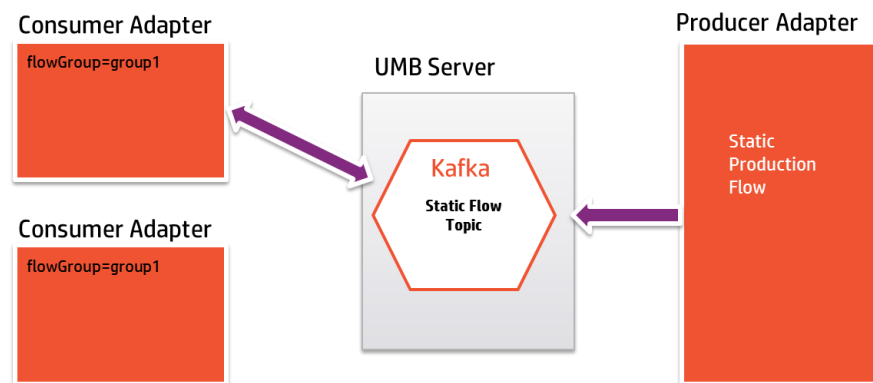
The flowGroup has a meaning only for Static flows for which we can have several consumers for one producer. Each message published to a StaticFlow is delivered to one instance within each subscribing consumer group.

- ✓ 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 belonging to different flow groups, 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. As the UMB flows have single one partition, 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 lose the connection with the kafka server, then this second adapter will get the messages that were not previously collected by the first adapter.

1.5 The Unified Mediation Bus Adapter Services

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

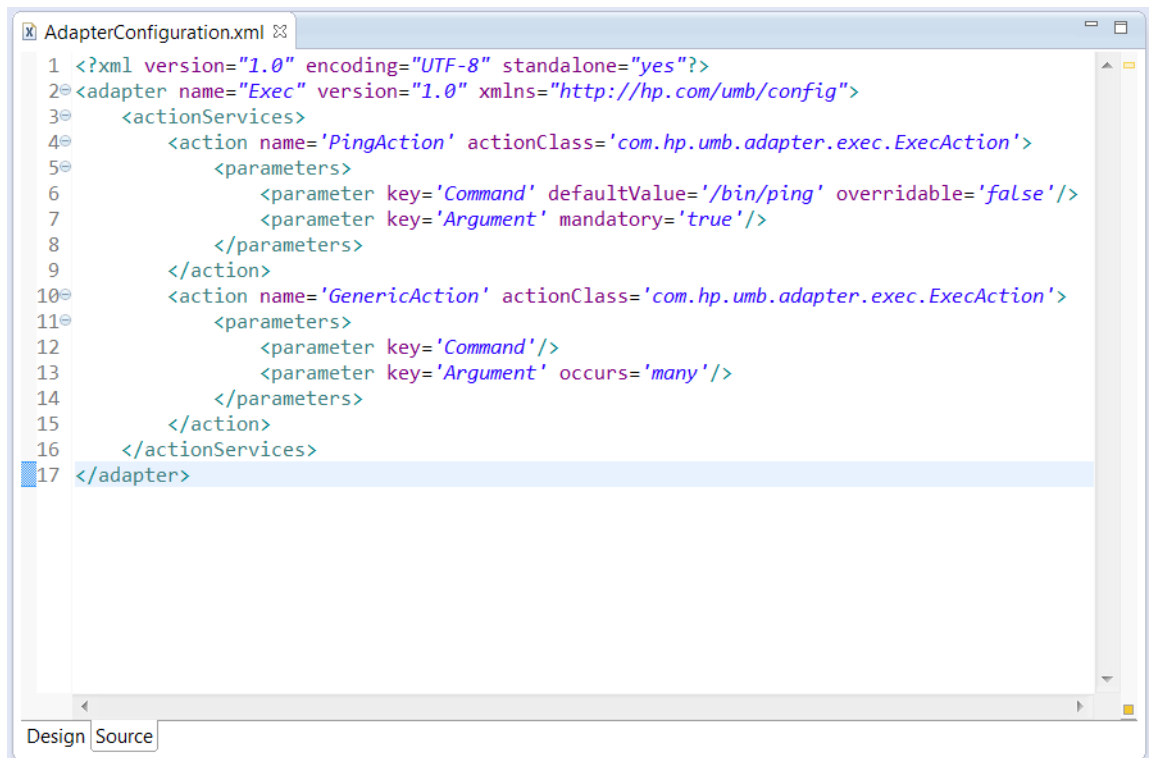
- **Action Services**
- **Flow Services**

1.5.1 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:



```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <adapter name="Exec" version="1.0" xmlns="http://hp.com/umb/config">
3   <actionServices>
4     <action name='PingAction' actionClass='com.hp.umb.adapter.exec.ExecAction'>
5       <parameters>
6         <parameter key='Command' defaultValue='/bin/ping' overridable='false' />
7         <parameter key='Argument' mandatory='true' />
8       </parameters>
9     </action>
10    <action name='GenericAction' actionClass='com.hp.umb.adapter.exec.ExecAction'>
11      <parameters>
12        <parameter key='Command' />
13        <parameter key='Argument' occurs='many' />
14      </parameters>
15    </action>
16  </actionServices>
17 </adapter>
```

Figure 7: 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).

A `<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. 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 1.5.1.1 "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.

1.5.1.1 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:** 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. It is optional to specify a value for the 'defaultValue' attribute.

- **overridable:** 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'.

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

- **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 an action failure.

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

- **mandatory:** 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 example, 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.

1.5.2 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' and must specify a 'collectorClass' (i.e. the class that will implement the production 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 time period.

Several consumers can consume the same static production flow. Each of the consumers will receive all the produced messages. 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):

"provider 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 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 follow (without the quotes):

"consumer Adapter name"- "requester Identifier"- "provider 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:

- **name:** The name of the flow. This name will be referenced by UMB Adapters wishing to consumer this flow. It is mandatory to specify a value for the 'name' attribute.
- **type:** The type of the flow: either "Static" or "Dynamic" (see the differences between static and dynamic flows above). There is no default value. It is mandatory to specify a value for the 'type' attribute.
- **collectorClass:** The full name of Java class that implements the flow producer. There is no default value. It is mandatory to specify a value for the 'collectorClass' attribute.
- **monitoringRestartPeriod (optional):** 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.
- **autoStarted(optional):** This is a Boolean attribute. When set to true (the default) the producer flow starts automatically at adapter startup.
- **lastEventReceivedFirstDuringResynchronization (optional):** This is a Boolean attribute. When set to false (the default) messages are produced by the flow in chronological order during a resynchronization, i.e. the oldest messages are sent first. When set to true, the reverse chronological order is used, i.e. the last messages are sent first.
- **serializerClass:** The full name of Java class that serializes the flow messages. It is optional to specify a value for the 'serializerClass' attribute. When no value is specified, a default serializer class is used.

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

1.5.2.1 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:** the 'defaultValue' 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 again.

1.5.3 Defining Flow Service Consumers from Configuration

If the `AdapterConfiguration.xml` file allows defining flow services (production side), it also allows defining 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.ConsumerMessageHandlerInterface` `<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).

Refer to section "3.2.4.3 Defining the flow message consumer class" for full description on how to define a Message Consumer object.

The following optional attribute can be defined:

- **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 if they fail.
- **monitoringRestartPeriod (in milliseconds default 30000):** in case of Consumer Flow start failure, if the Flow is monitored (the default), this represents restart attempt period.

- **messageConsumerTimeout (in milliseconds default 1000):** this attribute is applicable only if the specified messageConsumerClass implements the `com.hp.umb.adapter.consumer.ConsumerMessageSetHandlerInterface<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.ConsumerMessageSetHandlerInterface<K extends Event>`. It indicates the maximum size of the message set to return.
- **serializerClass:** this parameter can specify a custom serialization Class. The serialization class is the class in charge of linearizing (de-linearizing) the Event message into (and from) a byte array. The default linearization class is the UMB framework provided `com.hp.umb.adapter.internal.utilities.JavaClassSerializer` class which uses the standard java class linearization mechanism.

A custom linearization class must implement the following interfaces:

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

and

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

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 “auto” consumer flows:



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

Figure 8 - 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”

1.5.3.1 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 producers.

In such 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 tag `<autoNonUMBConsumer ... />`

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 than for the standard `autoConsumers`. No parameters can be defined.

Note

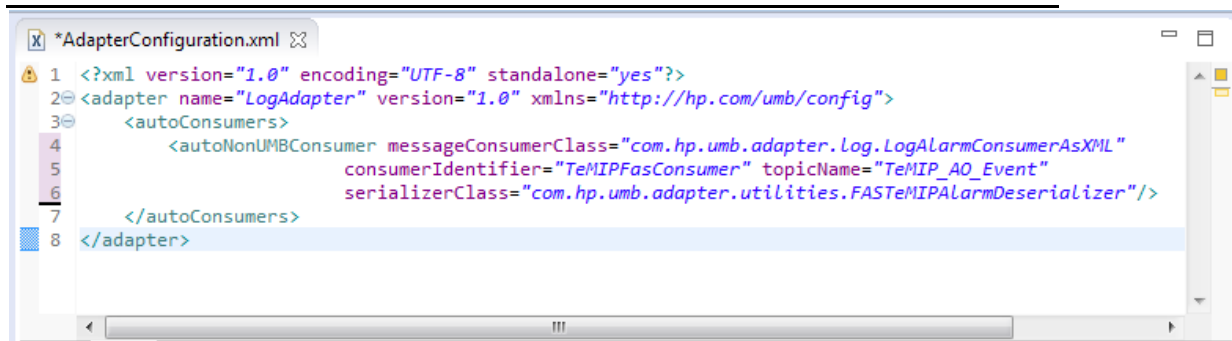
When using non UMB flows a specific attention must be paid to the deserialization process.

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 it into a java class.

Such de-serializer is specified by serializerClass attribute.

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



```
*AdapterConfiguration.xml
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <adapter name="LogAdapter" version="1.0" xmlns="http://hp.com/umb/config">
3   <autoConsumers>
4     <autoNonUMBConsumer messageConsumerClass="com.hp.umb.adapter.Log.LogAlarmConsumerAsXML"
5                           consumerIdentifier="TeMIPFasConsumer" topicName="TeMIP_AO_Event"
6                           serializerClass="com.hp.umb.adapter.utilities.FASTeMIPAlarmDeserializer"/>
7   </autoConsumers>
8 </adapter>
```

Figure 9- Example of autoNonUmbConsumer flow definition in the AdapterConfiguration.xml file

1.6 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. The same way the Objects are de-serialized when read from the Kafka server on the consumer side.

1.6.1 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 collector logging feature that will dump the collected messages using XML marshalling.

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 the 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:extends>
          </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 insert the JAXB annotation in 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;
    }
}
```

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™ application development environment.

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

Software	Version
Eclipse IDE	3.7 (Indigo) 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 3.7 (Indigo) 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 Luna 4.4.

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.

Once Eclipse IDE is installed on your system, and in order to get the full benefit of the Drools development environment in Eclipse, it is also necessary to download and install the Drools plug-in for Eclipse.

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, perform the following command:

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

You should get an output similar to the following:

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

2.2.2.2 Ant Configuration

The UMB Adapter packaging is based on the use of the Apache Ant tool. This tool requires a specific version and specific settings. Be sure to use the Apache Ant tool provided with UMB in the %UMB_DEV_HOME%\3pp\ant directory (\${UMB_DEV_HOME}/3pp/ant on Linux).

Be sure that you don't have the ANT_HOME environment variable set to the path of another version of Apache Ant, which would create conflicts with the version of Apache Ant in the 3pp\ant\bin folder. If you do, you should either clear the ANT_HOME environment variable:

```
C:\> set ANT_HOME=
```

Or set it to the directory of the Apache Ant version that comes with the UMB development kit:

```
C:\> set ANT_HOME=%UMB_DEV_HOME%\3pp\ant
```

```
$ANT_HOME/bin/ant -version
Apache Ant(TM) version 1.9.3 compiled on December 23 2013
```

The delivered Apache Ant version that comes with the UMB development kit is:

```
# $ANT_HOME/bin/ant -version
Apache Ant(TM) version 1.9.3 compiled on December 23 2013
```

2.2.3 Unified Mediation Bus Eclipse plug-in installation instructions

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

This plugin is delivered in the %UMB_DEV_HOME%\eclipseplugin\ umbEclipsePluginSite-1.0.0-assembly.zip file.

The installation of this plug-in is made 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:

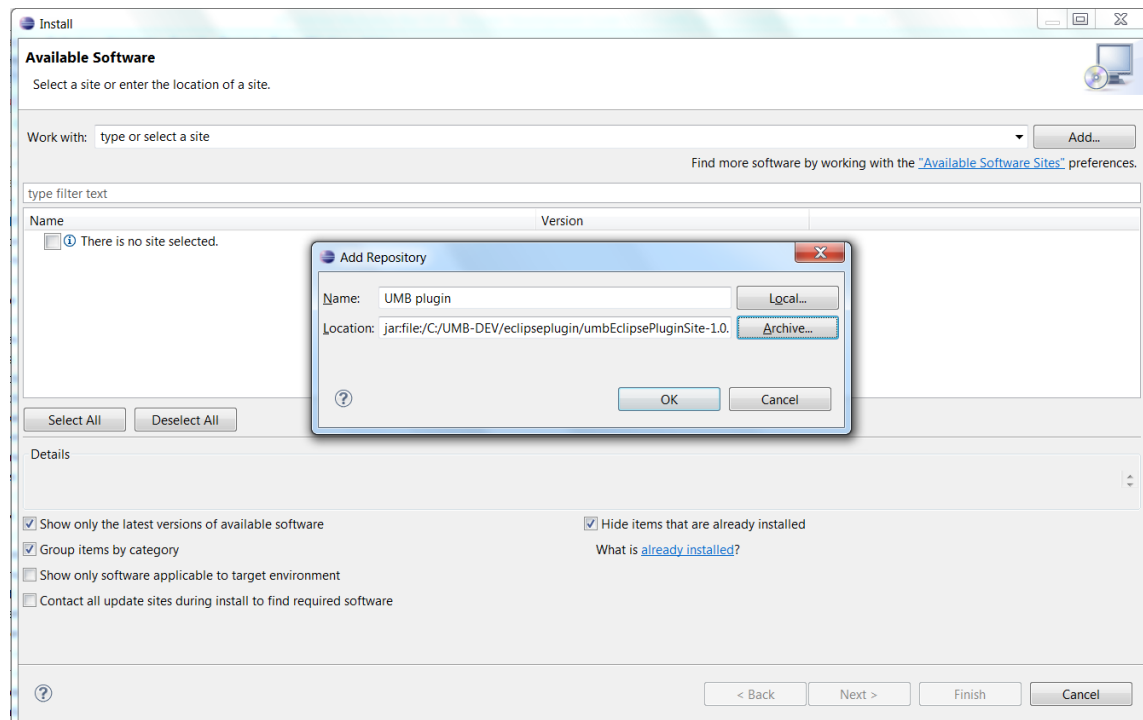


Figure 10 - Unified Mediation Bus plug-in: Installation step 1

Then click on the **OK** button.

The screen should then display the archive content as follow:

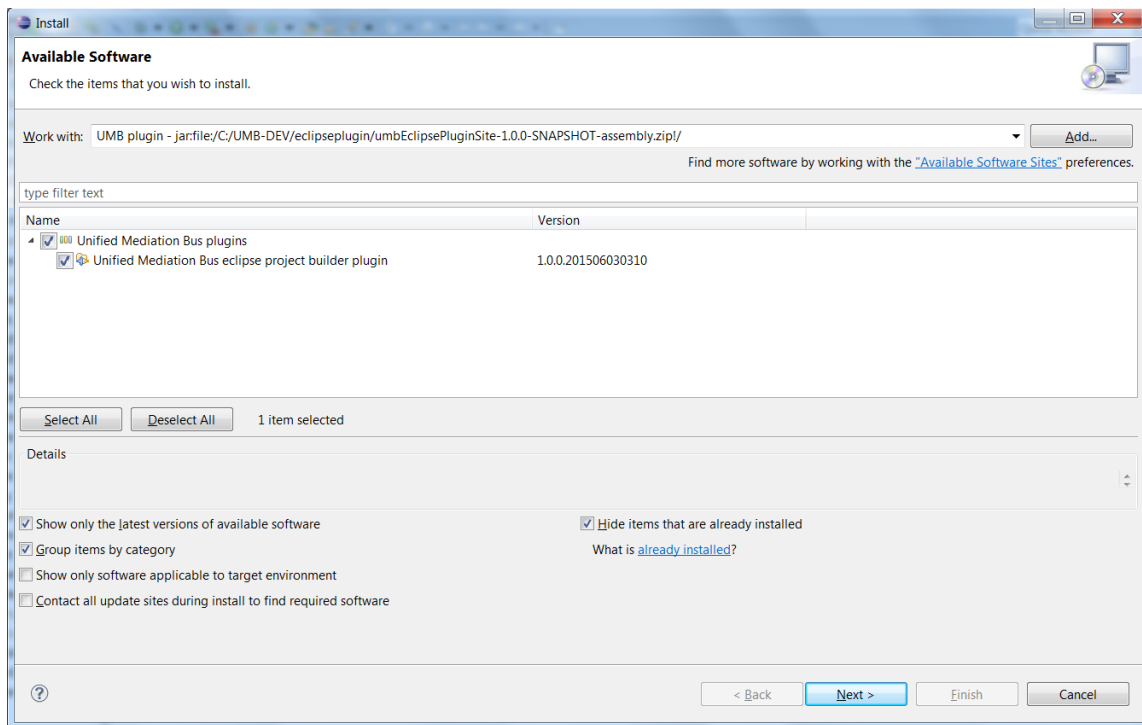


Figure 11 - Unified Mediation Bus Eclipse plug-in: Installation step 2

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:

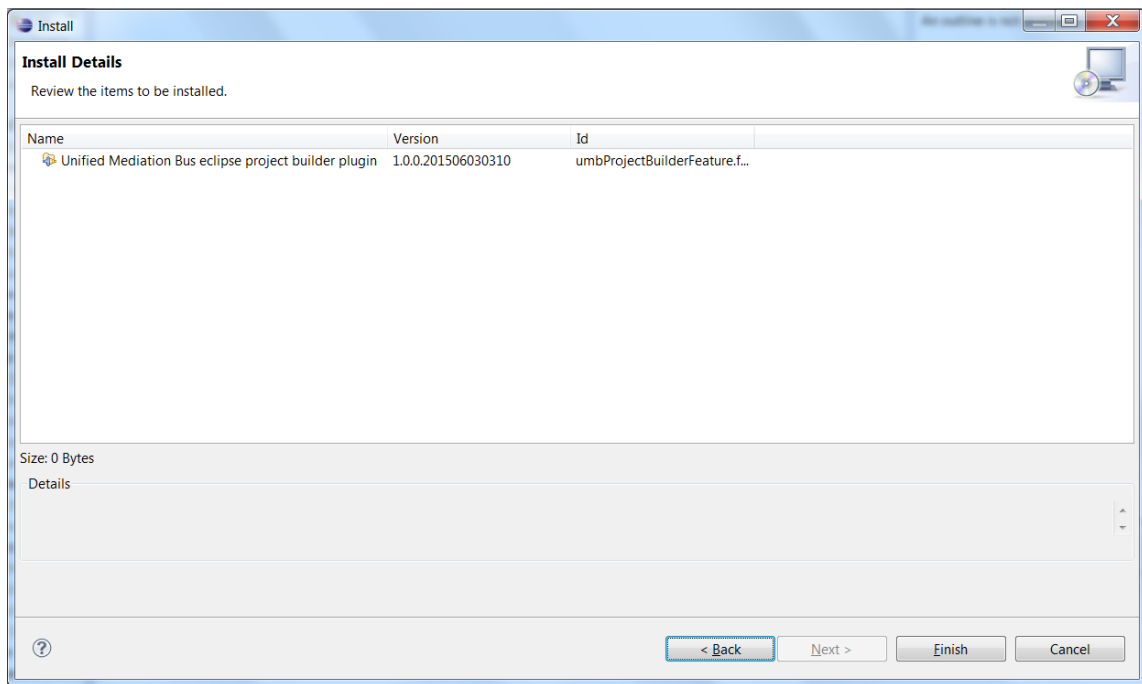
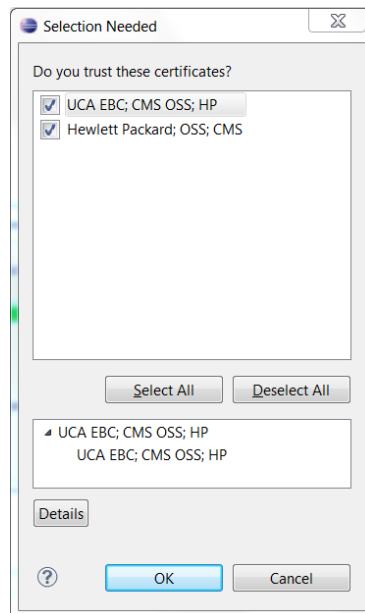


Figure 12 - Unified Mediation Bus Eclipse plug-in: Installation step 3

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

Note

The following message appears during the installation. This is a normal message as the provided jar files are signed.



Select the listed Certificated 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.

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:

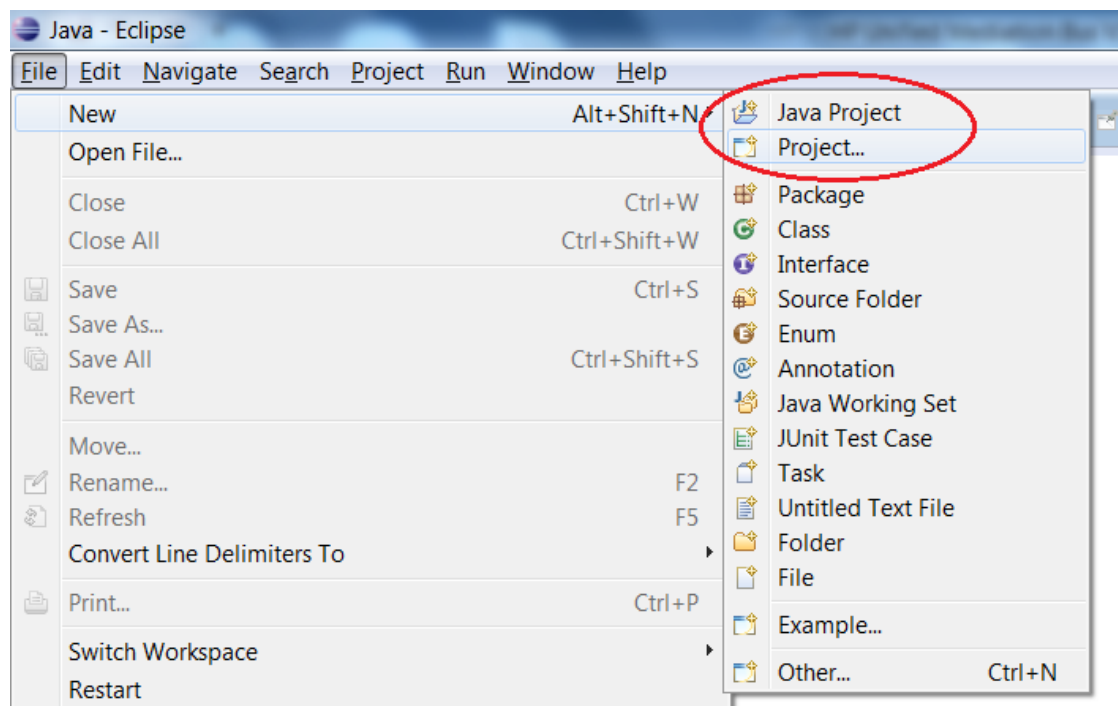
- Flow collection services
- Action services

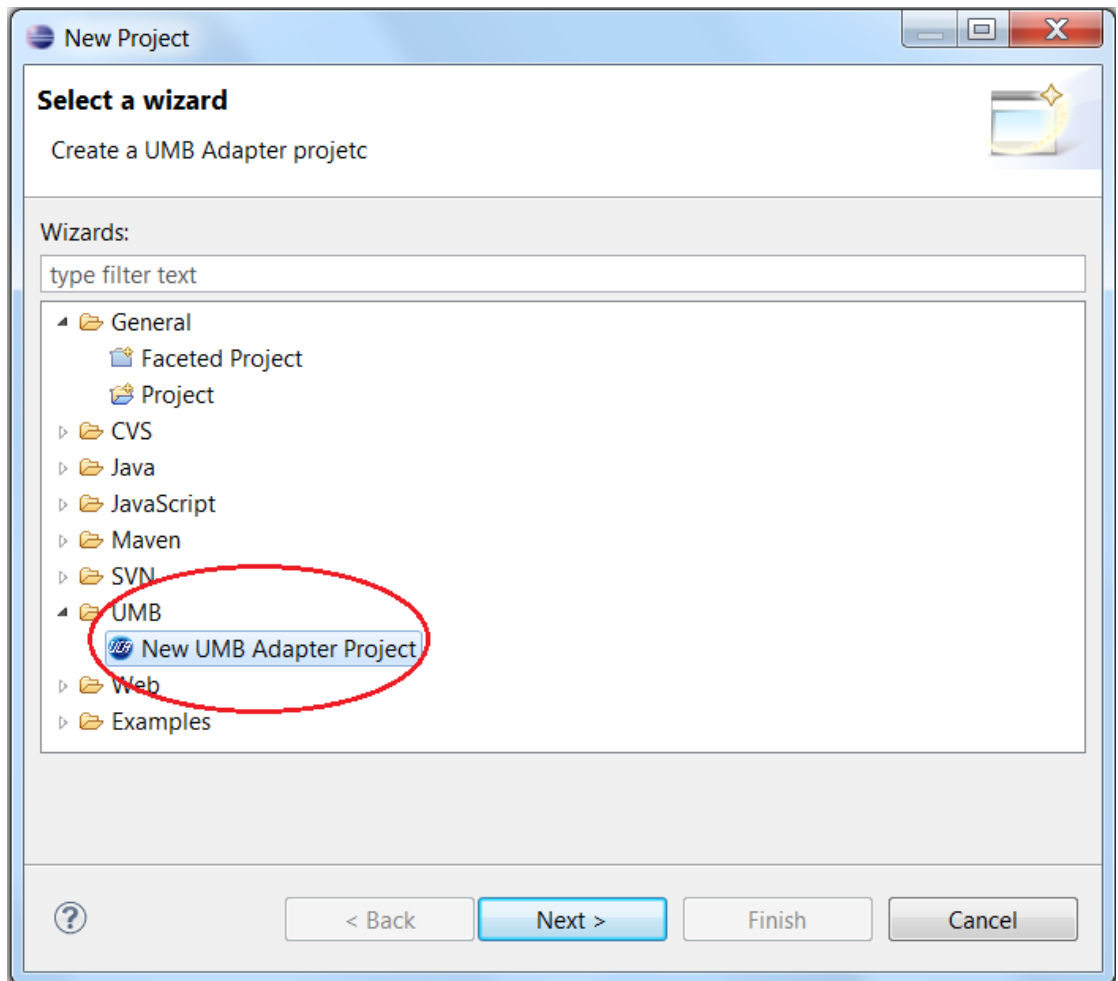
Each UMB Adapter can also act as a consumer of other UMB Adapters flow or action services.

3.1.1 Creating a UMB Adapter project within Eclipse

The UMB eclipse plug-in brings a project creation wizard that allows the creation of a new UMB Adapter project in just a few clicks and dialog boxes.

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





This launches the UMB Adapter Project wizard:

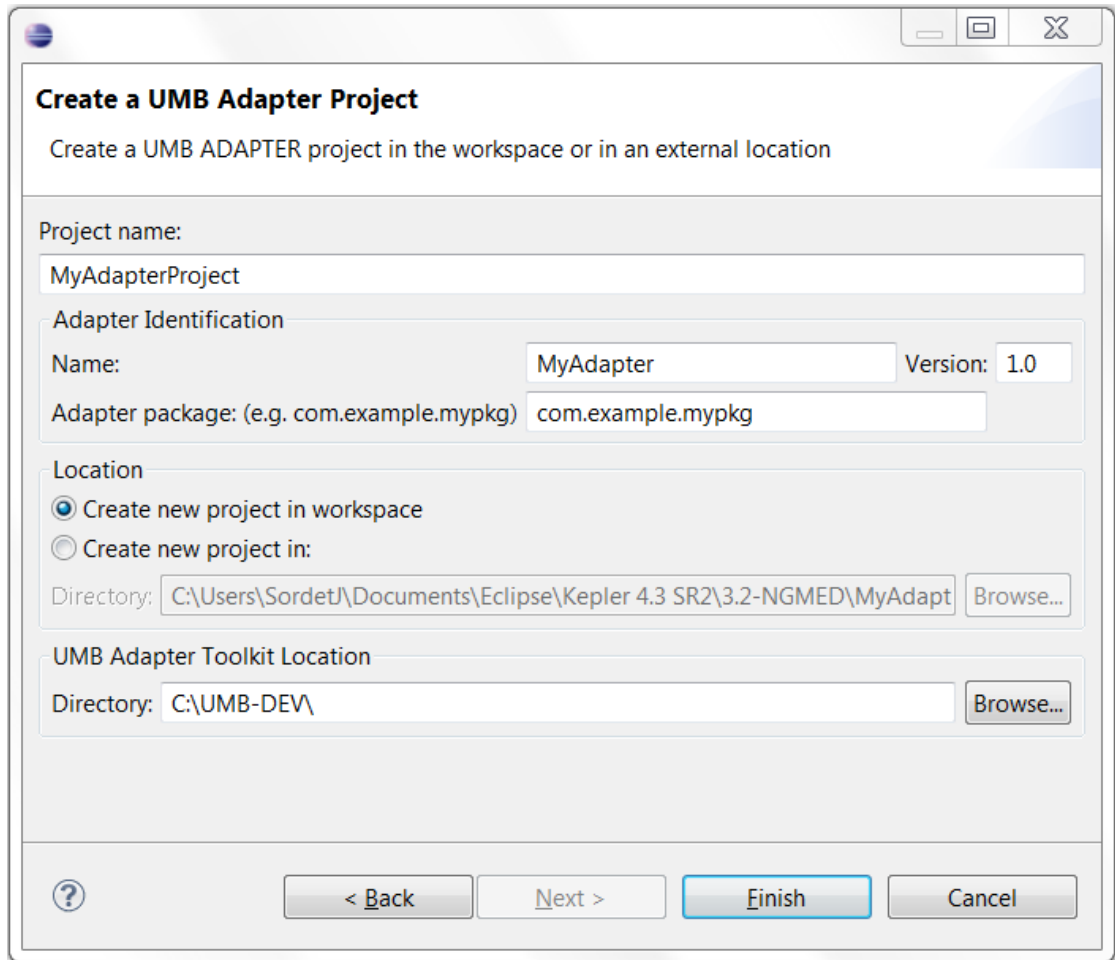


Figure 13 - UMB Adapter project creation wizard Step1

From the UMB Adapter project wizard window you can specify information regarding your UMB Adapter project:

- **Project name:** the name of the UMB Adapter Eclipse project to create
- **Adapter name and version:** the name and version of the UMB Adapter. The name of the UMB Adapter is used to identify the Adapter on the UMB framework
- **Adapter package:** the name of the Java package to be used for the Java classes of your UMB Adapter
- **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 Adapter Toolkit location:** location on the file system of the installation directory of the UMB Adapter Development Kit². 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)

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

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

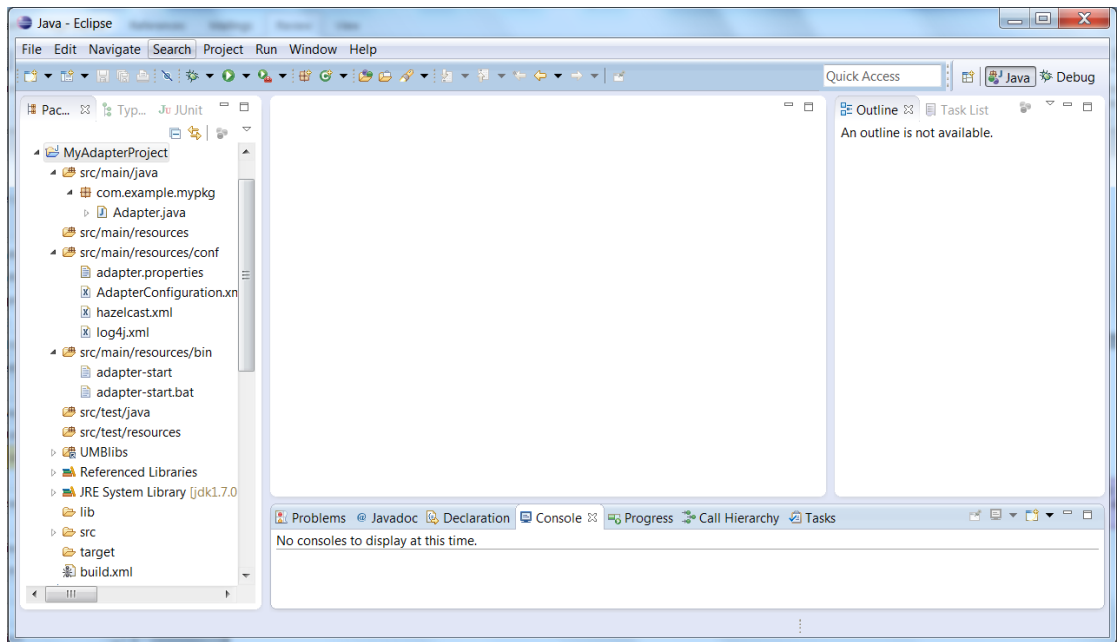


Figure 14 – New UMB Adapter project

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.

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

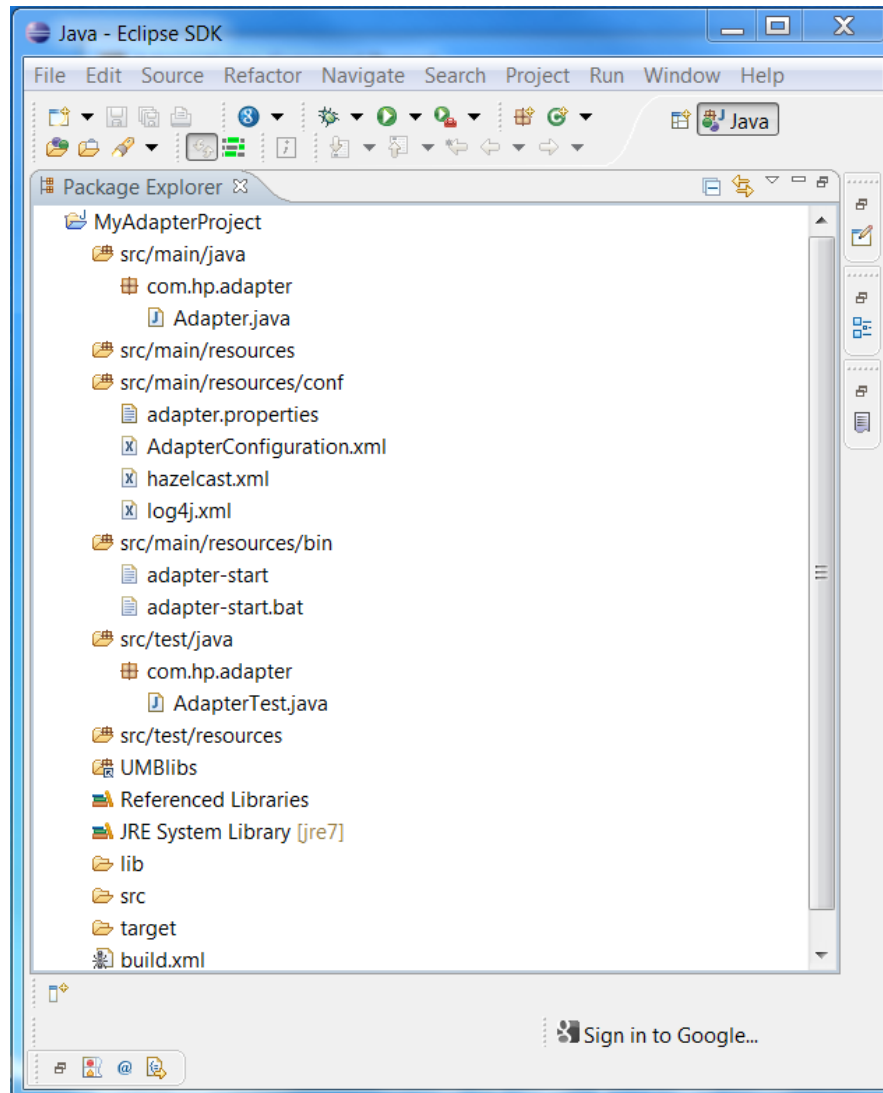


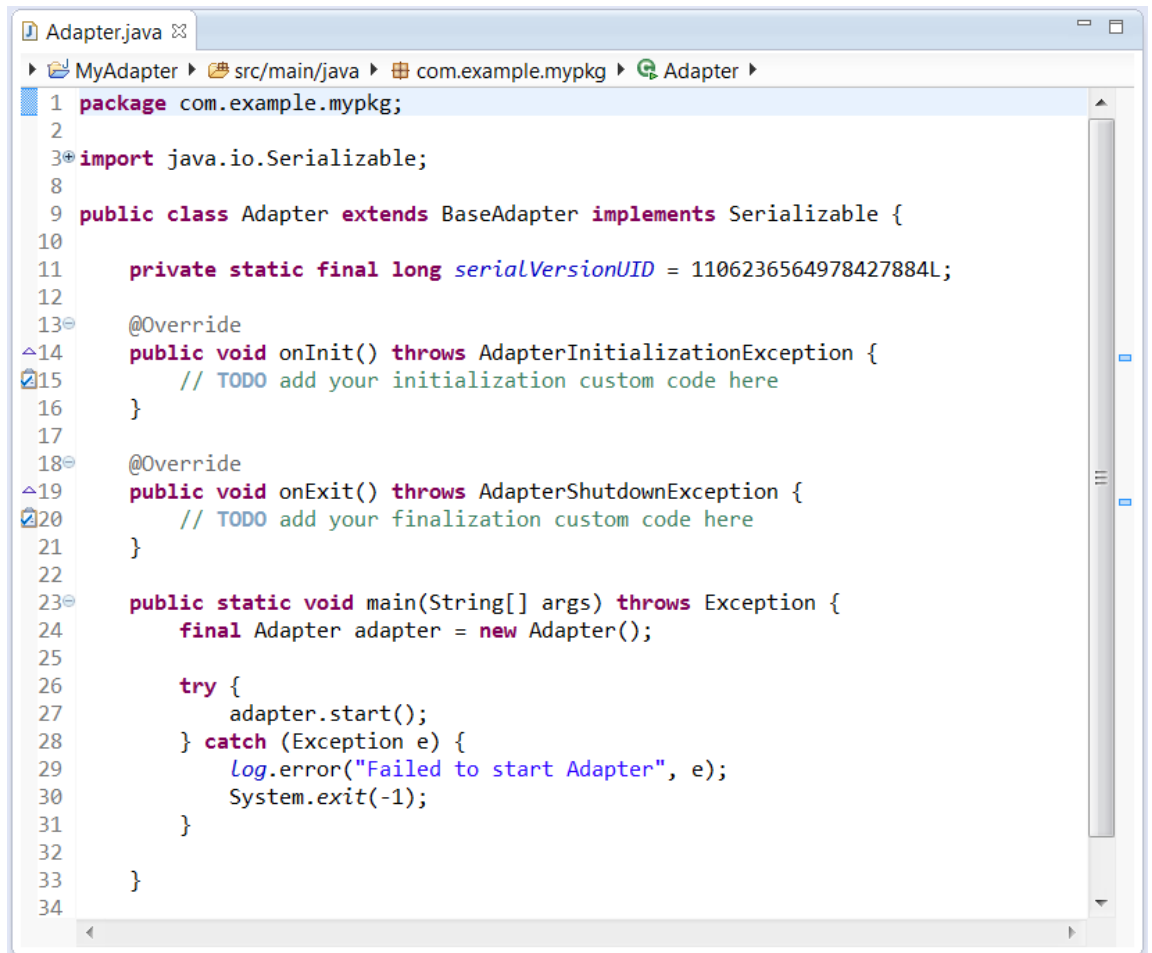
Figure 15 - Folder structure of the new UMB Adapter project

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 start scripts for both Windows (`adapter-start.bat` file) and Linux (`adapter-start` file) systems are located in the `src/main/resources/bin` folder.

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.



```
1 package com.example.mypkg;
2
3 import java.io.Serializable;
4
5
6
7
8
9 public class Adapter extends BaseAdapter implements Serializable {
10
11     private static final long serialVersionUID = 1106236564978427884L;
12
13     @Override
14     public void onInit() throws AdapterInitializationException {
15         // TODO add your initialization custom code here
16     }
17
18     @Override
19     public void onExit() throws AdapterShutdownException {
20         // TODO add your finalization custom code here
21     }
22
23     public static void main(String[] args) throws Exception {
24         final Adapter adapter = new Adapter();
25
26         try {
27             adapter.start();
28         } catch (Exception e) {
29             log.error("Failed to start Adapter", e);
30             System.exit(-1);
31         }
32     }
33 }
34
```

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

The created UMB Adapter project also comes with an Apache Ant `build.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 `AdapterTest.java`. This is a test skeleton 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).


```
1 package com.example.mypkg;
2
3 import static org.junit.Assert.assertEquals;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19 /**
20  * This is a JUnit test class of an UMB Adapter.
21  *
22  * @author UCA TEAM
23  */
24 public class AdapterTest {
25     static {
26         System.setProperty(
27             AdapterXmlConfiguration.ADAPTER_CONFIGURATION_PROPERTY,
28             "AdapterConfiguration.xml");
29     }
30
31     private static final Logger log = LoggerFactory
32         .getLogger(AdapterTest.class);
33     private static Adapter adapter = new Adapter();
34
35     static List<String> receivedAlarmIds;
36
37
38     @BeforeClass
39     public static void setUpBeforeClass() throws Exception {
40         try {
41             // Start Kafka and Zookeeper
42             EmbeddedKafka.startServer(2181, 9091);
43
44             // Start Adapter
45             adapter.start();
46         } catch (Exception e) {
```

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 a simple 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.BasicAdapter` 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 collection 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:

- **name:** this is the name of the flow used to identify the flow (for `CreateFlow` collection flow actions for example)
- **type:** the type of the flow: either **"Static"** or **"Dynamic"**. Static flows are started automatically when the Adapter starts whereas dynamic flows require a `CreateFlow` collection flow action to be started.
- **collectorClass:** this is the name of the Java class (including the Java package name) implementing the flow. For example: `com.example.MyCollector`. This class has to extend the `com.hp.umb.adapter.collector.BaseCollector` class and implement the `com.hp.umb.adapter.collector.CollectorInterface` Java interface.

The following optional attribute can be defined:

- **lastEventReceivedFirstDuringResynchronization:** a Boolean flag to indicate 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.
- **autoStarted (boolean default true):** This attribute is applicable only if the specified the flow type is "Static". It indicates that the UMB Framework should start this flow automatically at adapter startup.
- **monitoringRestartPeriod (in milliseconds default 30000):** This attribute is applicable only if the specified flow type is "Static" and `autoStarted='true'`. In case of Consumer Flow start failure, if the Flow is monitored (the default), this represents restart attempt period.

- **serializerClass:** this parameter can specify a custom serialization Class. The serialization class is the class in charge of linearizing (de-linearizing) the Event message into (and from) a byte array. The default linearization class is the UMB framework provided
`com.hp.umb.adapter.internal.utilities.JavaClassSerializer`
class which uses the standard java class linearization mechanism.

A custom linearization class must implement the following interfaces:

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

and

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

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. Each parameter must define all of the following mandatory attributes:

- **key:** this is the name of the action's parameter

The parameters defined in the `<parameters>...</parameters>` XML element will be used (alongside the parameters of the CreateFlow action if the flow is dynamic) when the flow is created.

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

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <adapter name="FileAdapter" version="1.0" xmlns="http://hp.com/umb/config">
3   <flowServices>
4     <flow name="AlarmFileStaticFlow" type="Static" collectorClass="com.hp.umb.adapter.file.FileCollector">
5       <parameters>
6         <parameter key="fileName" defaultValue="data/alarms.xml"/>
7       </parameters>
8     </flow>
9   </flowServices>
10 </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:

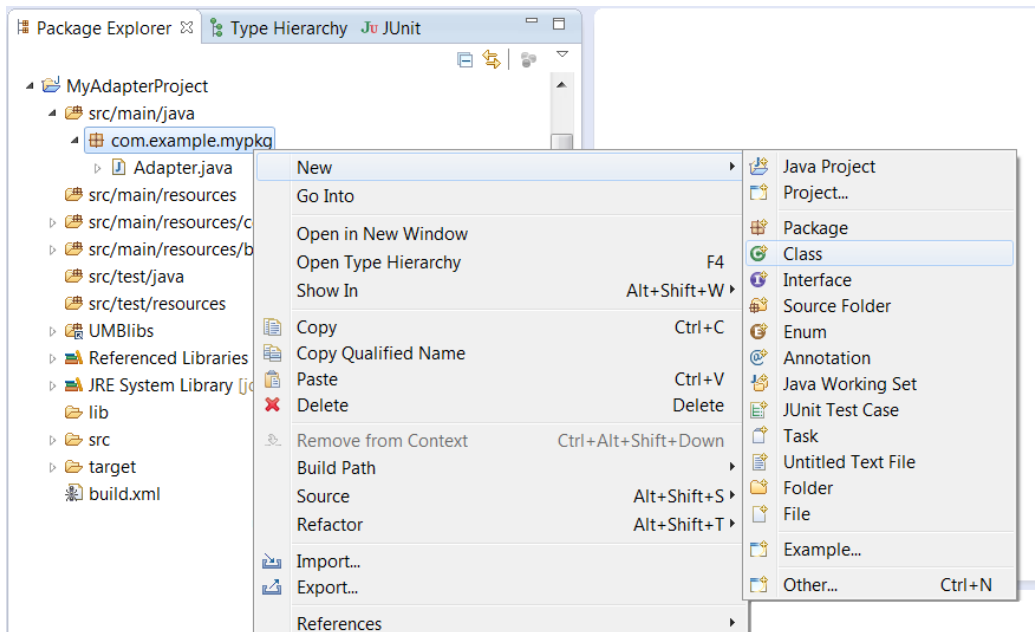


Figure 20 - Creating a “Collector” Java class – Step 1

This opens the New Class window.

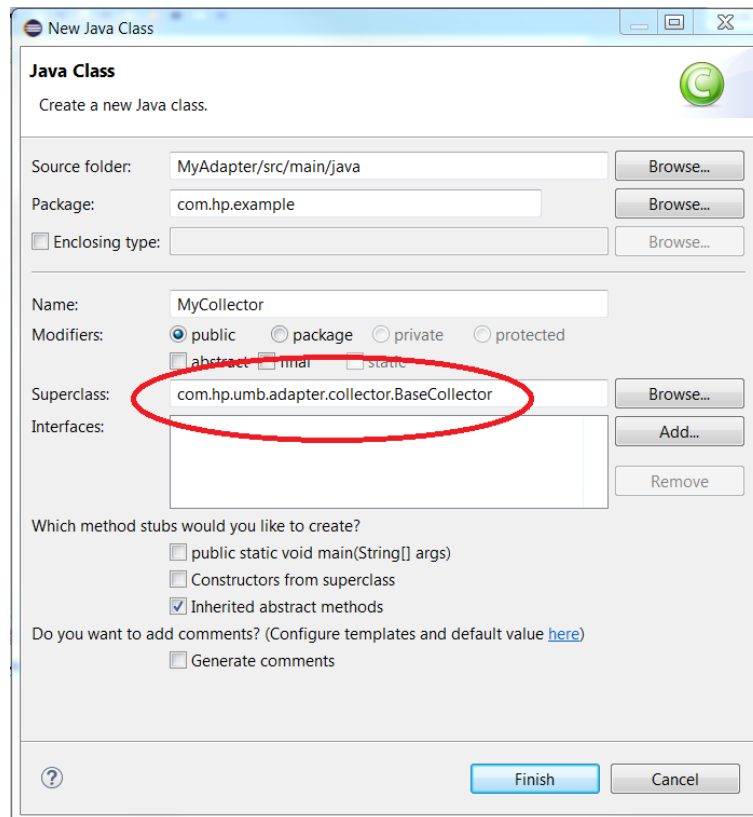
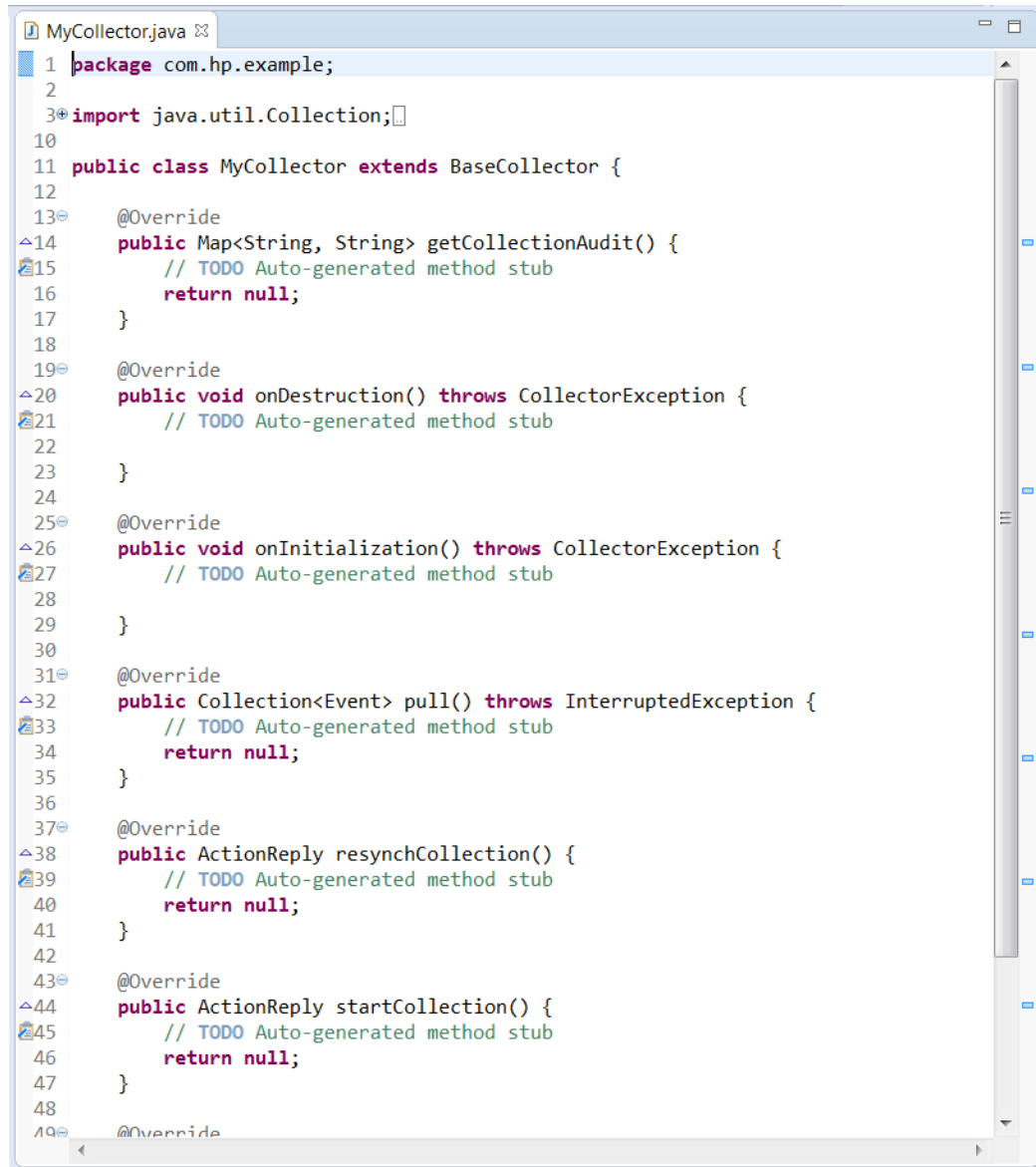


Figure 21 - Creating a "Collector" Java Class Step 2

Please make sure that the new “Collector” Java class that you’re creating extends the `com.hp.umb.adapter.collector.BaseCollector`. Automatically eclipse will create the methods from the `com.hp.umb.adapter.collector.CollectorInterface` Java interface.

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



```
1 package com.hp.example;
2
3 import java.util.Collection;
4
5
6
7
8
9
10
11 public class MyCollector extends BaseCollector {
12
13     @Override
14     public Map<String, String> getCollectionAudit() {
15         // TODO Auto-generated method stub
16         return null;
17     }
18
19     @Override
20     public void onDestruction() throws CollectorException {
21         // TODO Auto-generated method stub
22     }
23
24
25     @Override
26     public void onInitialization() throws CollectorException {
27         // TODO Auto-generated method stub
28     }
29
30
31     @Override
32     public Collection<Event> pull() throws InterruptedException {
33         // TODO Auto-generated method stub
34         return null;
35     }
36
37     @Override
38     public ActionReply resynchCollection() {
39         // TODO Auto-generated method stub
40         return null;
41     }
42
43     @Override
44     public ActionReply startCollection() {
45         // TODO Auto-generated method stub
46         return null;
47     }
48
49     @Override
```

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 Unified Mediation Bus 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 Unified Mediation Bus 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 Unified Mediation Bus 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 Unified Mediation Bus 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 re-synchronization facility. This is the case for TeMIP for example, that stores alarms and then is capable of re-sending all alarms at a given point in time, but will be true for any event provider that stores its event and is able to send it back again on request.
A resynchronized flow of events must be framed with two additional events: one indicating the beginning of the synchronization flow, and another one indicating the end of the resynchronization flow.
Such frame events must not vehicle any other information than the start and end of synchronization flags.
The Begin of Synchronization event is set with the attribute `beginOfSynchronization=true` and the End of synchronization event is set with the `endOfSynchronization=true`. Any event type extending the `com.hp.uca.expert.event.DefaultEvent` will inherit the two methods: `setBeginOfSynchronization()` and `setEndOfSynchronization()` that allow setting such attributes.
Any Event Type extending the `DefaultEvent` Class can therefore be used as Begin and End of synchronization events by setting these attributes.
In the case of an Alarm flow, the resynchronized Event flow starts with the specific `com.hp.uca.expert.alarm.internal.BeginSynchronization` event and is terminated by sending the `com.hp.uca.expert.alarm.internal.EndSynchronization` event.
 - **getCollectionAudit ()** : This method returns a description of the Collection flow. It is mainly used for troubleshooting purpose. It is called by the Unified Mediation Bus framework when a request 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 Unified Mediation Bus 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. 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.

You can find example 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:

- 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. Otherwise (if the producer collection flow is Dynamic), the Adapter will wait for a `CreateFlow` collection action request 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, 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 this state using the `setStatus()` method mainly to notify collection errors.

For such purpose there are two states available:

- **The FAILED state:**

This state is use to notify a fatal collection error. In such case the Collection Flow is considered as 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 with the ACTIVE state. In case of failure the state as to be turned to FAILED.

Note: Any attempts 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:

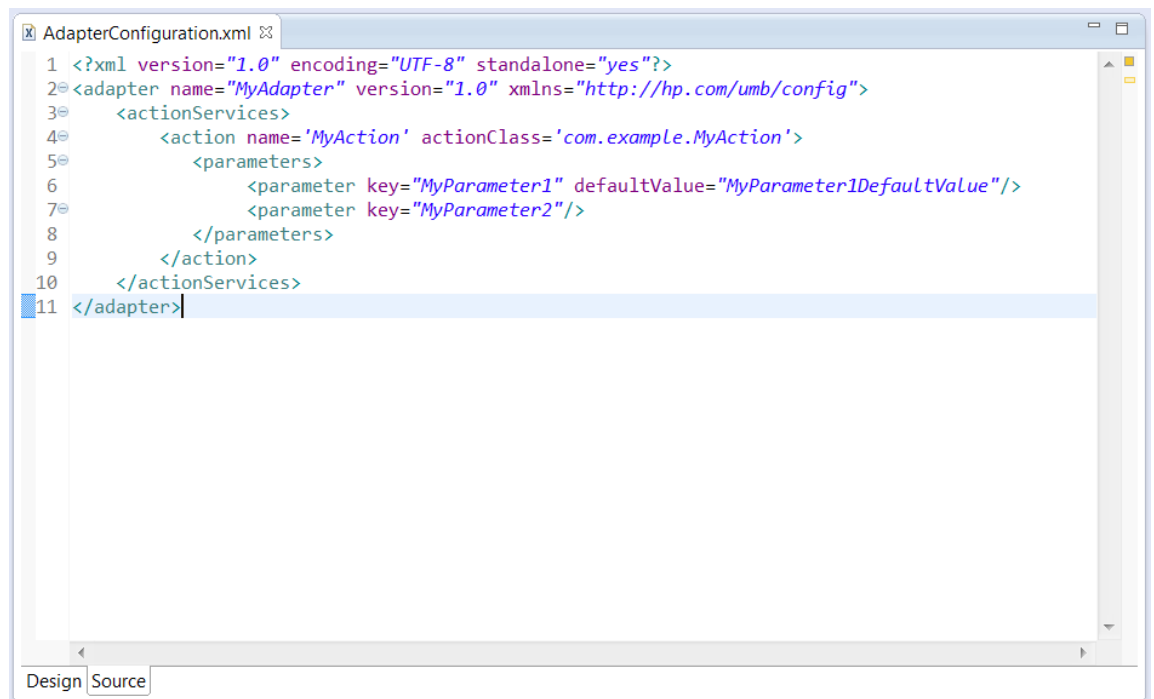
- **name:** this is the name of the action used to identify the action (for executing actions for example)
- **actionClass:** this is the name of the Java class (including the Java package name) implementing the action. For example: `com.example.MyAction`. This class has to extend the `com.hp.umb.adapter.BaseAction` abstract Java class.

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. Each parameter must define all of the following mandatory attributes:

- **key:** this is the name of the action's parameter. Only the name of the parameter defined here. The value of the parameter will be send when the action is executed. If no value is sent when the action is executed, the default value will be used.

Please refer to chapter 1.5.1 “Action Services” for more information on action services.

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



```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <adapter name="MyAdapter" version="1.0" xmlns="http://hp.com/umb/config">
3   <actionServices>
4     <action name='MyAction' actionClass='com.example.MyAction'>
5       <parameters>
6         <parameter key="MyParameter1" defaultValue="MyParameter1DefaultValue"/>
7         <parameter key="MyParameter2"/>
8       </parameters>
9     </action>
10  </actionServices>
11 </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:

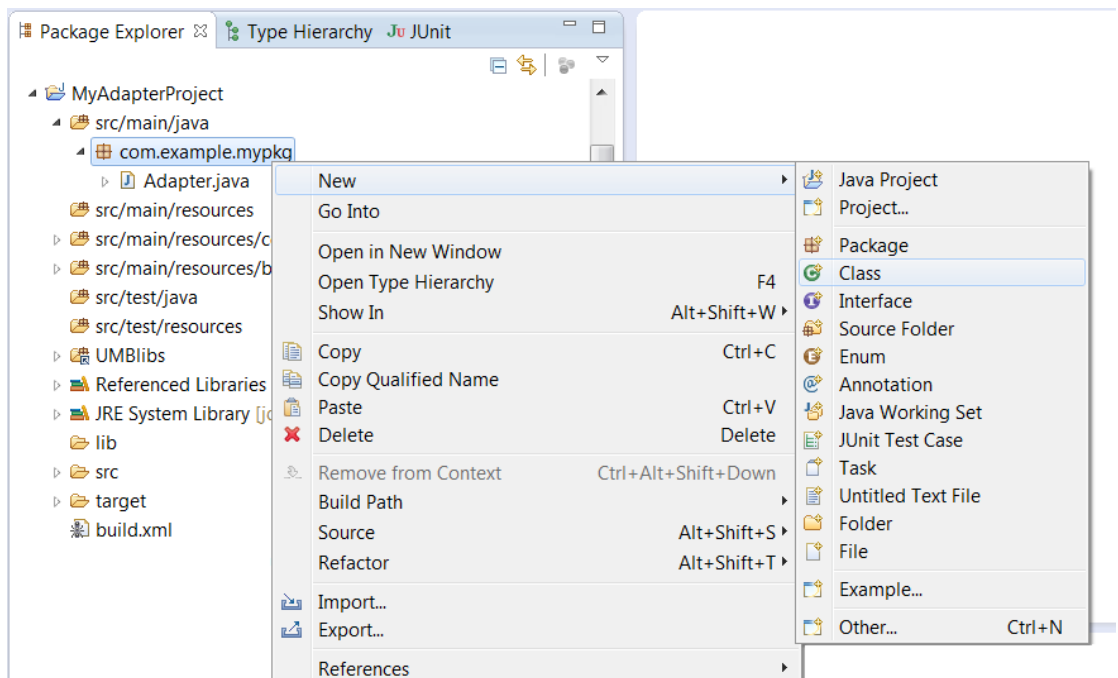


Figure 24 - Creating an “Action” Java class – Step 1

This opens the New Class window.

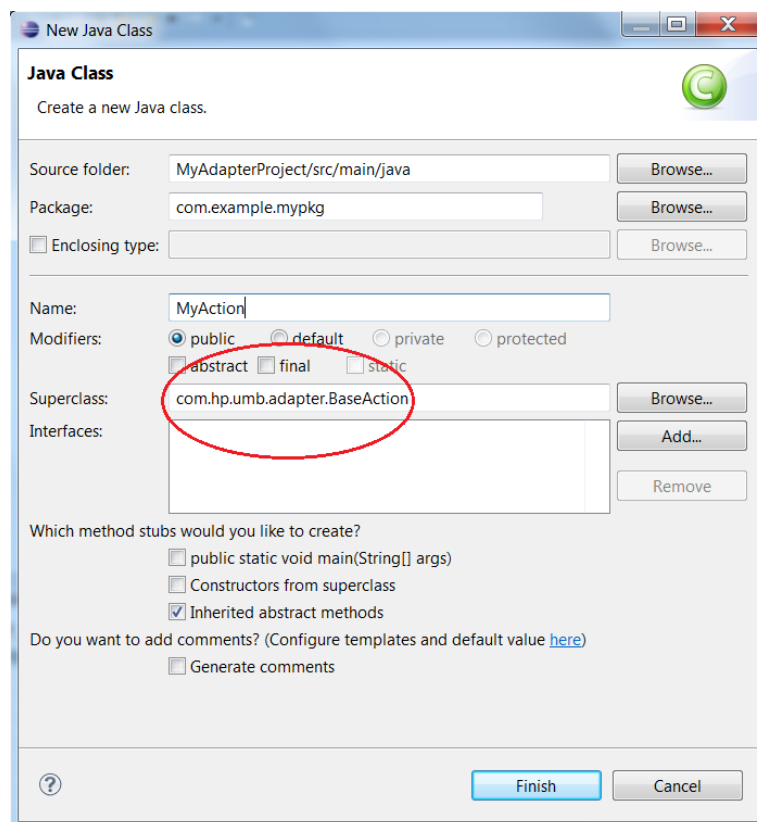
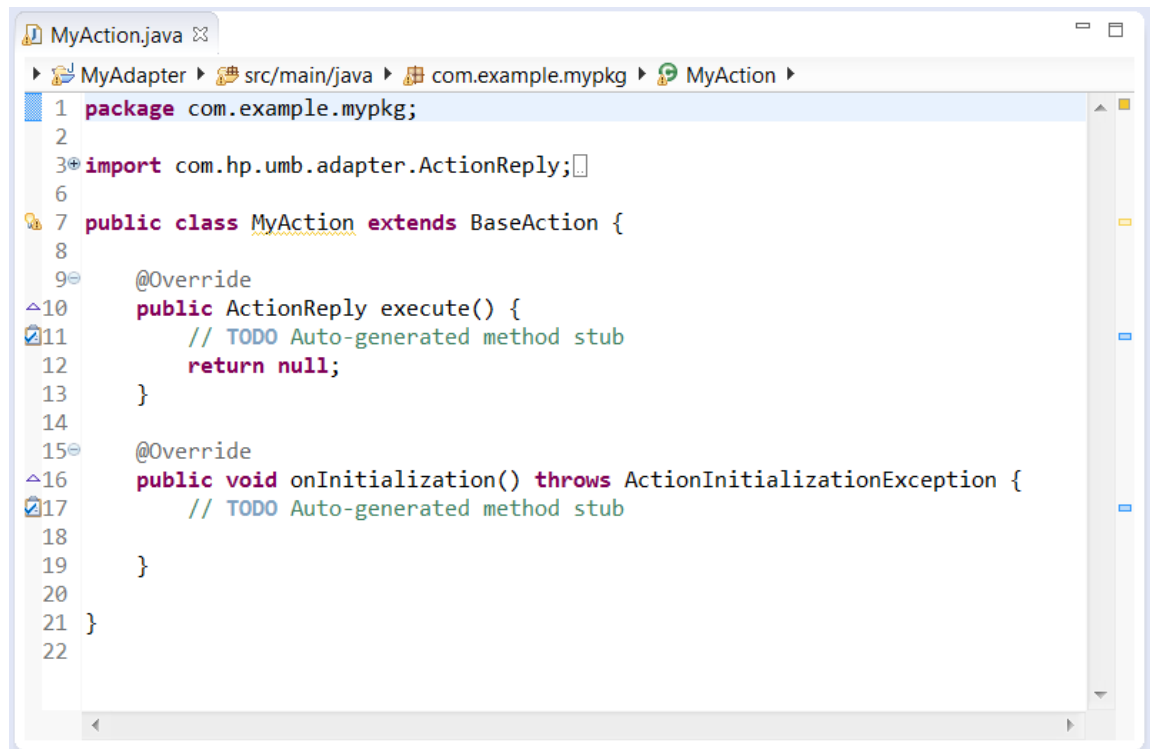


Figure 25 - Creating an “Action” Java class – Step 2

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:



```
1 package com.example.mypkg;
2
3 import com.hp.umb.adapter.ActionReply;
4
5
6
7 public class MyAction extends BaseAction {
8
9     @Override
10    public ActionReply execute() {
11        // TODO Auto-generated method stub
12        return null;
13    }
14
15    @Override
16    public void onInitialization() throws ActionInitializationException {
17        // TODO Auto-generated method stub
18    }
19 }
20
21
22
```

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.

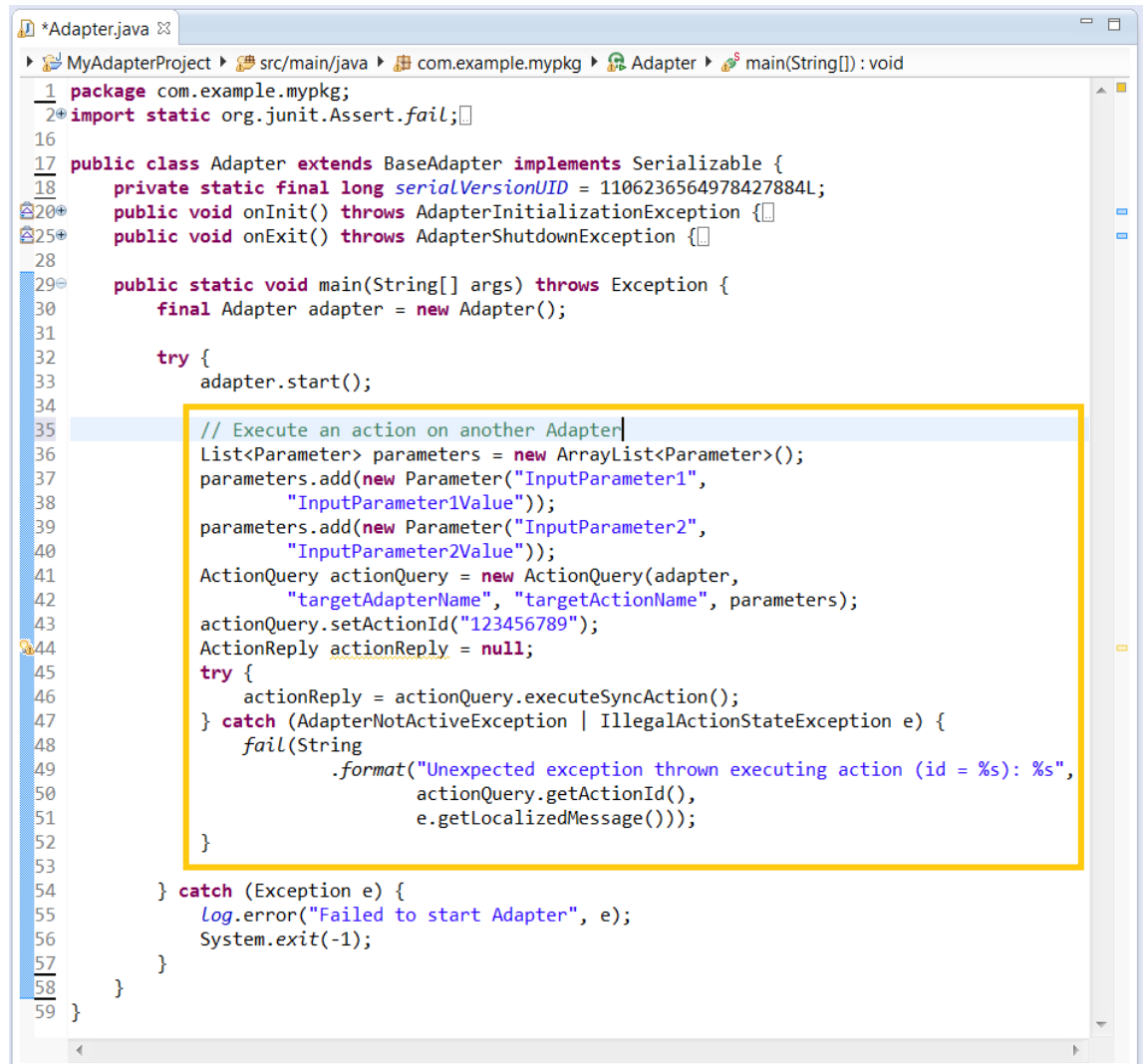
You can find example of implementation of “Action” Java classes in the Camel Adapter (`CamelAction.java` class) described in this document:

- 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:



```
1 package com.example.mypkg;
2 import static org.junit.Assert.fail;
16
17 public class Adapter extends BaseAdapter implements Serializable {
18     private static final long serialVersionUID = 1106236564978427884L;
20     public void onInit() throws AdapterInitializationException {}
25     public void onExit() throws AdapterShutdownException {}
28
29     public static void main(String[] args) throws Exception {
30         final Adapter adapter = new Adapter();
31
32         try {
33             adapter.start();
34
35             // Execute an action on another Adapter
36             List<Parameter> parameters = new ArrayList<Parameter>();
37             parameters.add(new Parameter("InputParameter1",
38                 "InputParameter1Value"));
39             parameters.add(new Parameter("InputParameter2",
40                 "InputParameter2Value"));
41             ActionQuery actionQuery = new ActionQuery(adapter,
42                 "targetAdapterName", "targetActionName", parameters);
43             actionQuery.setActionId("123456789");
44             ActionReply actionReply = null;
45             try {
46                 actionReply = actionQuery.executeSyncAction();
47             } catch (AdapterNotActiveException | IllegalActionStateException e) {
48                 fail(String
49                     .format("Unexpected exception thrown executing action (id = %s): %s",
50                         actionQuery.getActionId(),
51                         e.getLocalizedMessage()));
52             }
53
54         } catch (Exception e) {
55             Log.error("Failed to start Adapter", e);
56             System.exit(-1);
57         }
58     }
59 }
```

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):
 - o `public ActionReply executeSyncAction() throws IllegalActionStateException, AdapterNotFoundException, AdapterNotActiveException`

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

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 consumer flows that start automatically (when the Adapter starts) in the AdapterConfiguration.xml file
- Or define consumer flows and start them in the Java code of your Adapter

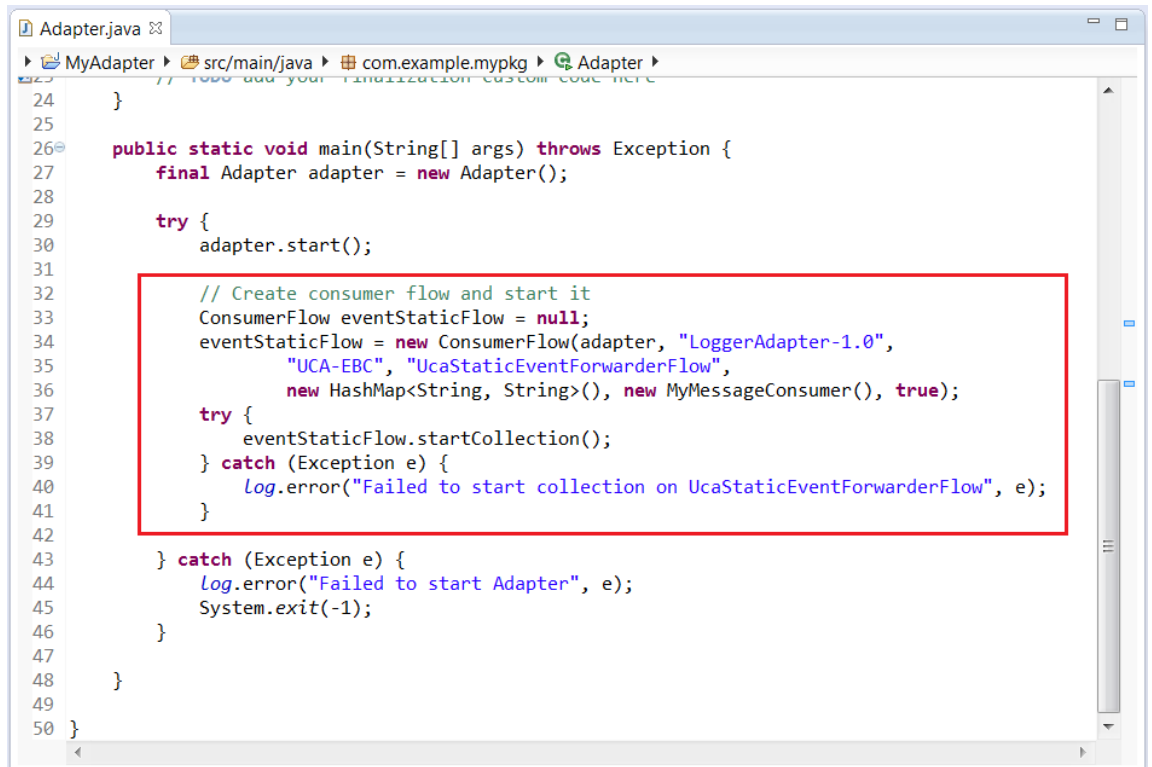
3.2.4.1 Automatically started Consumer flows

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

Refer to section 1.5.3 “Defining Flow Service Consumers from Configuration” for full description on how to define automatic consumers.

3.2.4.2 Consumer flows started by custom code

Alternatively to defining consumer flows that start automatically, it is also possible to define consumer flows and start them directly in the Java code of your Adapter:



```
Adapter.java
MyAdapter ▸ src/main/java ▸ com.example.mypkg ▸ Adapter ▸
24 }
25
26 public static void main(String[] args) throws Exception {
27     final Adapter adapter = new Adapter();
28
29     try {
30         adapter.start();
31
32         // Create consumer flow and start it
33         ConsumerFlow eventStaticFlow = null;
34         eventStaticFlow = new ConsumerFlow(adapter, "LoggerAdapter-1.0",
35             "UCA-EBC", "UcaStaticEventForwarderFlow",
36             new HashMap<String, String>(), new MyMessageConsumer(), true);
37
38         try {
39             eventStaticFlow.startCollection();
40         } catch (Exception e) {
41             log.error("Failed to start collection on UcaStaticEventForwarderFlow", e);
42         }
43     } catch (Exception e) {
44         log.error("Failed to start Adapter", e);
45         System.exit(-1);
46     }
47
48 }
49
50 }
```

Figure 28 - Java code to create consumer flows

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

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

The Consumer flows created from Adapter's custom code allows additionally to resynchronize the flow (in case the Flow Producer side allows this capability). In such case the re-synchronization can be requested by calling the `resyncCollection()`

3.2.4.3 Defining the flow message consumer class

Both the "auto" and java code created consumer flows 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.ConsumerMessageHandlerInterface<K extends Event>`

With this interface the consumed Events are returned one by one by the UMB framework. The framework is blocked on a read operation on the kafka topic. As soon

as an Event is received the `onNewMessage ()` method is called with this Event message as parameter.

- `com.hp.umb.adapter.consumer.ConsumerMessageSetHandlerInterface< K extends Event>`

With this interface the messages are not returned one by one but by Sets.

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

Example:

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

which sets the maximum set size to 100.

The UMB framework reads the Kafka topic until no message arrives for a given timeout. This timeout is a configuration 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 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 as elapsed.

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

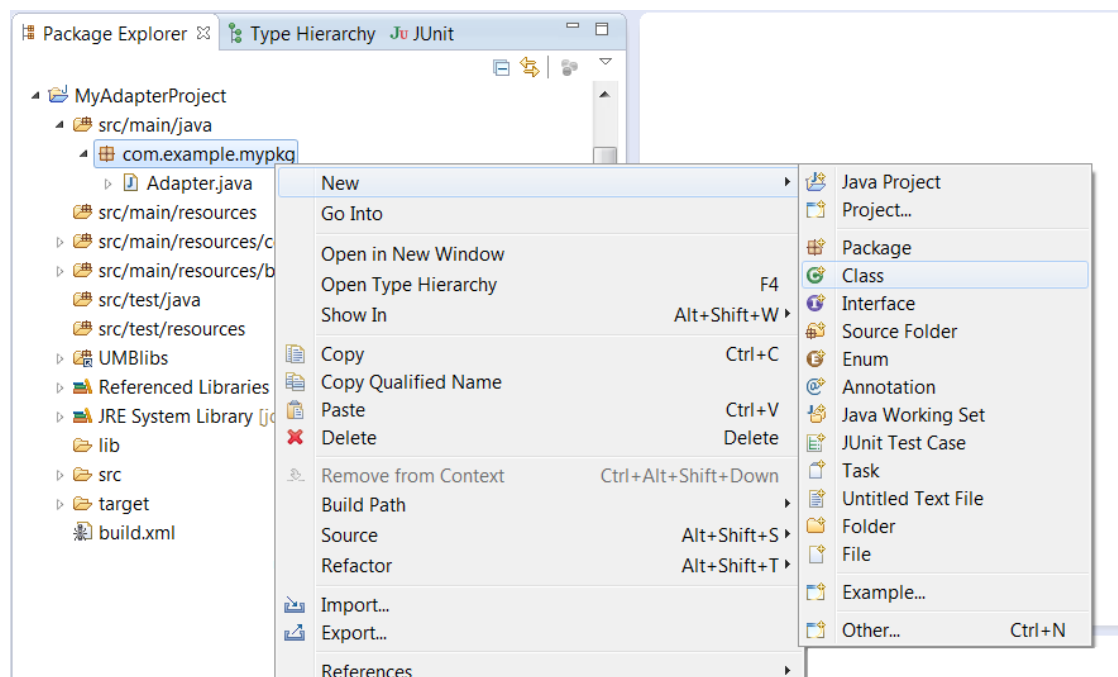


Figure 29 - Creating a “MessageConsumer” Java class – Step 1

This opens the New Class window.

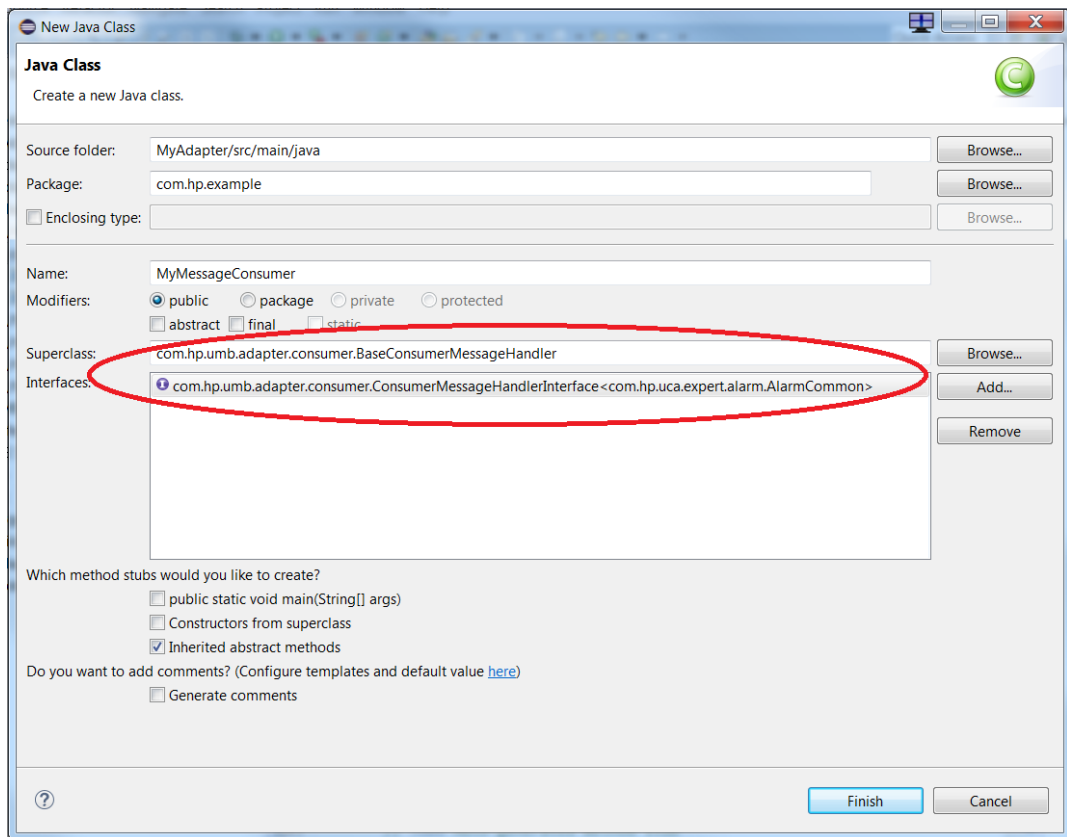
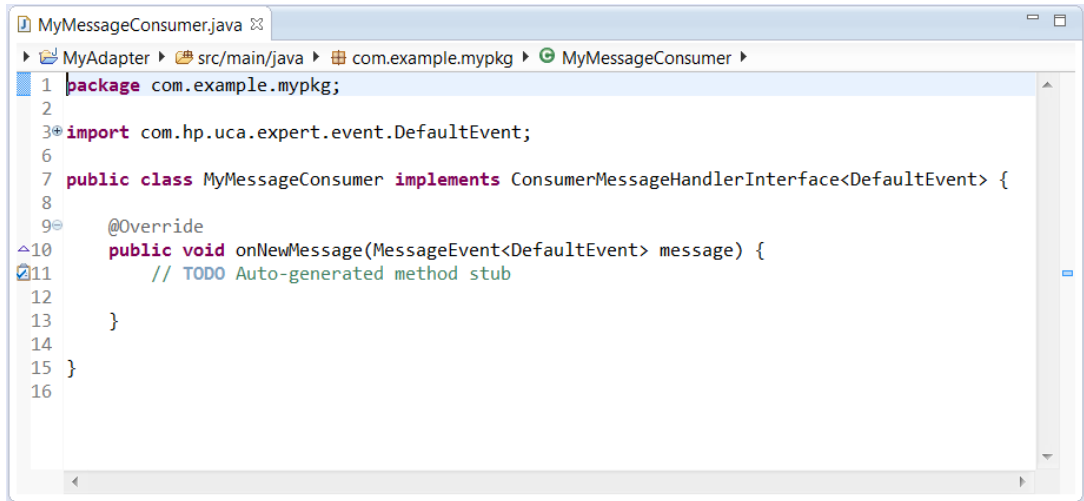


Figure 30 - Creating a “MessageConsumer” Java class – Step 2

Please make sure that the new “MyMessageConsumer” Java class that you’re creating extends the `com.hp.umb.adapter.consumer.BaseConsumerMessageHandler` class and implements one of `com.hp.umb.adapter.consumer.ConsumerMessageHandlerInterface<K extends Event>` or `com.hp.umb.adapter.consumer.ConsumerMessageSetHandlerInterface<K extends Event>` Java interface.

Note also that you have to specify the Interface’s formal type K (in the example : `com.hp.uca.alarm.AlarmCommon`)

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



```
1 package com.example.mypkg;
2
3 import com.hp.uca.expert.event.DefaultEvent;
4
5
6
7 public class MyMessageConsumer implements ConsumerMessageHandlerInterface<DefaultEvent> {
8
9     @Override
10    public void onNewMessage(MessageEvent<DefaultEvent> message) {
11        // TODO Auto-generated method stub
12    }
13
14 }
15
16
```

Figure 31 - Creating a “MessageConsumer” Java class – Step 3

In order to finalize the new “MessageConsumer” Java class, only one method need to be implemented:

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

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

The Consumer Flow owns a Status that reflects the state of the collection. This status is returned by calling the Consumer Flow `getStatus()` method.

The Consumer flow status diagram is as Follow:

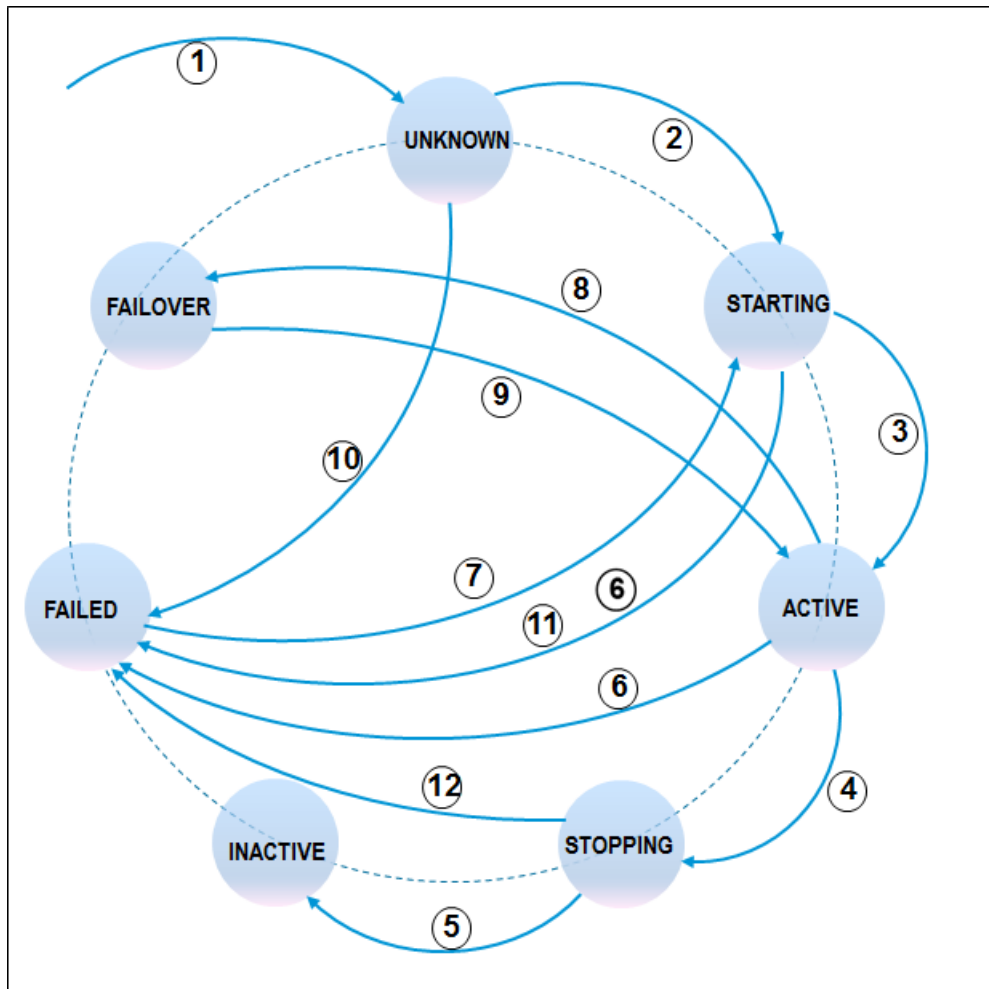


Figure 32 - Consumer Flow status diagram

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

1. ConsumerFlow Creation
2. *startCollection requested*
3. *startCollection completed*
4. *stopCollection requested*
5. Collection successfully stopped
6. Production flow Failed or Production Flow's Adapter stopped or Kafka server connection lost
7. ConsumerFlow Restarted by Monitoring
8. Producer collector recovering
9. Producer collector recovered
10. Adapter not started
11. Flow service does not exist

12. producer error while stopping flow

The Adapter can react to Consumer flow status changes by positioning a Flow status change listener. This is done by calling the adapter's method `addFlowStatusChangeListener()`. 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` class.

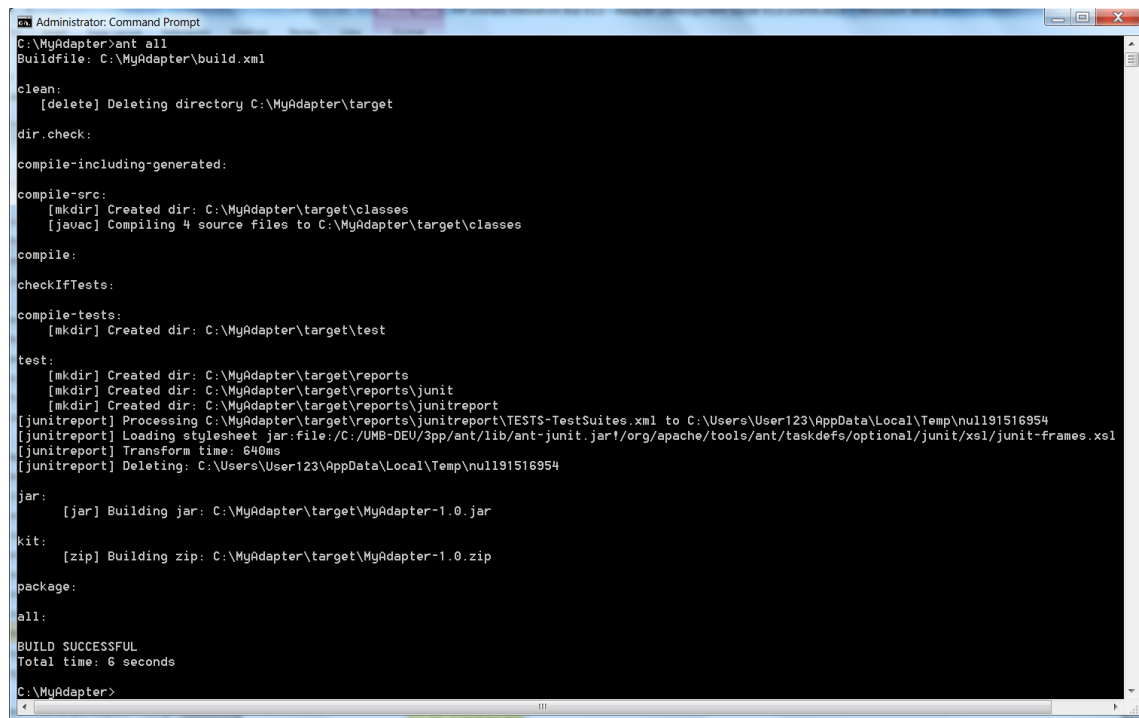
3.3 Generating the UMB Adapter kit

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

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

```
C:\> cd <Project Base>
C:\> ant all
```

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



```
Administrator: Command Prompt
C:\MyAdapter>ant all
BuildFile: C:\MyAdapter\build.xml

clean:
  [delete] Deleting directory C:\MyAdapter\target

dir.check:

compile-including-generated:

compile-src:
  [mkdir] Created dir: C:\MyAdapter\target\classes
  [javac] Compiling 4 source files to C:\MyAdapter\target\classes

compile:

checkIfTests:

compile-tests:
  [mkdir] Created dir: C:\MyAdapter\target\test

test:
  [mkdir] Created dir: C:\MyAdapter\target\reports
  [mkdir] Created dir: C:\MyAdapter\target\reports\junit
  [mkdir] Created dir: C:\MyAdapter\target\reports\junitreport
  [junitreport] Processing C:\MyAdapter\target\reports\junitreport\TESTS-TestSuites.xml to C:\Users\User123\AppData\Local\Temp\nu1191516954
  [junitreport] Loading stylesheet jar:file:/C:/UMB-DEU/3pp/ant/lib/ant-junit.jar!/org/apache/tools/ant/taskdefs/optional/junit/xsl/junit-frames.xsl
  [junitreport] Transform time: 640ms
  [junitreport] Deleting: C:\Users\User123\AppData\Local\Temp\nu1191516954

jar:
  [jar] Building jar: C:\MyAdapter\target\MyAdapter-1.0.jar

kit:
  [zip] Building zip: C:\MyAdapter\target\MyAdapter-1.0.zip

package:

all:

BUILD SUCCESSFUL
Total time: 6 seconds
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 `MyAdapter-1.0.zip`):

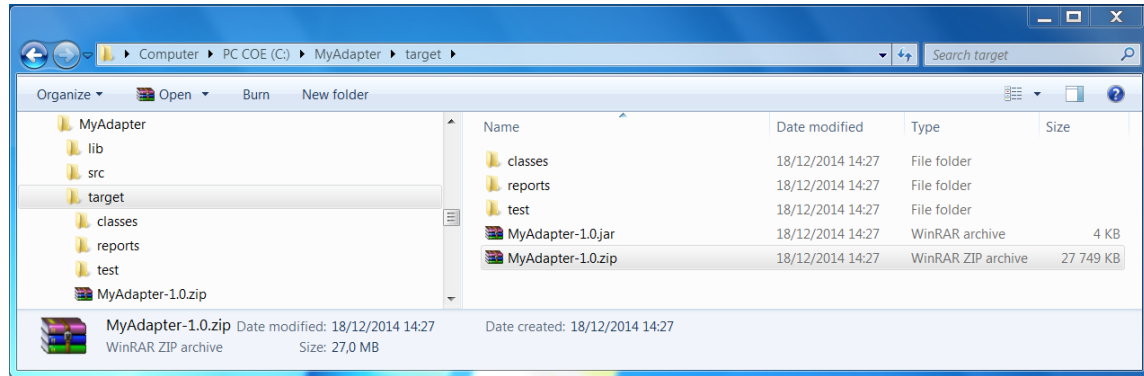


Figure 34 – Location of the kit of your Adapter

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

- **bin/ sub-folder that contains the Adapter start scripts:**
 - **adapter-start:** the Adapter’s Linux start shell script
 - **adapter-start.bat:** the Adapter’s Windows start batch script
- **conf/ sub-folder which contains the Adapter’s configuration files:**
 - **adapter.properties:** defines 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
- **lib/ sub-folder that contains the library files (jar files) necessary to run the Adapter**

Below is the full list of the contents of the ZIP file of your Adapter:

```
C:\MyAdapter\target>7z t MyAdapter-1.0.zip
7-Zip [64] 9.20 Copyright (c) 1999-2010 Igor Pavlov 2010-11-18
Processing archive: MyAdapter-1.0.zip

Testing      MyAdapter
Testing      MyAdapter\lib
Testing      MyAdapter\lib\MyAdapter-1.0.jar
Testing      MyAdapter\lib\annotations-1.3.2.jar
Testing      MyAdapter\lib\aopalliance-1.0.jar
Testing      MyAdapter\lib\camel-core-2.14.0.jar
Testing      MyAdapter\lib\camel-spring-2.14.0.jar
Testing      MyAdapter\lib\commons-configuration-1.10.jar
Testing      MyAdapter\lib\commons-io-1.4.jar
Testing      MyAdapter\lib\commons-lang-2.4.jar
Testing      MyAdapter\lib\commons-logging-1.1.1.jar
Testing      MyAdapter\lib\hamcrest-core-1.3.jar
Testing      MyAdapter\lib\hazelcast-3.2.3.jar
Testing      MyAdapter\lib\jaxb-api-2.2.7.jar
Testing      MyAdapter\lib\jaxb-impl-2.2.6.jar
Testing      MyAdapter\lib\jaxb-xjc-2.2.6.jar
```

```

Testing      MyAdapter\lib\jline-0.9.94.jar
Testing      MyAdapter\lib\jopt-simple-3.2.jar
Testing      MyAdapter\lib\junit-4.11.jar
Testing      MyAdapter\lib\kafka 2.10-0.8.1.1.jar
Testing      MyAdapter\lib\log4j-1.2.17.jar
Testing      MyAdapter\lib\metrics-core-2.2.0.jar
Testing      MyAdapter\lib\opencsv-2.3.jar
Testing      MyAdapter\lib\scala-library-2.10.1.jar
Testing      MyAdapter\lib\slf4j-api-1.7.5.jar
Testing      MyAdapter\lib\slf4j-log4j12-1.7.5.jar
Testing      MyAdapter\lib\snappy-java-1.0.5.jar
Testing      MyAdapter\lib\spring-aop-3.2.11.RELEASE.jar
Testing      MyAdapter\lib\spring-beans-3.2.11.RELEASE.jar
Testing      MyAdapter\lib\spring-context-3.2.11.RELEASE.jar
Testing      MyAdapter\lib\spring-core-3.2.11.RELEASE.jar
Testing      MyAdapter\lib\spring-expression-3.2.11.RELEASE.jar
Testing      MyAdapter\lib\spring-tx-3.2.11.RELEASE.jar
Testing      MyAdapter\lib\uca-ant-assembly-3.2-NGMED-SNAPSHOT.zip
Testing      MyAdapter\lib\uca-common-3.2-NGMED-SNAPSHOT.jar
Testing      MyAdapter\lib\umb-adapter-fmk-1.0-SP1-SNAPSHOT.jar
Testing      MyAdapter\lib\umb-demo-classes-1.0-SP1-SNAPSHOT.jar
Testing      MyAdapter\lib\umbEclipsePluginSite-1.0.0-SNAPSHOT-
assembly.zip
Testing      MyAdapter\lib\zkclient-0.3.jar
Testing      MyAdapter\lib\zookeeper-3.3.4.jar
Testing      MyAdapter\conf
Testing      MyAdapter\conf\AdapterConfiguration.xml
Testing      MyAdapter\conf\adapter.properties
Testing      MyAdapter\conf\hazelcast.xml
Testing      MyAdapter\conf\log4j.xml
Testing      MyAdapter\bin
Testing      MyAdapter\bin\adapter-start
Testing      MyAdapter\bin\adapter-start.bat

Everything is Ok

Folders: 4
Files: 44
Size:      31523046
Compressed: 28414597

```

Figure 35 - Contents of the kit of your Adapter

3.4 Installing the UMB Adapter kit

Copy the kit of your Adapter (the ZIP file located at: `target/<Adapter name>-<Adapter version>.zip`) to any directory on the system where you want to install it and unzip it.

For example:

```

C:\> cd <Adapter Install Dir>
<Adapter Install Dir>> unzip <Adapter name>-<Adapter
version>.zip

```

3.5 Starting the UMB Adapter kit

Once the Adapter is installed, you can start it using either the Windows start script or the Linux start script:

On Windows:

```
<Adapter Install Dir>> adapter-start.bat
```

On Linux:

```
<Adapter Install Dir>$ adapter-start
```

Advance development topics

4.1 Extending the DefaultEvent class

By default, UMB adapters produce or consume events which are java instances of the class DefaultEvent.java.

However it is possible to create more specific event classes, provided they extend the class DefaultEvent.java

Suppose you want your adapter to handle temperature metrics.

This paragraph explains the steps to produce or consume temperature metrics

The general idea is to create a Temperature class, and to add this class to the classpath of the adapter(s) producing temperature metrics and to the classpath of the adapter(s) consuming the temperature metrics

One way is to define the Temperature java class in a separate java project producing a small jar file containing the Temperature.class

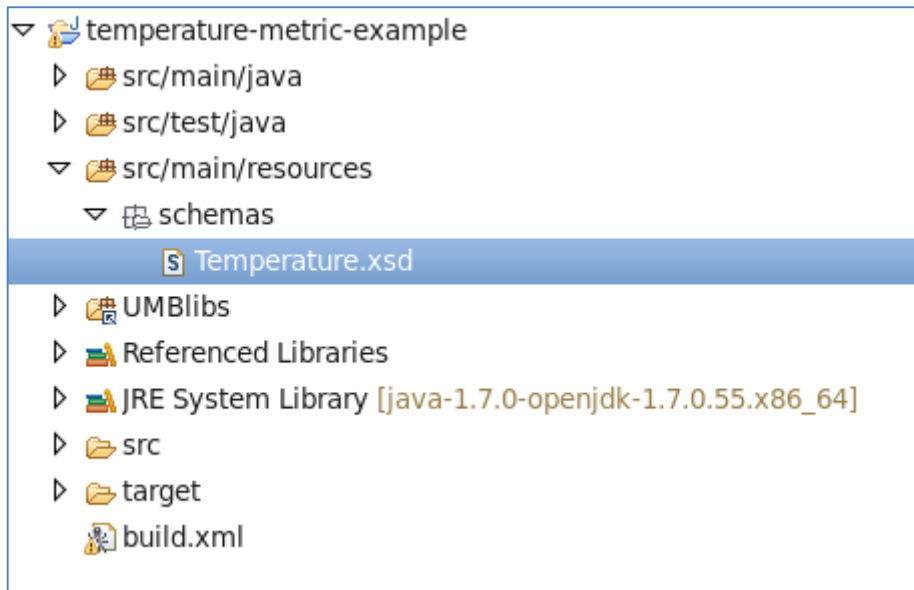
Alternatively you can define the Temperature.java directly inside your adapter (producer of temperature metrics) project. But the Temperature.class will still have to be packaged in a separated small .jar file to be used at the consumer side

The UMB Adapter Developer Kit `${UMB_DEV_HOME}/adapter-examples/` folder on Linux, `%UMB_DEV_HOME%\adapter-examples` folder on Windows

contains the **umb-demo-classes project**

This project exactly demonstrates the introduction of the temperature metric.
Here are the steps to follow:

4.1.1 Defining the new metric schema



Start by defining the Temperature.xsd XML schema.

This will be used by JAXB to automatically generate a TemperatureBase java class.

Temperature.xml events will have to follow the Temperature.xsd schema.

Below is the generic JAXB diagram for reference.

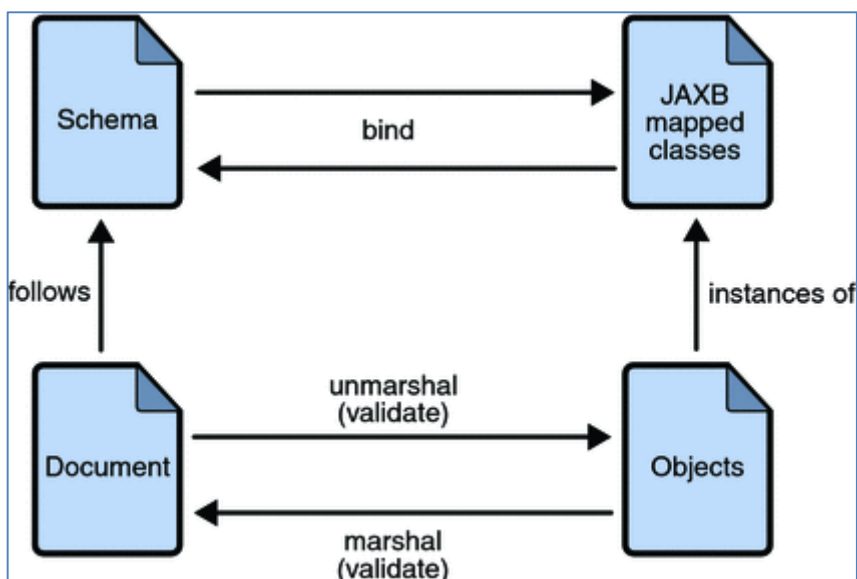


Figure 36 - JAXB Diagram

```
<?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"
```

```

jaxb:extensionBindingPrefixes="xjc"
jaxb:version="1.0"

<!-- FORCE ALL CLASSES IMPLEMENTS SERIALIZABLE -->
<xs:annotation>
  <xs:appinfo>
    <jaxb:globalBindings generateIsSetMethod="true">
      <xjc:serializable uid="123456" />
      <xjc:superClass name="com.hp.uca.expert.event.DefaultEvent" />
    </jaxb:globalBindings>
  </xs:appinfo>
</xs:annotation>

<!-- -->
<!-- ELEMENTS DEFINITION -->
<!-- -->
<xs:element name="temperatureBase">
  <xs:complexType>
    <xs:annotation>
      <xs:appinfo>
        <jaxb:class name="TemperatureBase"
implClass="com.hp.uca.expert.demo.Temperature" />
      </xs:appinfo>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="value" type="xs:double" minOccurs="1" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>

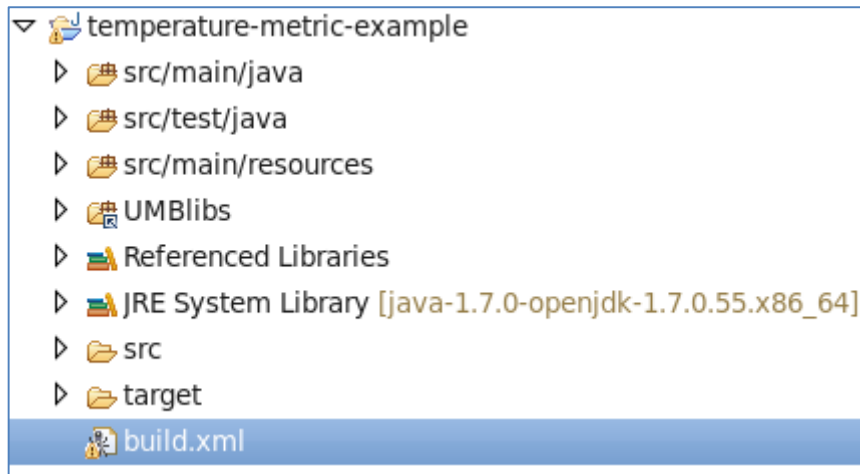
```

Temperature.xsd

The lines highlighted are essential. The globalBindings tag is used to specify that the generated TemperatureBase class will be extending DefaultEvent.java

4.1.2 Generating the new metric java class

The build.xml file contained in the umb-demo-classes contains the steps to automatically generate the TemperatureBase.java



```
<!-- UMB libraries location -->
<property name="umb.libs" value="${env.UMB_DEV_HOME}/lib" />

<!-- Environment directory tree -->
<property name="src.dir" value="src/main" />
<property name="src-resources.dir" value="src/main/resources" />
<property name="schemas.dir" value="src/main/resources/schemas" />
<property name="build.dir" value="target" />

<property name="generated-resources.dir" value="${build.dir}/generated-
sources/xjc" />

<taskdef name="xjc" classname="com.sun.tools.xjc.XJCTask">
  <classpath>
    <fileset dir="${umb.libs}" includes="*.jar" />
  </classpath>
</taskdef>

<target name="xjc-generate-sources">
  <mkdir dir="${generated-resources.dir}" />
  <xjc destdir="${generated-resources.dir}" extension="true"
removeOldOutput="true">
    <schema dir="${schemas.dir}" includes="**/*.xsd"/>
  </xjc>
</target>
```

Extract of the build.xml with highlights of important lines

```
// This file was generated by the JavaTM Architecture for XML Binding(JAXB)
Reference Implementation, v2.2.6

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.XmlElement;
import javax.xml.bind.annotation.XmlType;
```

```

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "value"
})
public class TemperatureBase
    extends DefaultEvent
    implements Serializable
{

```

Generated TemperatureBase.java

4.1.3 Adding marshaller and unmarshaller for the new metric

We need the temperature metrics objects to be marshallable when we want to store them in a file to replay them later.

We also need the temperature metrics stored in a file to be unmarshallable so as to load them from the file and use them in adapters and other java applications.

Since the TemperatureBase.java is a generated class, there is not much flexibility to add attributes and marshal() /unmarshal() methods to it.

That is why we create a Temperature.java class extending TemperatureBase.java,

This Temperature.java class initializes the marshaller and unmarshaller only once, the first time it is used. And it also implements the marshal() and unmarshal() methods.

See below the Temperature.java class

```

@XmlRootElement
public class Temperature extends TemperatureBase {

    private static final long serialVersionUID = -1380279792464305020L;
    private static JAXBContext jaxbContext;
    private static Marshaller jaxbMarshaller;
    private static Unmarshaller jaxbUnmarshaller;
    private static String jaxbMarshallerError;

    static {
        try {

            /*
             * initializing the marshaller and unmarshaller once and only once
             * since it was defined in a static block
             */
            jaxbContext = JAXBContext.newInstance(Temperature.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) {
        jaxbMarshallerError = e.getMessage();
    }
}

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

    if (jaxbMarshaller != null) {
        try {
            jaxbMarshaller.marshal((TemperatureBase) this, os);
            returnValue = os.toString("UTF-8");
        } catch (JAXBException e) {
            returnValue = e.getMessage();
        } catch (UnsupportedEncodingException e) {
            returnValue = e.getMessage();
        }
    } else {
        returnValue = jaxbMarshallerError;
    }
    return returnValue;
}

public static Temperature unmarshal(StringBuffer in) throws JAXBException {
    return (Temperature) jaxbUnmarshaller.unmarshal(new StreamSource(new
StringReader(in.toString())));
}

public static Temperature unmarshal(String in) {
    try {
        return (Temperature) jaxbUnmarshaller.unmarshal(new
StreamSource(new StringReader(in)));
    } catch (JAXBException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
        return null;
    }
}
}
}

```

Temperature.java

4.1.4 Generating the new metric jar file

Below is the extract of the build.xml used to generate the .jar file

```
<!-- Make the java code a jar file -->
```

```

<target name="jar" description="Make the ${project.artifactId}-
${project.version} project code jar file.">
  <jar jarfile="${build.dir}/${lib.name}.jar">
    <!-- ship the code -->
    <fileset dir="${classes.dir}" includes="**/*.class" />
  </jar>
</target>

```

Extract of [build.xml](#) showing the generation of the .jar file containing the Temperature.class

```
[hpossadm@tempeature-metric-classes]$ ant all
```

Run ant all command to produce the .jar file containing the Temperature.class and add it to the classpath of all your adapters and applications needing to handle temperature metrics

4.2 Customizing the serialization Class

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

A custom linearization class can be used instead of the default one.

Such class must implement the following interfaces:

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

and

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

Here is a skeleton of a custom serialization class:

```

import kafka.utils.VerifiableProperties;
import kafka.serializer.Decoder;
import kafka.serializer.Encoder;

import com.hp.umb.adapter.internal.utilities.JavaClassSerializer;

public class MySerializer implements Encoder<Object>, Decoder<Object> {

    public MySerializer(VerifiableProperties verifiableProperties) {
        // leave this empty
    }

    // linearization method
    public byte[] toBytes(Object message) {
        // put your linearization code here
        return total;
    }

    // de-linearization method
    public Object fromBytes(byte[] bytes) {

```

```

        // put your de-linearization code here
        return obj;
    }
}

```

The use of a custom serialization class is configured on both the producer and the consumer sides:

- On the Producer side:

In the flow service description (`AdapterConfiguration.xml` file) using the 'serializerClass' attribute.

Example:

```

<?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:

- By setting the serializer class on the consumerFlow object:

```

myFlow.setSerializerClass("com.hp.umb.adapter.collection.MySerializer")
;

```

- Or by setting the serializer class on the autoConsumer definition in the `AdapterConfiguration.xml` file:

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 Discovering the solution's adapter topology and states

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

This is done mainly through two functionalities:

1. Get the list of known remote adapters
2. Get notifications on adapter addition, adapter deletion, adapter state change.

4.3.1 Getting the list of known Adapters

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

This method can be used as follow:

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

4.3.2 Getting Adapter's Notifications

Adapter's notifications are emitted each any adapter state change:

- A new adapter just started.
- An adapter stopped.
- An adapter's state has changed (STARTING to ACTIVE), (ACTIVE to STOPPING).

Handling Adapters notifications starts by writing an AdapterProxy Listener 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());
}
}

```

The this AdapterProxy listener must be registered to the UMB framework using the `addAdapterProxyListener()` method of the `com.hp.umb.adapter.BaseAdapter` class:

```

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

```

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 Camel³ 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:
 - 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 provided by the adapter
 - 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`

³ Please see <http://camel.apache.org/> for more information on Camel

The Adapter uses the Camel Spring API⁴ instead of the Camel Java API⁵ 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.

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

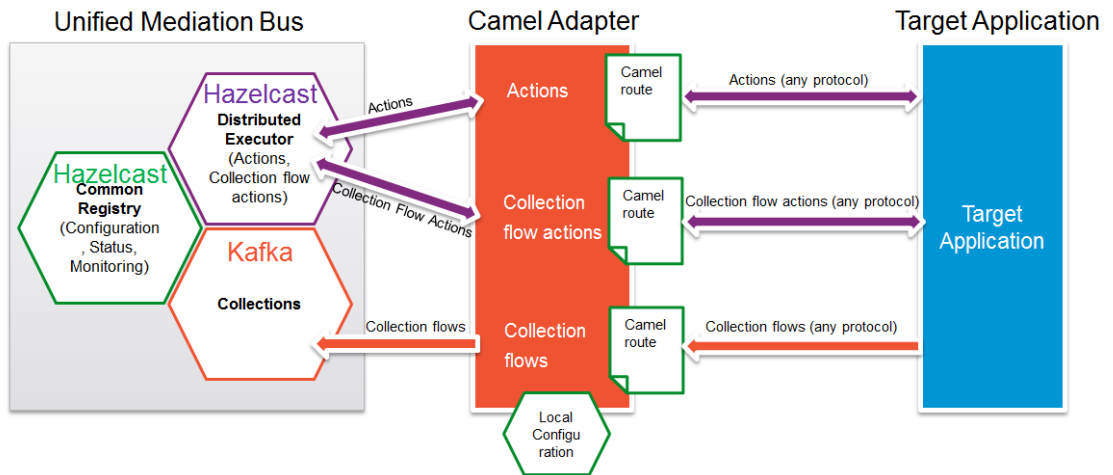


Figure 37 - Camel adapter overview

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

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

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 `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).

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.



```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <adapter name="Camel" version="1.0" xmlns="http://hp.com/umb/config">
3   <actionServices>
4     <action name="CamelAction" actionClass='com.hp.umb.adapter.camel.CamelAction'>
5       <parameters>
6         <parameter key="ActionRouteURI" defaultValue="direct:startAction"/>
7       </parameters>
8     </action>
9   </actionServices>
10  <flowServices>
11    <flow name="CamelDynamicFlow" type="Dynamic" collectorClass="com.hp.umb.adapter.camel.CamelCollector">
12      <parameters>
13        <parameter key="CollectionActionRouteURI" defaultValue="direct:startCollectionAction"/>
14        <parameter key="CollectionRouteURI" defaultValue="direct:endCollection"/>
15      </parameters>
16    </flow>
17  </flowServices>
18 </adapter>
```

Figure 38 - 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.

Figure 39 - 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 routes⁷.

The Camel Adapter defines 3 routes:

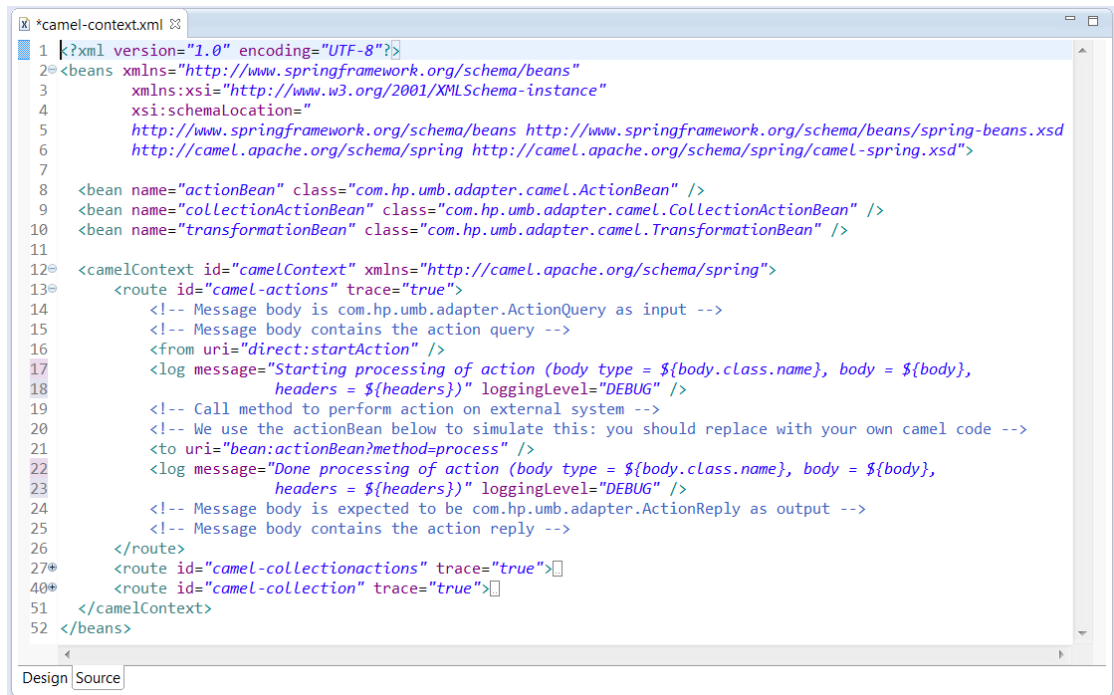
- The “camel-actions” route: this route processes action requests for actions named “CamelAction”
- The “camel-collectionactions” route: this route processes collection flow action requests, i.e. `CreateFlow/DeleteFlow/ResynchFlow/StatusFlow` for the flow named “CamelDynamicFlow”
- The “camel-collection”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

The “camel-actions” route

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

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.



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="
5         http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6         http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">
7
8   <bean name="actionBean" class="com.hp.umb.adapter.camel.ActionBean" />
9   <bean name="collectionActionBean" class="com.hp.umb.adapter.camel.CollectionActionBean" />
10  <bean name="transformationBean" class="com.hp.umb.adapter.camel.TransformationBean" />
11
12  <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
13    <route id="camel-actions" trace="true">
14      <!-- Message body is com.hp.umb.adapter.ActionQuery as input -->
15      <!-- Message body contains the action query -->
16      <from uri="direct:startAction" />
17      <log message="Starting processing of action (body type = ${body.class.name}, body = ${body},
18            headers = ${headers})" loggingLevel="DEBUG" />
19      <!-- Call method to perform action on external system -->
20      <!-- We use the actionBean below to simulate this: you should replace with your own camel code -->
21      <to uri="bean:actionBean?method=process" />
22      <log message="Done processing of action (body type = ${body.class.name}, body = ${body},
23            headers = ${headers})" loggingLevel="DEBUG" />
24      <!-- Message body is expected to be com.hp.umb.adapter.ActionReply as output -->
25      <!-- Message body contains the action reply -->
26    </route>
27*   <route id="camel-collectionactions" trace="true">
40*   <route id="camel-collection" trace="true">
51 </camelContext>
52 </beans>
```

Figure 40 - “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.

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.

```
*camel-context.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="
5         http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6         http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">
7
8   <bean name="actionBean" class="com.hp.umb.adapter.camel.ActionBean" />
9   <bean name="collectionActionBean" class="com.hp.umb.adapter.camel.CollectionActionBean" />
10  <bean name="transformationBean" class="com.hp.umb.adapter.camel.TransformationBean" />
11
12 <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
13   <route id="camel-actions" trace="true">
14     <route id="camel-collectionactions" trace="true">
15       <!-- Message body is java.util.Map<String, String> as input -->
16       <!-- Message body contains the parameters to the collection flow action -->
17       <!-- including "ActionHint" key which contains the type of collection flow action
18       (CreateFlow, DeleteFlow, ResynchFlow, or StatusFlow) -->
19       <!-- and "FlowName" which contains the name of the flow -->
20       <from uri="direct:startCollectionAction" />
21       <log message="Starting processing of collection flow action (body type = ${body.class.name},
22       body = ${body}, headers = ${headers})" loggingLevel="DEBUG" />
23       <!-- Call method to perform collection flow action on external system -->
24       <!-- We use the collectionActionBean below to simulate this: you should replace with your
25       own Camel code -->
26       <to uri="bean:collectionActionBean?method=process" />
27       <!-- Message body is expected to be com.hp.umb.adapter.ActionReply as output -->
28       <!-- Message body contains the collection flow action reply -->
29     </route>
30   </route>
31 </camelContext>
32 </beans>
```

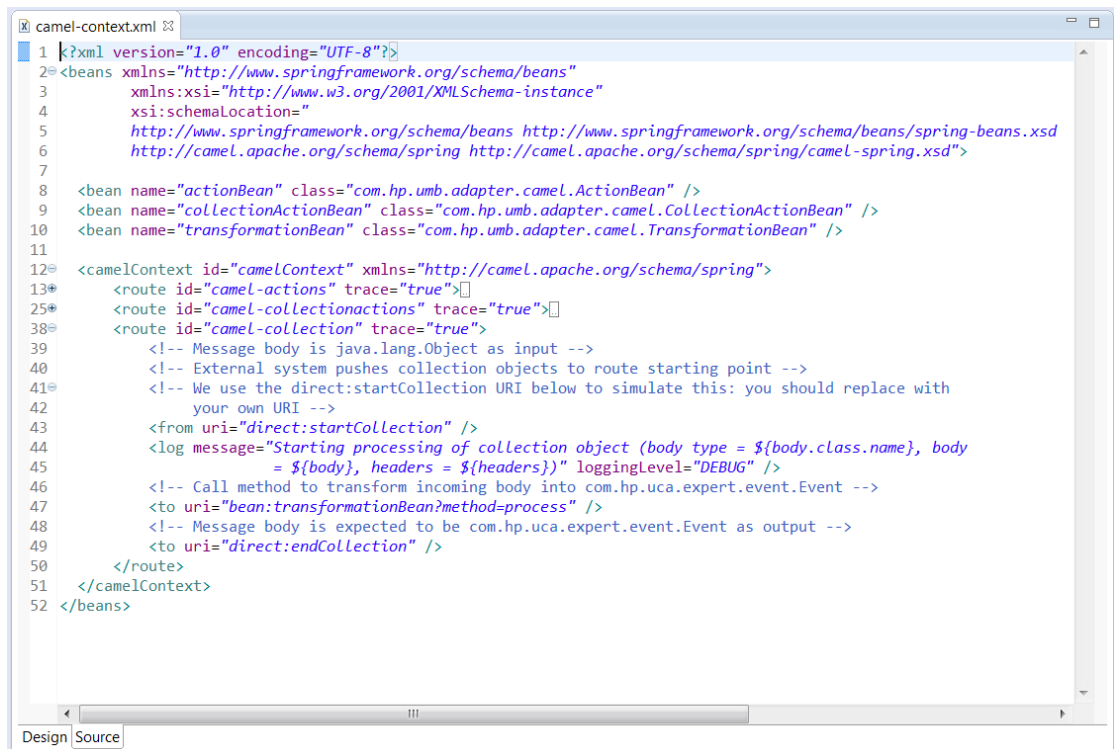
Figure 41 - “camel-collectionactions” route in the camel-context.xml file

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.

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.



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="
5         http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
6         http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">
7
8   <bean name="actionBean" class="com.hp.umb.adapter.camel.ActionBean" />
9   <bean name="collectionActionBean" class="com.hp.umb.adapter.camel.CollectionActionBean" />
10  <bean name="transformationBean" class="com.hp.umb.adapter.camel.TransformationBean" />
11
12  <camelContext id="camelContext" xmlns="http://camel.apache.org/schema/spring">
13    <route id="camel-actions" trace="true">
14      <from uri="direct:startCollection" />
15      <log message="Starting processing of collection object (body type = ${body.class.name}, body
16        = ${body}, headers = ${headers})" loggingLevel="DEBUG" />
17      <!-- Call method to transform incoming body into com.hp.uca.expert.event.Event -->
18      <to uri="bean:transformationBean?method=process" />
19      <!-- Message body is expected to be com.hp.uca.expert.event.Event as output -->
20      <to uri="direct:endCollection" />
21    </route>
22  </camelContext>
23 </beans>
```

Figure 42 - “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.

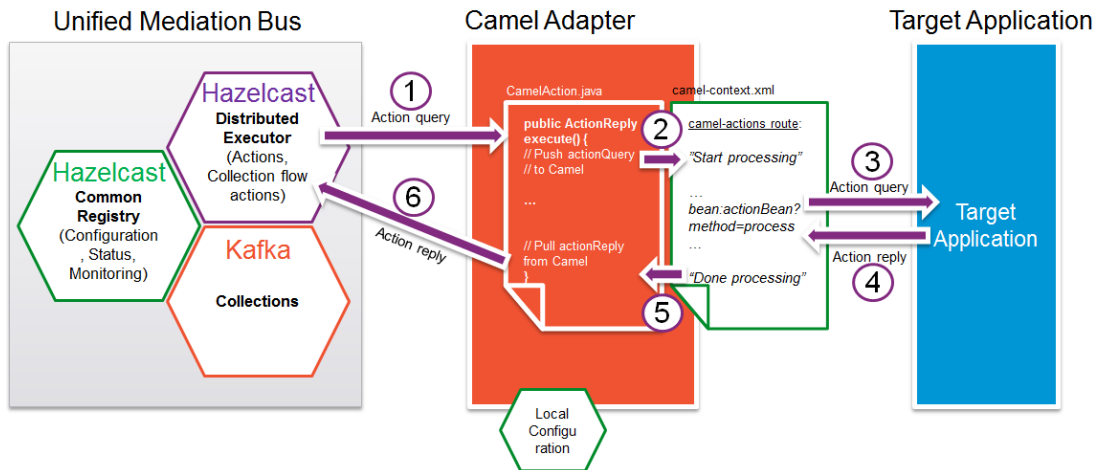


Figure 43 - Processing Actions in the Camel Adapter

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 processing⁸. 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 `com.hp.umb.adapter.ActionQuery` as input of the route and `com.hp.umb.adapter.ActionReply` as output of the route.

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

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.

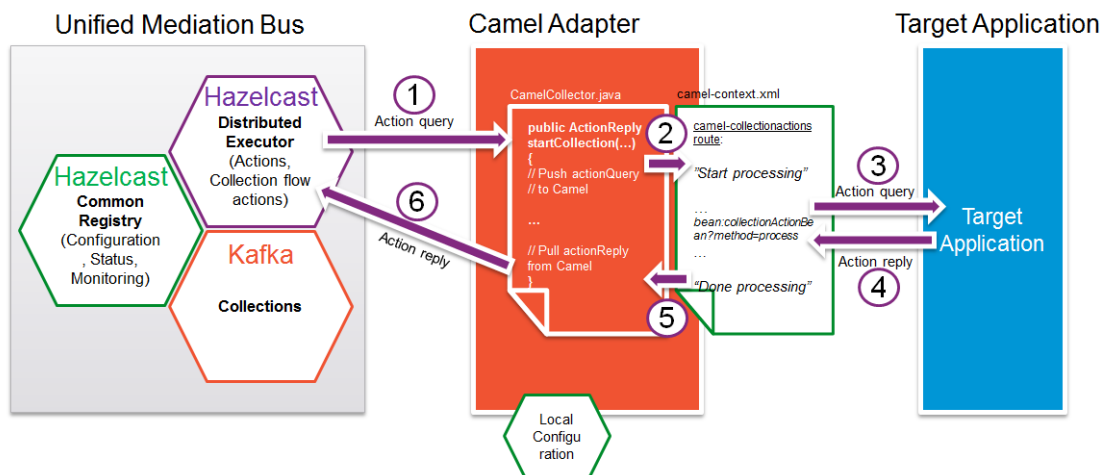


Figure 44 - Processing Collection Flow Actions in the Camel Adapter

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):

```
o public ActionReply startCollection()
o public ActionReply stopCollection()
o public ActionReply resynchCollection()
```



```
o 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)

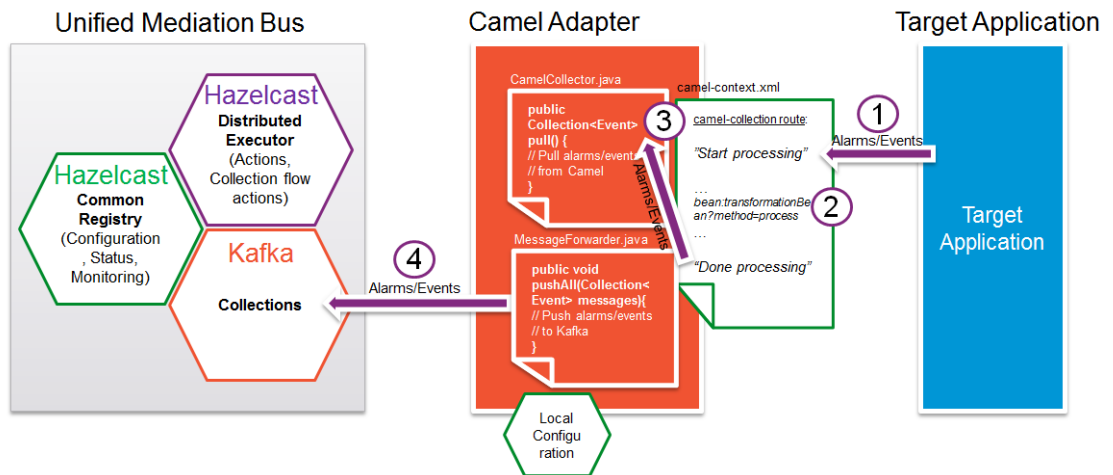


Figure 45 - Processing Collections in the Camel Adapter

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.

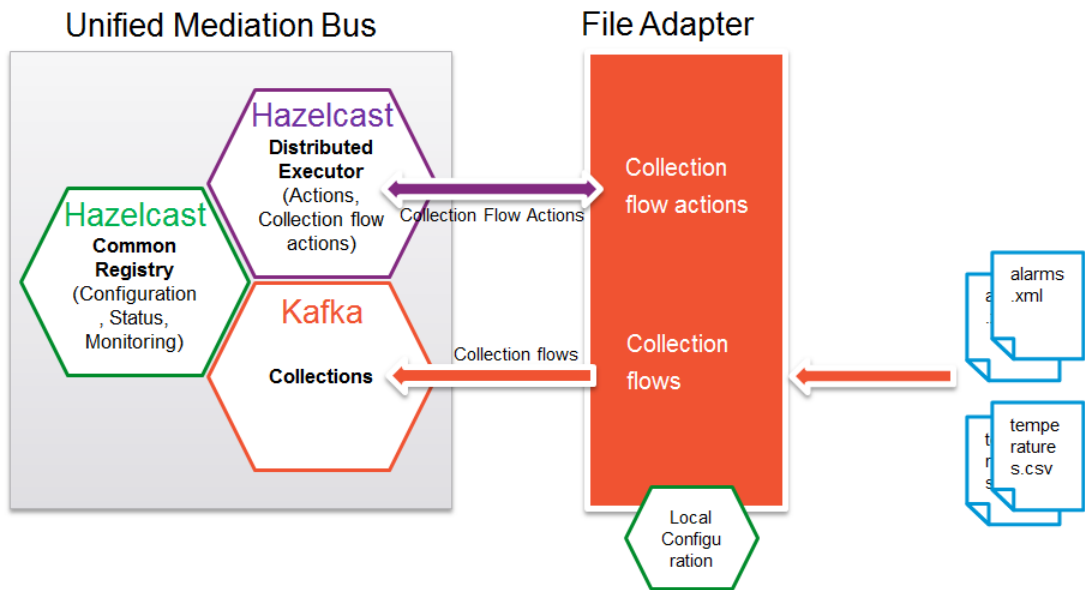


Figure 46 - File adapter overview

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.metadata.broker.list:** a list of Kafka broker `<host>:<port>` information
- **producer.request.required.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.

5.2.1.2 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.



```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <adapter name="FileAdapter" version="1.0" xmlns="http://hp.com/umb/config">
3   <flowServices>
4     <flow name="AlarmFileStaticFlow" type="Static" collectorClass="com.hp.umb.adapter.file.FileCollector">
5       <parameters>
6         <parameter key="fileName" defaultValue="data/alarms.xml"/>
7       </parameters>
8     </flow>
9     <flow name="AlarmFileDynamicFlow" type="Dynamic" collectorClass="com.hp.umb.adapter.file.FileCollector">
10      <parameters>
11        <parameter key="fileName" defaultValue="data/alarms.xml"/>
12      </parameters>
13    </flow>
14    <flow name="TemperaturesStaticFlow" type="Static" collectorClass="com.hp.umb.adapter.file.TemperaturesCollector">
15      <parameters>
16        <parameter key="fileName" defaultValue="data/temperatures.csv"/>
17      </parameters>
18    </flow>
19    <flow name="TemperaturesDynamicFlow" type="Dynamic" collectorClass="com.hp.umb.adapter.file.TemperaturesCollector">
20      <parameters>
21        <parameter key="fileName" defaultValue="data/temperatures.csv"/>
22      </parameters>
23    </flow>
24  </flowServices>
25 </adapter>
```

Figure 47 - The File Adapter's AdapterConfiguration.xml file

By default, 4 flows are defined:

- 2 alarm flows: one static and one dynamic⁹
- 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.

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.

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.

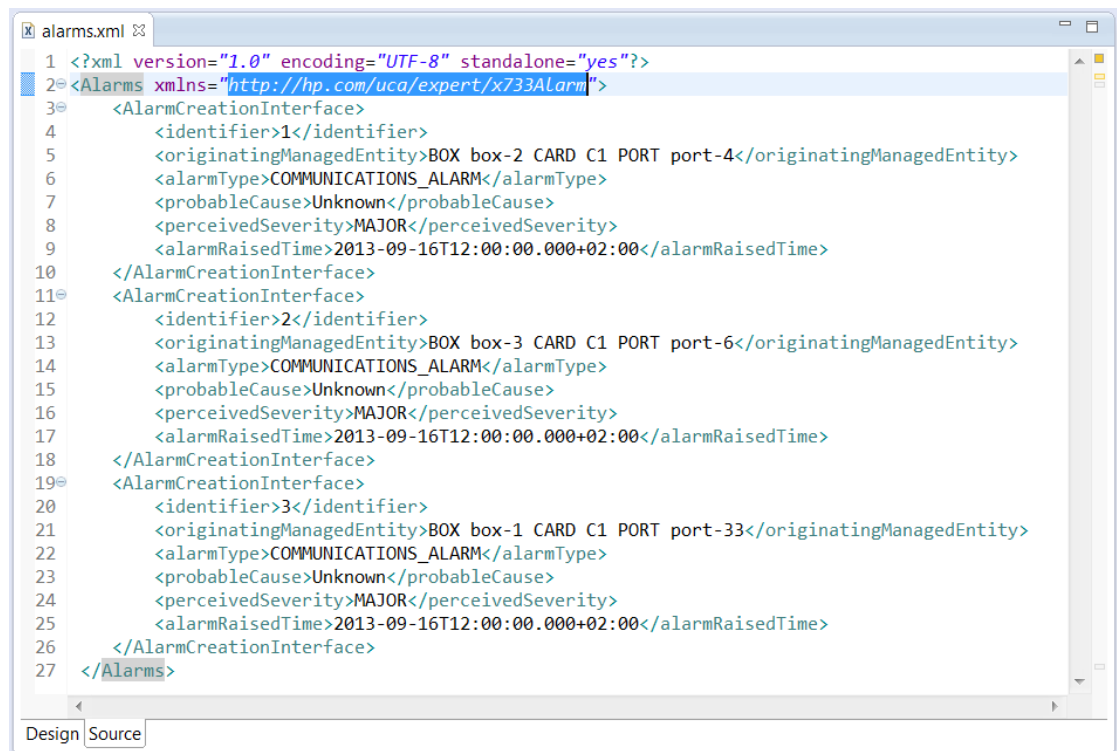
The flow named “TemperaturesDynamicFlow” is identical to the “TemperaturesStaticFlow” except that it is declared to be dynamic.

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

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>).



```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <Alarms xmlns="http://hp.com/uca/expert/x733Alarm">
3   <AlarmCreationInterface>
4     <identifier>1</identifier>
5     <originatingManagedEntity>BOX box-2 CARD C1 PORT port-4</originatingManagedEntity>
6     <alarmType>COMMUNICATIONS_ALARM</alarmType>
7     <probableCause>Unknown</probableCause>
8     <perceivedSeverity>MAJOR</perceivedSeverity>
9     <alarmRaisedTime>2013-09-16T12:00:00.000+02:00</alarmRaisedTime>
10  </AlarmCreationInterface>
11  <AlarmCreationInterface>
12    <identifier>2</identifier>
13    <originatingManagedEntity>BOX box-3 CARD C1 PORT port-6</originatingManagedEntity>
14    <alarmType>COMMUNICATIONS_ALARM</alarmType>
15    <probableCause>Unknown</probableCause>
16    <perceivedSeverity>MAJOR</perceivedSeverity>
17    <alarmRaisedTime>2013-09-16T12:00:00.000+02:00</alarmRaisedTime>
18  </AlarmCreationInterface>
19  <AlarmCreationInterface>
20    <identifier>3</identifier>
21    <originatingManagedEntity>BOX box-1 CARD C1 PORT port-33</originatingManagedEntity>
22    <alarmType>COMMUNICATIONS_ALARM</alarmType>
23    <probableCause>Unknown</probableCause>
24    <perceivedSeverity>MAJOR</perceivedSeverity>
25    <alarmRaisedTime>2013-09-16T12:00:00.000+02:00</alarmRaisedTime>
26  </AlarmCreationInterface>
27 </Alarms>
```

Figure 48 - File Adapter’s “alarms.xml” data file

The “temperatures.csv” data file

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

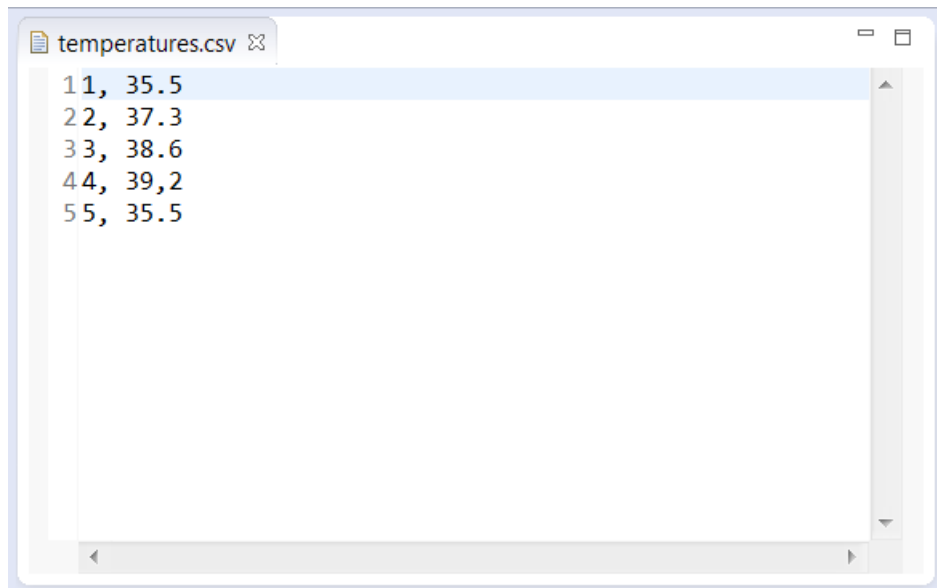


Figure 49 - File Adapter's "temperatures.csv" data file

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

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.

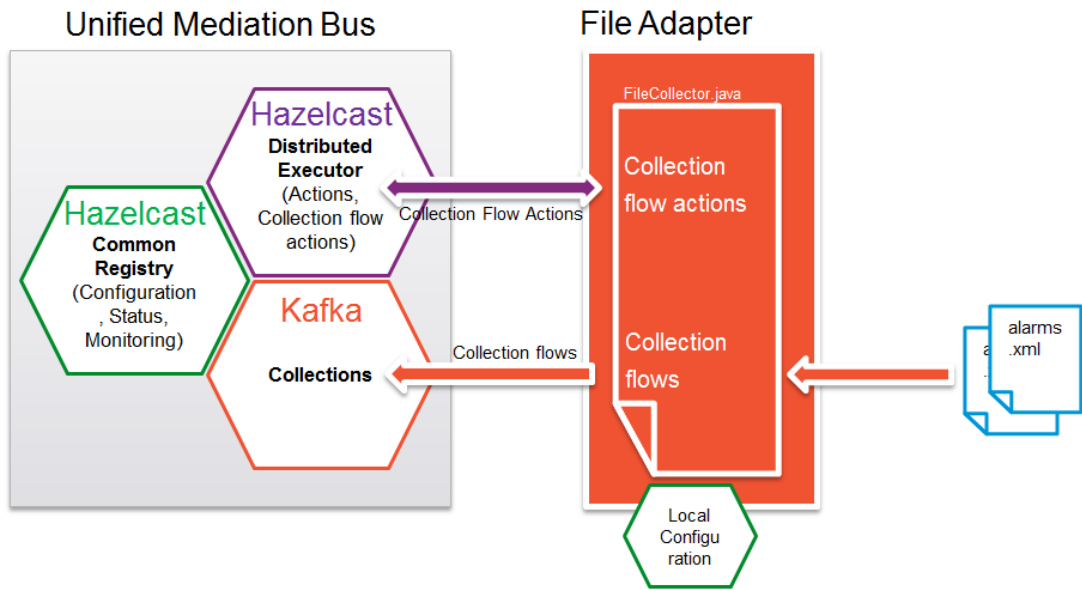


Figure 50 - File Adapter's alarms collections

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.

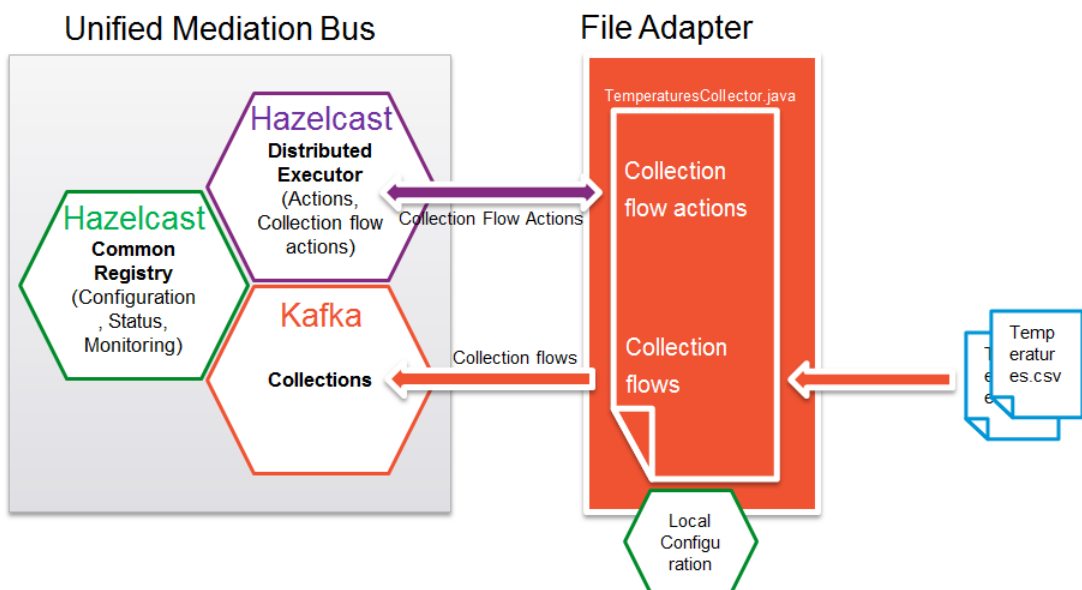


Figure 51 - File Adapter's temperatures collections

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 alarms¹⁰:
`com.hp.umb.adapter.log.LogAlarmConsumer`
- One that can consume flows of events¹¹:
`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

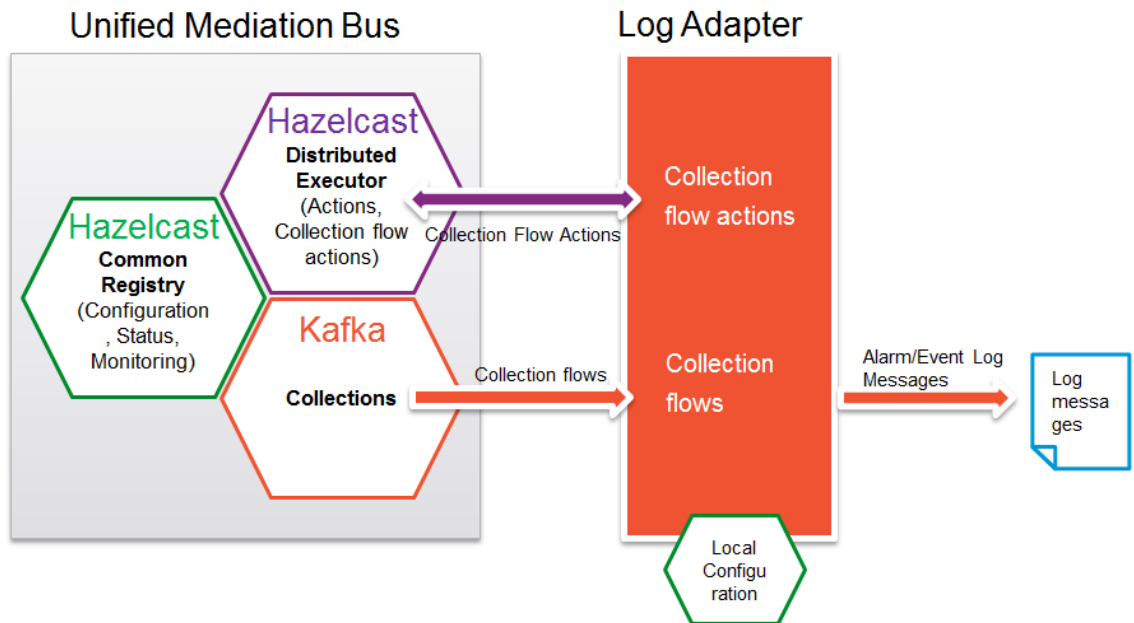


Figure 52 - Log adapter overview

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:

- **consumer.zookeeper.connect:** a list of ZooKeeper `<host>:<port>` information
- **consumer.zookeeper.session.timeout.ms:** set to 6000 by default
- **consumer.zookeeper.sync.time.ms:** set to 203 by default
- **consumer.auto.commit.interval.ms:** set to 1000 by default
- **consumer.auto.offset.reset:** set to `smallest` by default

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.



```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <adapter name="LogAdapter" version="1.0" xmlns="http://hp.com/umb/config">
3   <autoConsumers>
4     <autoConsumer consumerIdentifier="EventLogger" targetAdapterName="FileAdapter"
5       targetFlowName="TemperaturesDynamicFlow"
6       messageConsumerClass="com.hp.umb.adapter.Log.LogEventConsumer"/>
7     <autoConsumer consumerIdentifier="AlarmLogger" targetAdapterName="FileAdapter"
8       targetFlowName="AlarmFileStaticFlow"
9       messageConsumerClass="com.hp.umb.adapter.Log.LogAlarmConsumer"/>
10  </autoConsumers>
11 </adapter>
```

Figure 53 - 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

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 handler¹² class that defines what to do with each message consumed from the collection flow. Messages take the form of alarms¹³ in the case of the `UcaStaticForwarderFlow` flow and events¹⁴ 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

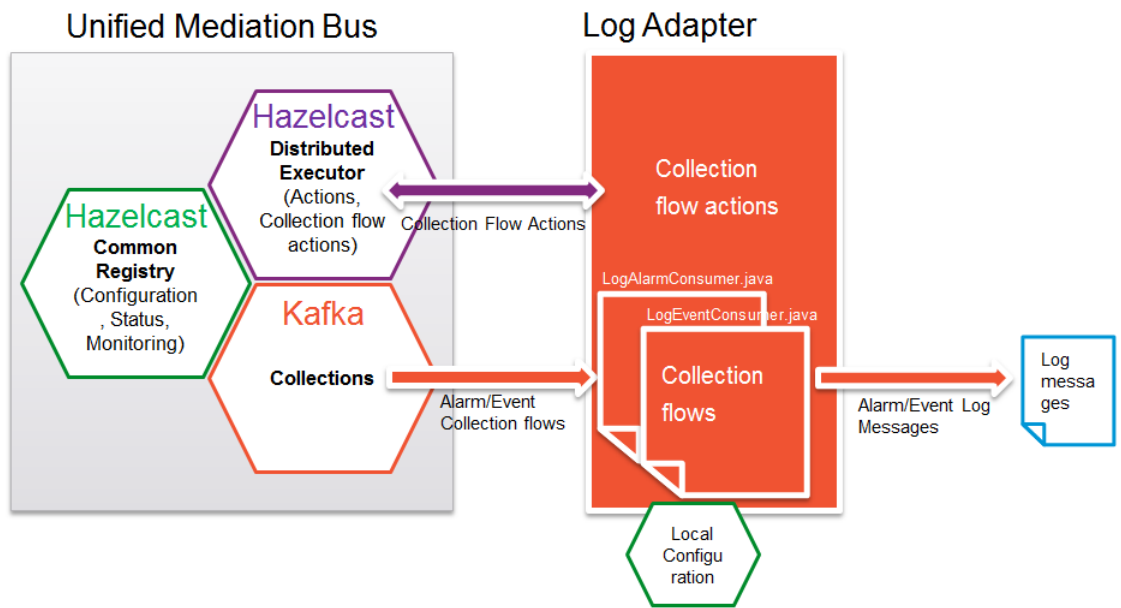


Figure 54 - Log Adapter consuming alarms/events collections

A. Ant *build.xml* targets

The value pack examples provided with UMB come with an Ant *build.xml* file that can build and package the project as described in this document.

Following is the full list of Apache Ant targets defined in the *build.xml* file that can be executed from the command line using the **ant** tool:

eclipse

Command:

```
# ant eclipse
```

Creates the *.project* and *.classpath* files used by eclipse when importing a project.

clean

Command:

```
# ant clean
```

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

compile

Command:

```
# ant compile
```

Compiles all Java files of the project.

test

Command:

```
# ant test
```

Runs the JUnit tests defined in the project.

package

Command:

```
# ant package
```

Build the final, “ready to deploy” value pack ZIP file.

all

Command:

```
# ant all
```

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

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