UltraStudio

USER GUIDE | 17.07

AdroitLogic

UltraStudio User Guide

User Guide Revision

Revision 01 - Covers up to 17.07.0 release

Bug Reports

If you’d like to report any bugs, typos, or suggestions just email us at: info@adroitlogic.com.

Authors

Sajith Edirisinghe, Udith Gunaratna, Sudheera Palihakkara

Project X Concepts

PROJECT X CONCEPTS

What is Project X

Simply put, Project X is the next generation of UltraESB. It consists of all the API definitions, core implementations of those APIs, messaging engine, message format definitions and implementations, metrics engine etc. These core functionalities and capabilities provided by project x can be enriched by using it in combination with transports, connectors and processing elements. These sub components makes the usage of UltraESB more user intuitive and user friendly than the previous versions.

Message Formats

Message formats are the various kinds of message payloads supported and handled by project-x. As of now, the following types of Message Formats are supported.

★ ByteArray Format

This format can be used to keep the payload of the message as an array of bytes. The number of bytes in the array is considered as the size of the payload.

★ **Empty Format**

This format can be used to create a message without an actual payload. This will be useful in situations where the payload of the message is not yet available at the time of the message creation.

★ **File Format**

This format can be used to have a regular operating system file as the payload of the message. The length of the file in bytes is considered as the size of this type of payload.

★ **Map Format**

This message format keeps a Java Map as the payload. The size of the map i.e. the number of key-value mappings is considered as the size of this type of payload. Along with the common API methods implementations, this message format provides methods to access the properties of the underlying map.

★ **Message File Format**

This format is also similar to the FileFormat but it contains the payload in a XMessageFile i.e. an entry in UltraESB File Store rather than in a regular operating system file.

★ **Object Format**

This format can be used to have any Java Object as the payload of the message. Since the actual behavior or the properties if the underlying object cannot be guaranteed, this format does not support API methods such as reading payload as a stream or retrieving size of the payload.

★ **StringFormat**

This format can be used to have a Java String as the payload. The length of the String object is considered as the size of the payload.

Message Context

Message Context is a wrapper object that includes an actual message object inside. Each message context has a unique identifier and contains all the contextual properties related to the processing of the message such as transactional information, resource handlers, scope information, context properties and also provides means to manipulate contextual data such as scope variables.

In a simple integration flow which does not clone or split the message, the same message context is being used throughout the flow. Even in a request-response scenario, the same message context will be used in both request path and response path, though the request message and response message are different.

Transports

Transports implement the low level communication protocols supported by the ESB to communicate with other ESB instances or outside systems. Currently the ESB supports the following transport protocols and many more.

★ HTTP (Netty/NIO)

★ JMS

★ File Polling

★ File NIO

★ SFTP

★ FIX

★ AMQP

★ SCP

★ Timer

Almost all of these transport implementations consist of a listener implementation as well as a sender implementation. Transport listeners are used to receive messages to the ESB and transport senders are used to send messages out of the ESB. While one or more ingress connectors can depend on a single transport listener, one or more egress connectors can depend on a single transport sender.

Connectors

Connectors represent the high level communication methodologies supported by the ESB. There are mainly two types of connectors as Ingress Connectors and Egress Connectors.

An ingress connector uses a specific transport listener implementation to receive messages, while an egress connector uses a specific transport sender to send messages out.

These ingress connectors and egress connectors can be further divided into 2 categories as one-directional and bi-directional. One-directional ingress connectors only accept messages from an outside system while bi-directional ingress connectors accept messages and also send back a response for the received message. Similarly, one-directional egress connectors only send messages to an outside system while bi-directional egress connectors send messages out and also receive a response for the sent message.

The following connectors and many more are currently supported by the ESB.

★ HTTP (Netty/NIO) Ingress and Egress

★ JMS Ingress and Egress

★ AS2 Ingress and Egress

★ File Polling Ingress and Egress

★ File NIO Ingress

★ SFTP Ingress and Egress

★ FIX Ingress and Egress

★ AMQP Ingress and Egress

★ IBM-MQ Egress

★ SCP Ingress and Egress

★ Timer Ingress

Any Integration Flow in UltraESB-X must be started with an ingress connector element.

Processors

Processors (Processing Elements) are used to process a message inside an integration flow. Each processing element accepts a message context, performs a specific processing on it and emits one or more message contexts depending on the type of processing.

Depending on the type, each processing element has one or more outports. These outports also has 2 different types as single-outports and multiple-outports. A single-outports can be connected only to a single processing element/connector while a multiple-outport can be connected to more than one processing elements/connectors.

One of the outports of a processing elements is always an "On Error" outport, to which the message context is directed in case of an error occurred while processing.

Currently the ESB supports several processing elements under the categories such as,

★ Header management

★ Scope management

★ Scope variable manipulation

★ Payload transformation

★ Payload validation

★ Payload extraction

★ Attachment processing

★ Logging and auditing

Section 2

Installing UltraStudio

This section describes how to install UltraStudio within IntelliJ IDEA and set it up properly.

If you have already downloaded the IntelliJ IDEA community edition where the UltraStudio is pre-built in, you can skip this section. On the other hand if you have downloaded only the UltraStudio plugin, follow this section to install it properly on IntelliJ IDEA.

Integrating UltraStudio with IntelliJ IDEA

Prerequisite:

★ Oracle JDK 1.8

★ Maven 3.3.x

★ IntelliJ IDEA Community/Ultimate Edition 2017.1 or newer version

Open IntelliJ IDEA settings window (`Files -> Settings`) and under plugins section, click on "Install Plugin from disk..." button. Next select the "UltraStudio-17.07.zip" file and click on open button (figure a). After that UltraStudio will be integrated with IntelliJ IDEA and you will need to restart IDEA to take effect.

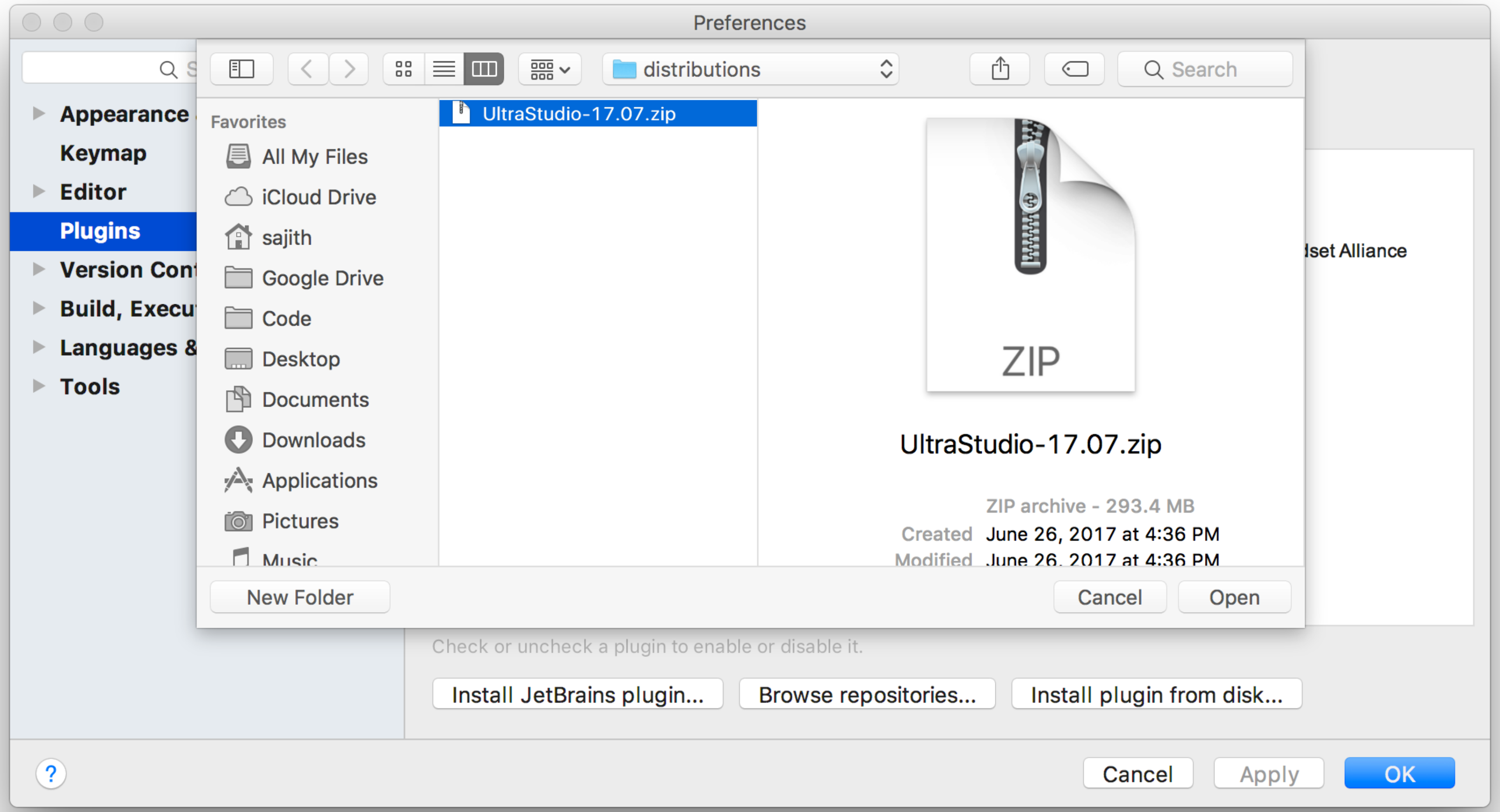


Figure a: Installing UltraStudio

Section 3

Project Creation

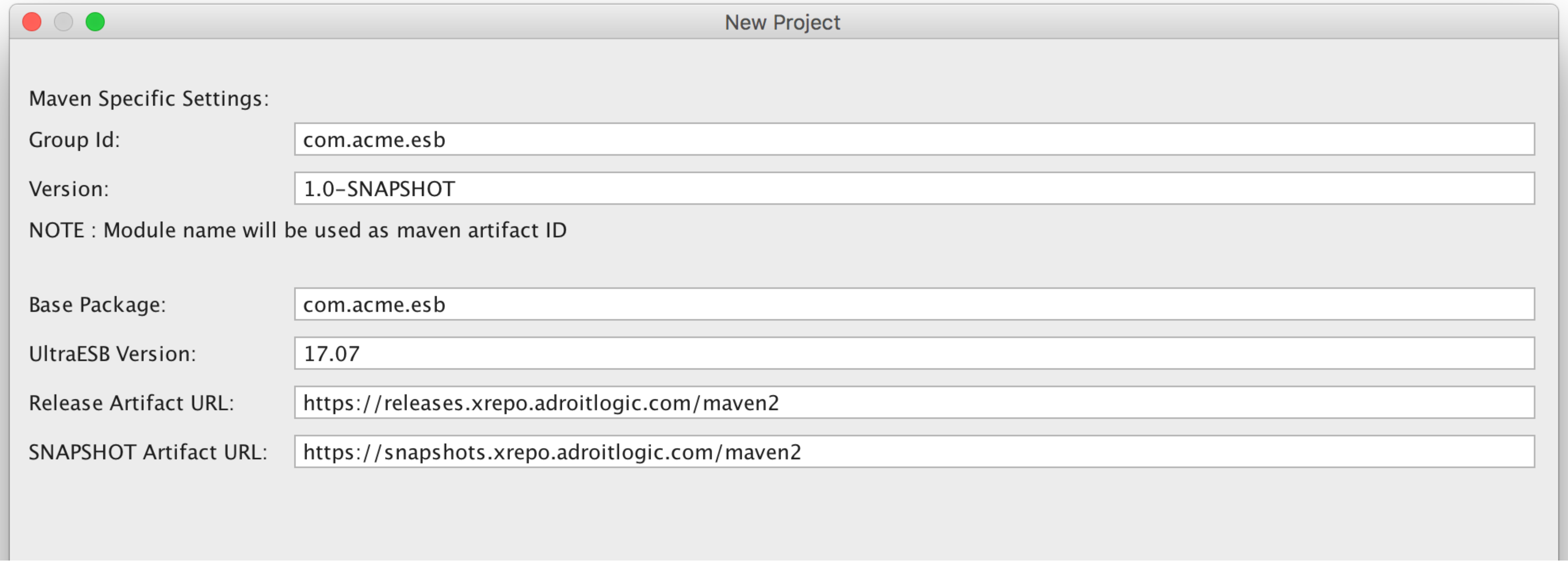
PROJECT CREATION

Within Ultra Studio, there are mainly two ways to create a new project. You can create a new project with only the required component dependencies and keep on building the integration flows you want.

Otherwise, if you are completely new to the Ultra Studio, the best way to start exploring UltraStudio is through a sample project. In the sample project repository, there are various types of sample projects and you can select one and create a project which includes the sample integration flows in that project. Once, you create a sample project, you can directly run it and send messages through it without any configuration.

Now let us see how we can create a project in each approach mentioned above in details.

Creating an Empty Project



Maven Specific Settings:

Group Id:	com.acme.esb
Version:	1.0-SNAPSHOT

NOTE : Module name will be used as maven artifact ID

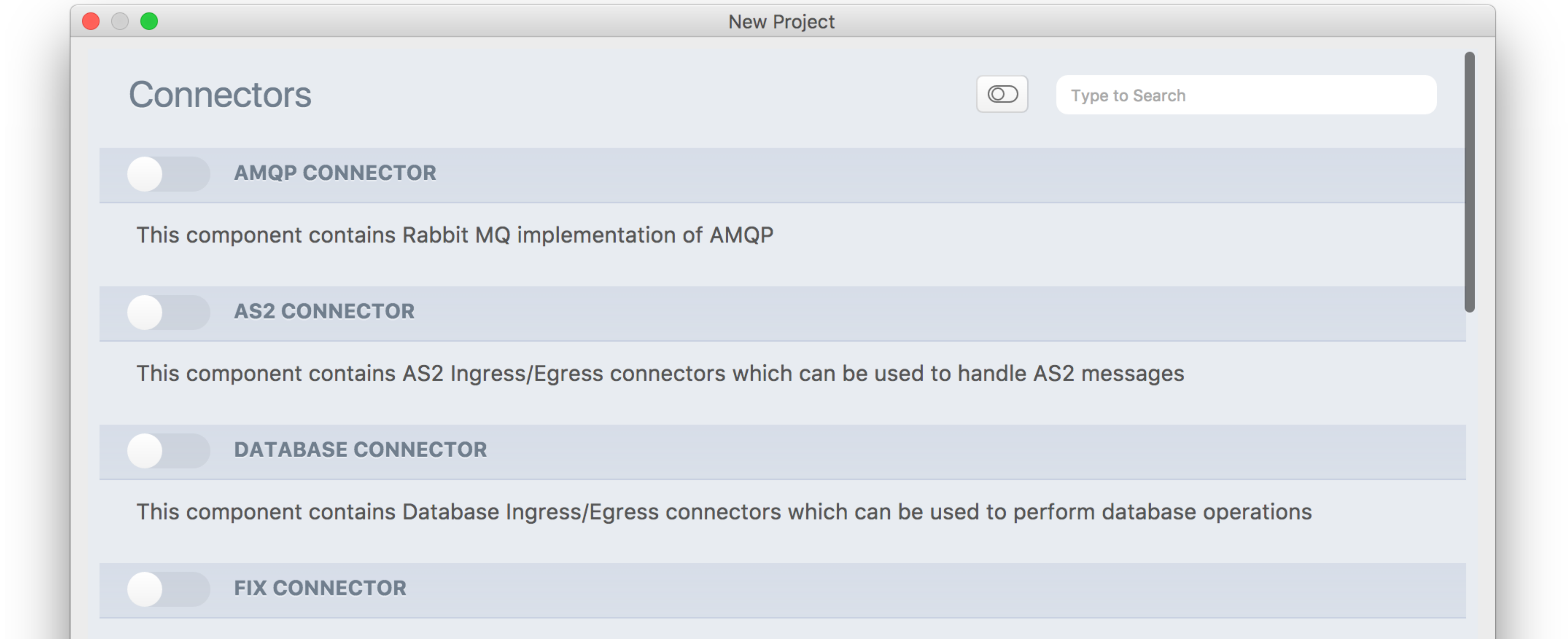
Base Package:	com.acme.esb
UltraESB Version:	17.07
Release Artifact URL:	https://releases.xrepo.adroitlogic.com/maven2
SNAPSHOT Artifact URL:	https://snapshots.xrepo.adroitlogic.com/maven2

Figure 3.1: New Project

In-order to create an empty project go to File menu and select **New Project**. Next select ***Empty Ultra Project*** and continue. After that, as shown in the figure 3.1, you will be presented with a wizard to enter the project details. Keep in mind that this new project will be a maven project and hence, you have to specify the maven specific properties such as *Group ID*, and *Version*. The base package represent the base package name for your custom Java classes and UltraESB version is the specific version of the ESB which should be used with this project.

Next, you have to specify the release artifact URL. This URL must point to the remote maven repository where the UltraESB release artifacts can be obtained. Similarly, the SNAPSHOT URL represent the remote maven repository where the SNAPSHOT artifacts of the UltraESB can be obtained.

NOTE: It is highly discouraged to use SNAPSHOT artifacts in production environment since they are highly brittle. You must always use release artifacts which are stable, in production environment



AMQP CONNECTOR

This component contains Rabbit MQ implementation of AMQP

AS2 CONNECTOR

This component contains AS2 Ingress/Egress connectors which can be used to handle AS2 messages

DATABASE CONNECTOR

This component contains Database Ingress/Egress connectors which can be used to perform database operations

FIX CONNECTOR

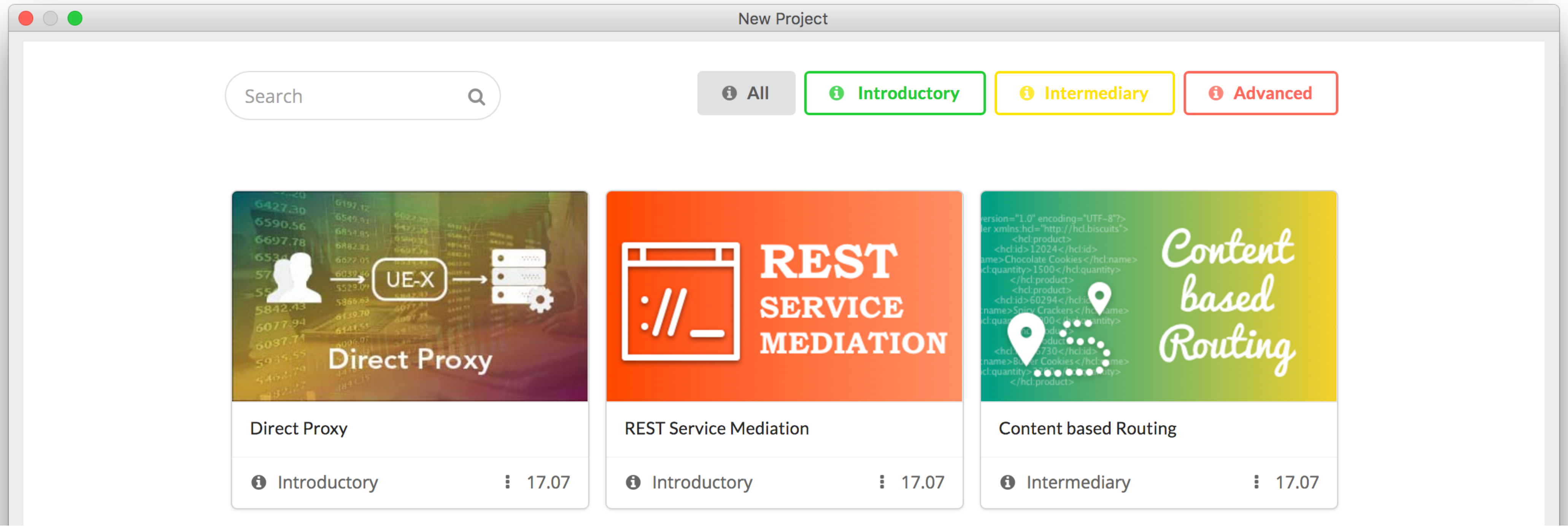
Figure 3.2: Connector Component List

After Clicking next button, you will be presented with the connector component list. This list contains various connectors which provide the facility to handle various transports such as HTTP/S, JMS, SFTP, etc. You can select connectors you want for your project from this list and that connector will be automatically added to you project as a dependency.

In the next step of the wizard, you will be presented with a list of processors which can be added to your project as dependencies. This processor list contains various types of processing elements which you can use in your integration flows to manipulate messages which are flowing through the UltraESB.

After selecting which components you want to be added to your project, you will be presented with the final step of the project creation wizard. In this step, you have to specify the name of your project and the location on your file system where this should be created. With that you are done with creating a new empty project and click on the finish button to continue.

Creating a Sample Project



Direct Proxy
Introductory 17.07

REST Service Mediation
Introductory 17.07

Content based Routing
Intermediary 17.07

Figure 3.3: Sample Repository

In-order to create a sample project, select ***Sample Ultra Project*** from the new project creation wizard. Next as shown in Figure 1, specify the project specific details and proceed by clicking the next button.

Then the sample repository will be loaded and you can directly download the sample you want by clicking on the download button (Figure 3.3) or you can read more about the sample by clicking on the view button.

Mainly there are three types of samples in the *Sample Repository* and those are '**Introductory**', '**Intermediary**' and '**Advanced**'.

If you are new to UltraStudio, you should start with an Introductory sample to get an idea about the whole concept.

After selecting the sample, it will be automatically downloaded and click on the next button and specify the location on your file system where the project should be created.

Section 4

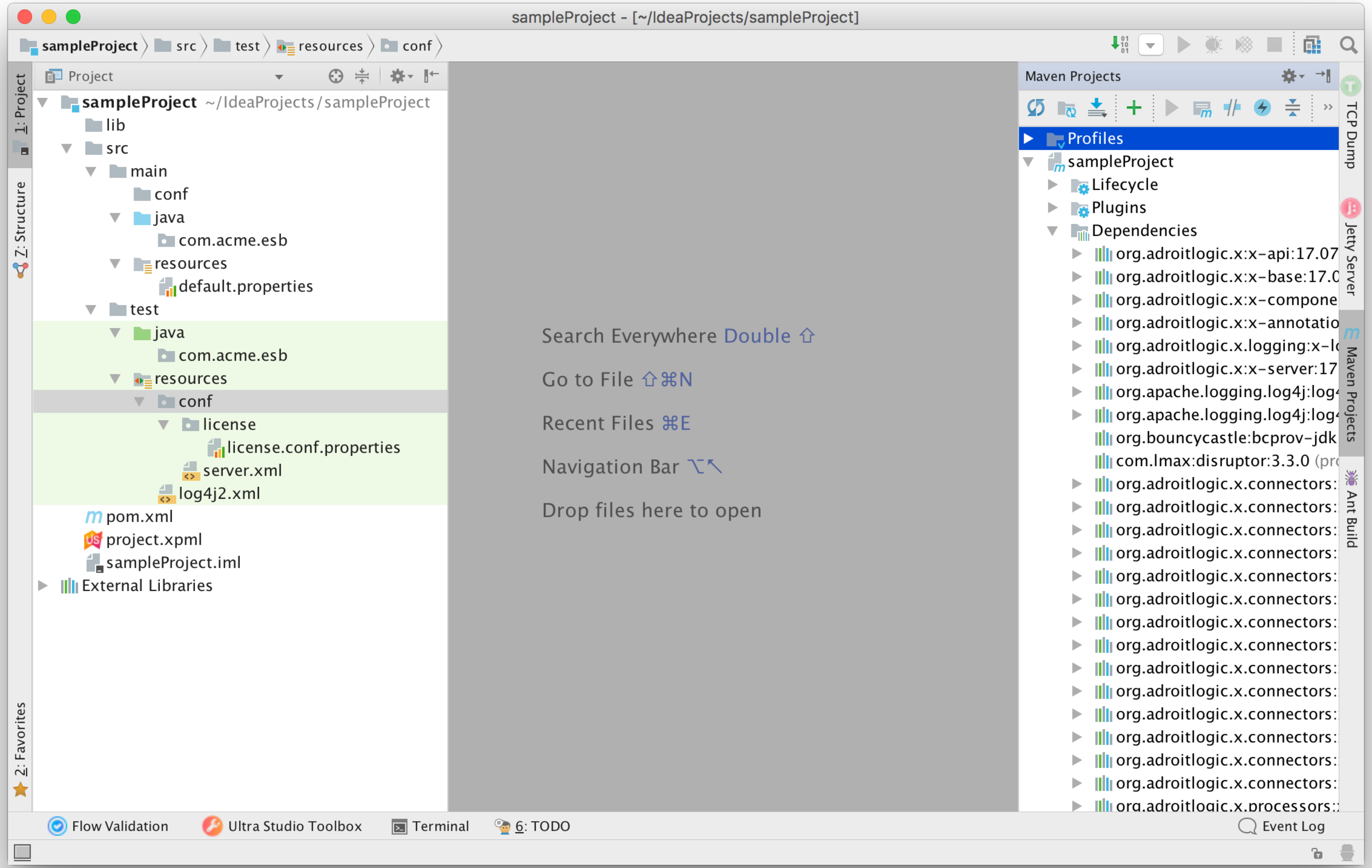
Overview of Project Structure

OVERVIEW OF PROJECT STRUCTURE

In this chapter the basic component of an Ultra Project and the main components of the UltraStudio integration flow development environment will be discussed.

After creating an Empty Ultra Project as described in the previous section, the overall project structure will look as Figure 4.1. In the maven Projects tab, all the dependencies specified in the pom.xml file will be shown as well.

Further, within the main directory, there is a directory named ***conf*** and this is where all your integration flows should reside. Apart from that the project structure is similar to an empty Maven Project.



```
sampleProject ~/IdeaProjects/sampleProject
    lib
    src
        main
            conf
            java
                com.acme.esb
            resources
                default.properties
        test
            java
                com.acme.esb
            resources
                conf
                    license
                        license.conf.properties
                    server.xml
                log4j2.xml
    pom.xml
    project.xpml
    sampleProject.iml
External Libraries
```

```
Profiles
sampleProject
    Lifecycle
    Plugins
    Dependencies
        org.adroitlogic.x:x-api:17.07
        org.adroitlogic.x:x-base:17.0
        org.adroitlogic.x:x-compone
        org.adroitlogic.x:x-annotatio
        org.adroitlogic.x.logging:x-l
        org.adroitlogic.x:x-server:17
        org.apache.logging.log4j:log
        org.apache.logging.log4j:log
        org.bouncycastle:bcprov-jdk
        com.lmax:disruptor:3.3.0 (pr
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.connectors:
        org.adroitlogic.x.processors:
```

Figure 4.1: Project

Project Structure

Now let's see what are the new components in the Ultra Project.

Project.xpml File

This file contains the information about your project such as name, version and description.

```
<x:project id="mySampleProject" name="mySampleProject"
version="1.0-SNAPSHOT"
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:x="http://www.adroitlogic.org/x/x-project"
xsi:schemaLocation="http://www.adroitlogic.org/x/x-project
http://schemas.adroitlogic.org/x/x-project-1.0.0.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.1.
xsd">

    <x:description>
     A sample project that demonstrates the projects concept
     </x:description>

    <x:flows>
        <x:flow id="sampleFlow" file="helloWorldFlow.xcml"/>
    </x:flows>

    <x:resources>
    </x:resources>

</x:project>
```

Figure 4.2: Sample project.xpml file

Further, within the ***flows*** tag, all the integration flows in your project where file path is specified relative to the *conf* directory will be added. When you run the project, only the integration flows specified in this section will be deployed

Apart from that within the resources tag, you can specify re-usable complex spring beans which in return used by components in your integration flow. We will look into in details in What is project.xpml section.

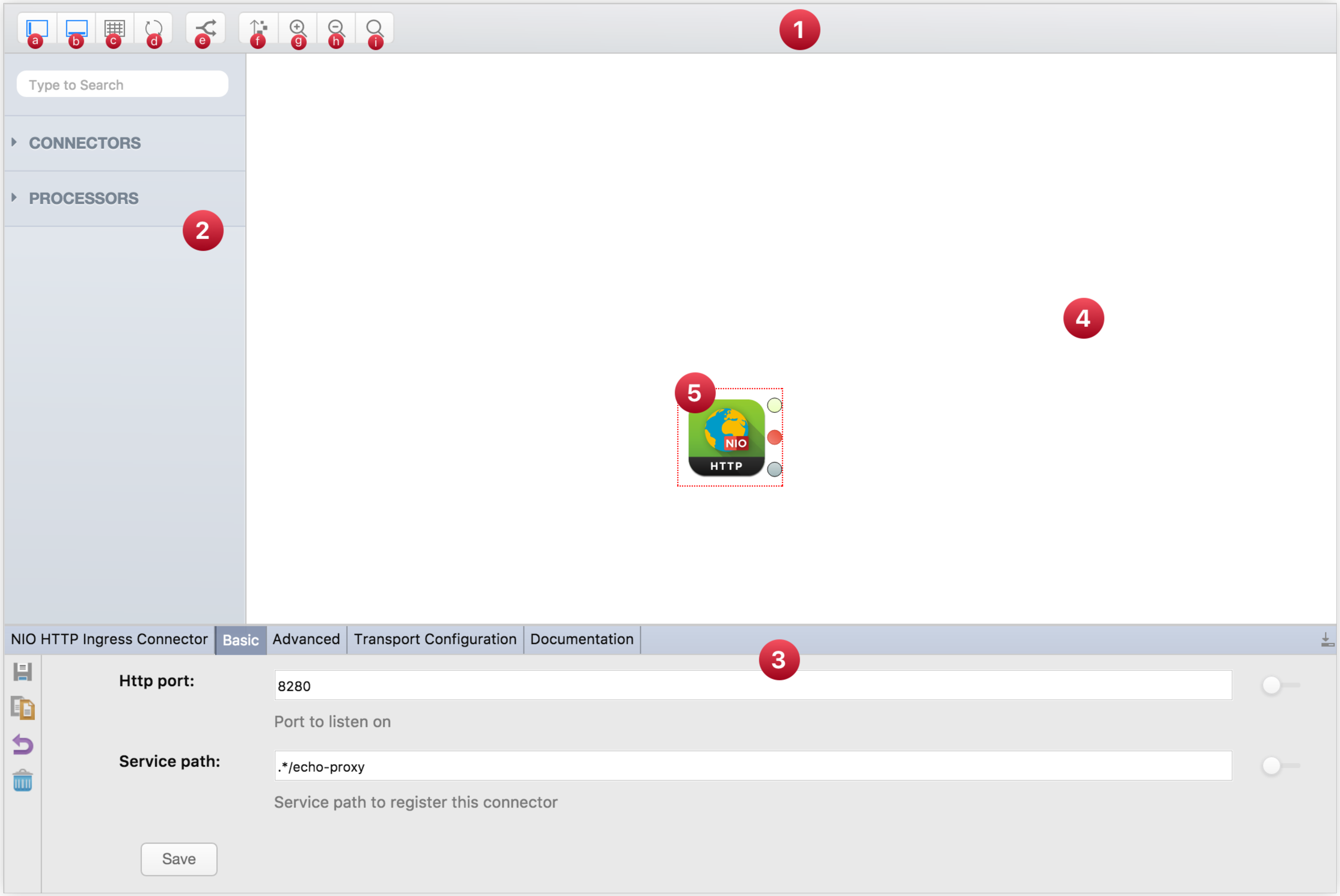
Conf Directory

This is where all your integration flows must reside. It is not recommended to create integration flows anywhere else in the project. Although, you can create sub-directories within conf directory to organize your flows.

Apart from these main two items, there are few files in the test/resources directory and for now you do not need to worry about those.

Now let's create our first integration flow. Keep in mind that all the integration flows which are created must reside within the conf directory. First right click on the conf directory and from the context menu, select Integration Flow. After that specify helloWorldFlow as the flow name and click on OK button.

Now you can see that there is a new file added to the conf directory and double click on it to open it in the editor view.



NIO HTTP Ingress Connector | Basic | Advanced | Transport Configuration | Documentation

Http port: 8280

Port to listen on

Service path: .*/echo-proxy

Service path to register this connector

Figure 4.3: Design View

Main Components of the Editor View

1. Main Toolbar

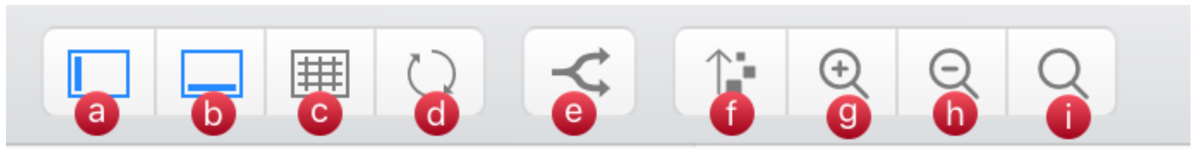


Figure 4.4: Main Toolbar

The main toolbar (Figure 4.3:1) contains all the important functionalities which affect the design view. Below list contains brief description of each button and the respective functionality of that button (Figure 4.4).

a. Toggle the Component Pallet's visibility

b. Toggle the Property Pane's visibility

c. Toggle the visibility of grids on the design canvas

d. Refresh the Design Canvas

e. Highlight the message execution path in the integration flow

f. Scroll to the beginning of the integration flow

g. Zoom in the Design Canvas

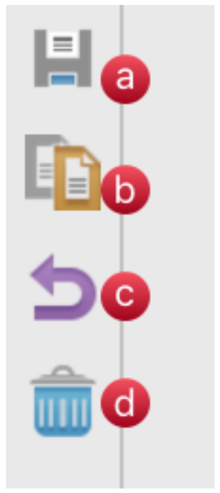
h. Zoom out the Design Canvas

i. Reset the Zoom view of Design Canvas

2. Component Pallet

This pallet (Figure 4.3:2) contains all the components required to create an integration flow. There are ingress/egress connectors, processors, and so on. If you want to refresh your memory on what these are, refer to the core-concepts of project-x chapter.

3. Properties Pane



This pane (Figure 4.3:3) contains all the properties which can be specified by the user for a particular component. i.e. for a given component to function properly, user has to manually specify some properties and these properties varies from one component to another. Apart from that, this pane contains a *Documentation* tabs which provides the basic documentation of the specific component.

Further, the property pane contains few functionalities to manipulate the component properties such as

a. Saving the modified configuration properties of a component

b. Clone the component with the existing configuration properties

c. Reset the configuration properties of the component

d. Delete the component from the design pane

Apart from that, in the property pane for each property, there is a slider at the right side of the property pane. If you hover over it, a tooltip with the message

```
Externalize Property as : <PropertyName>
```

will be shown. If the slider is enabled, the actual property value will be written to the `src/main/resources/default.properties` file and a placeholder will be added to the integration flow file.

This feature is useful when you want to externalize the property values when running the project in UltraESB-X container. After building the project, if you want to modify the value of an externalized property, you can do it easily without doing any modification to the project bundle by modifying the respective *properties* file in UltraESB-X container.

4. Design Canvas

This is where you can create your integration flow. You can drag and drop component from the component pallet in to the design pane and keep on creating your flow (Figure 4.3:4).

5. Components

Each component is unique and designed to execute a specific logic. As shown in figure 4.5 the circles surrounding the component icon are called ports. Mainly there are four port types.

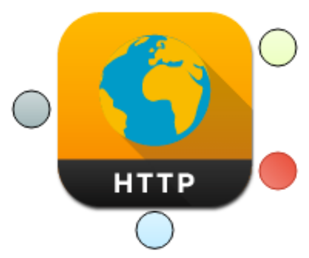


Figure 4.5: Sample Component

✦ *In-Port (grey color circle)* - This is where messages are given as input to the component. A component can have only one in-port and specific type of components such as ingress connectors have zero in-ports

✦ *Out-Port (green color circle)* - This is where the output message is emitted from the component. There can be zero or more outports for a particular component.

✦ *Side-ports (blue color circle)* - These ports are used to connect other components as parameters for a particular component. There can be zero or more side ports for a particular component.

✦ *Exception-Port (red color circle)* - This is a sub set of out ports and an output is emitted from the component through this port if an exception or an error occurs while executing the component's logic, so that the user can gracefully handle it.

Section 5

Developing an Integration Flow

DEVELOPING AN INTEGRATION FLOW

In this chapter let's see how we can develop, run and debug a very basic Integration Flow in the UltraStudio.

In this integration flow, let's create an Echo HTTP endpoint where when we run the UltraESB-X with the developed integration flow deployed we expose an HTTP resource endpoint so that we can send HTTP requests with any payload and the response for the HTTP request will be the same as the content of the request payload.

NOTE: For this project it is required to select **HTTP NIO CONNECTOR** from the connector list when creating the project. Refer to Section 3, in-order to refresh your memory on how to create a project with connectors and processor components.

Creating an Integration Flow File

First right click on the `src/main/conf` directory and from the context menu, select *Integration Flow* as shown in figure 5.1

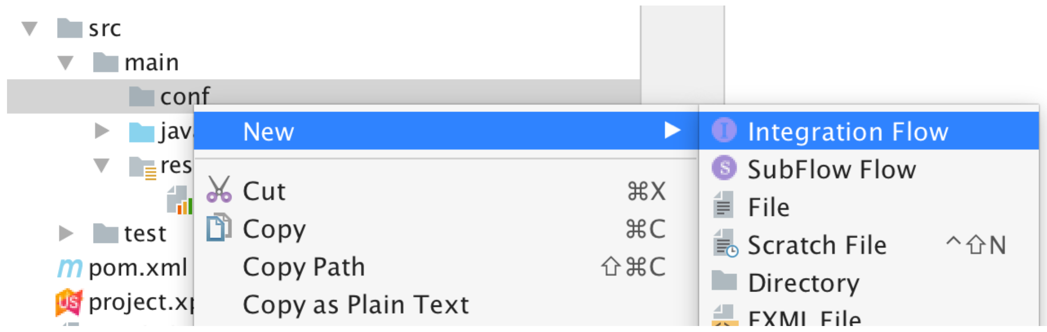
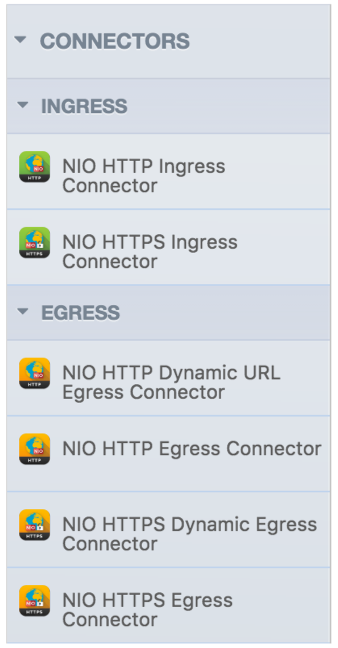


Figure 5.1: Context Menu

Next, specify *HelloWorld* as the name of the Integration flow and create our first Integration flow.


CONNECTORS
INGRESS
NIO HTTP Ingress Connector
NIO HTTPS Ingress Connector
EGRESS
NIO HTTP Dynamic URL Egress Connector
NIO HTTP Egress Connector
NIO HTTPS Dynamic Egress Connector
NIO HTTPS Egress Connector

Now Let's add our components to the Design Canvas so that we in-order to complete the flow. First, select the **NIO HTTP Ingress Connector** from the component pallet and drag-and-drop it into the design canvas. You will be presented with a dialog box to specify a name to the component and let's specify *http-in-component* as the name. Next, drag-and-drop the **NIO HTTP Egress Connector** to the design canvas and as for its name, let's specify *http-out-component*.

After that, we need to connect these two component as shown in figure 5.2. We have to connect the ***Processor*** out port of the HTTP NIO Ingress Connector with the in-port of the HTTP NIO Egress Connector and the ***Response Processor*** out-port of the HTTP NIO Egress Connector with the in-port of the HTTP NIO Ingress Connector.

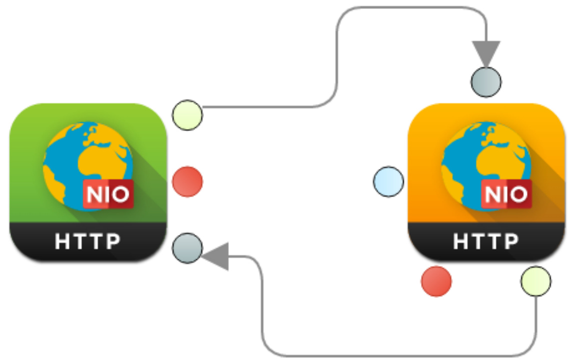


Figure 5.2: Connected Integration Flow

We have created our first Integration Flow. Now all we have to do is to specify the required component properties.

Running Flow Validation

Next, we need to run the Flow Validator to findout if there is any error in out integration flow. The Flow Validator resides at the bottom of UltraStudio tool pane as shown in figure 5.3.

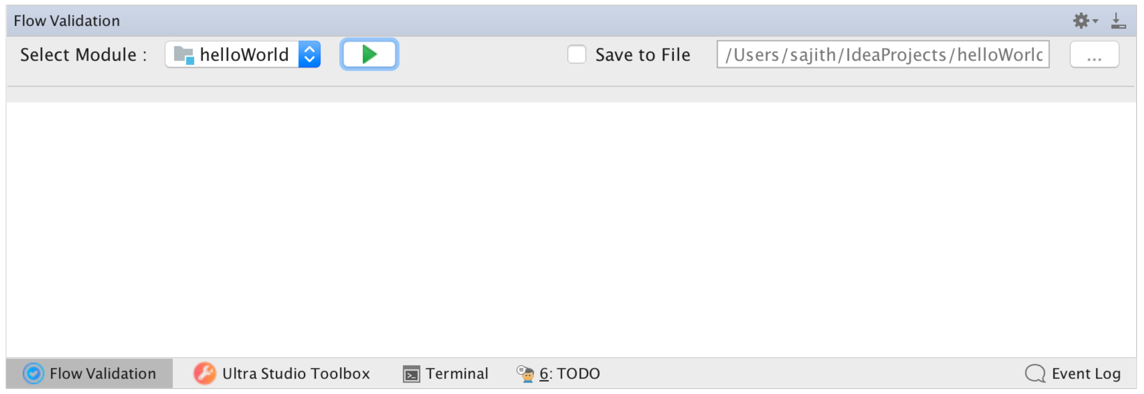
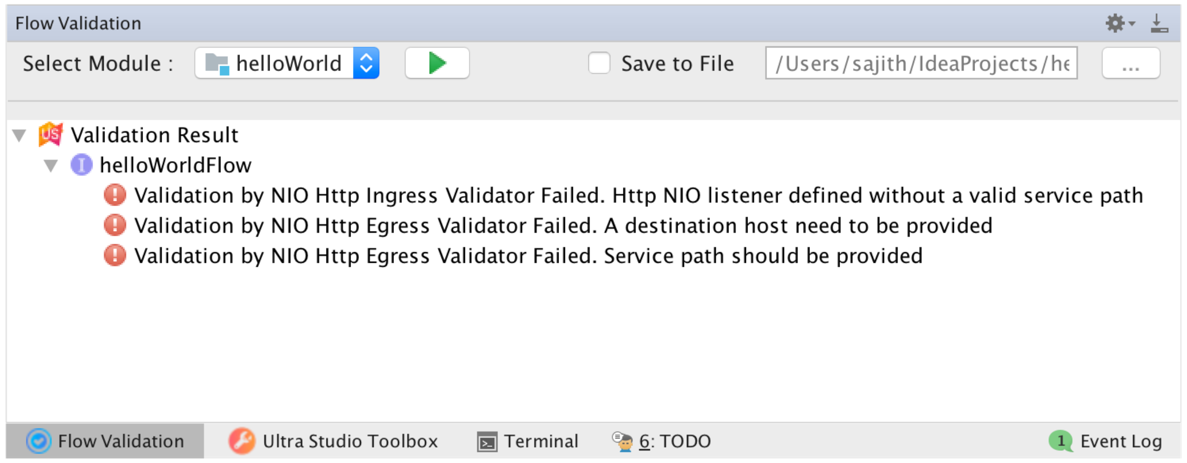


Figure 5.3: Flow Validation Panel

Click on the green colored run button to start the flow validator and after the completion of flow validation you will see the result as shown in figure 5.4



```
Validation Result
  helloWorldFlow
    Validation by NIO Http Ingress Validator Failed. Http NIO listener defined without a valid service path
    Validation by NIO Http Egress Validator Failed. A destination host need to be provided
    Validation by NIO Http Egress Validator Failed. Service path should be provided
```

Figure 5.4: Flow Validation Result

As you can see, there are couple of error within our flow. If you double click on a error result, it will navigate you to the erronous component.

These error are expected since we have not specified the mandatory proprties of the components on the design canvas. If we are to run the integration project in current state, it will fail since the integration flow is incomplete.

As you can see, Flow Validator can be used to makesure the integration flow you have developed is in correct state and all the requirements are fullfiled before running the project to avoid erronous scenarios due to invalid or incomplete configuration.

Configuring Component Properties

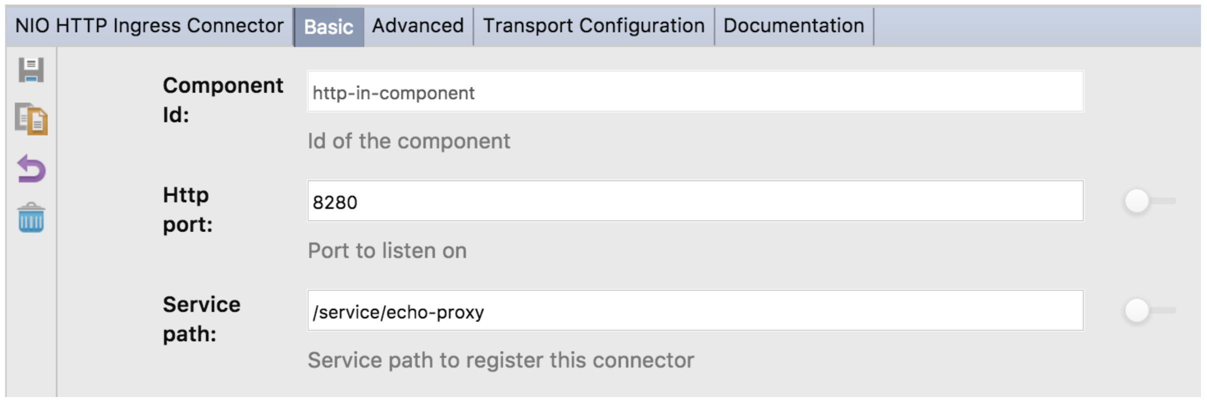
Now let’s specify the required properties in the each component.

First, click on the NIO HTTP Ingress Connector and there are mainly two mandatory properties we need to specify under the Basic Tab (Figure 5.5).

HTTP Port - The port we want our HTTP Ingress Connector to start listening for incoming request. For this example let’s specify `8280`.

Service Path - The resource path where we want to filter out the incoming requests. Specify `/service/echo-proxy` for that.

After specifying those properties, click on the save button. Now we have configured our HTTP Ingress Connector to listen at `http://localhost:8280/service/echo-proxy` for incoming requests.



Component Id: http-in-component
Id of the component

Http port: 8280
Port to listen on

Service path: /service/echo-proxy
Service path to register this connector

Figure 5.5: Property configuration for NIO HTTP Ingress Connector

Similarly, we need to configure our NIO HTTP Egress Connector as well (Figure 5.6).

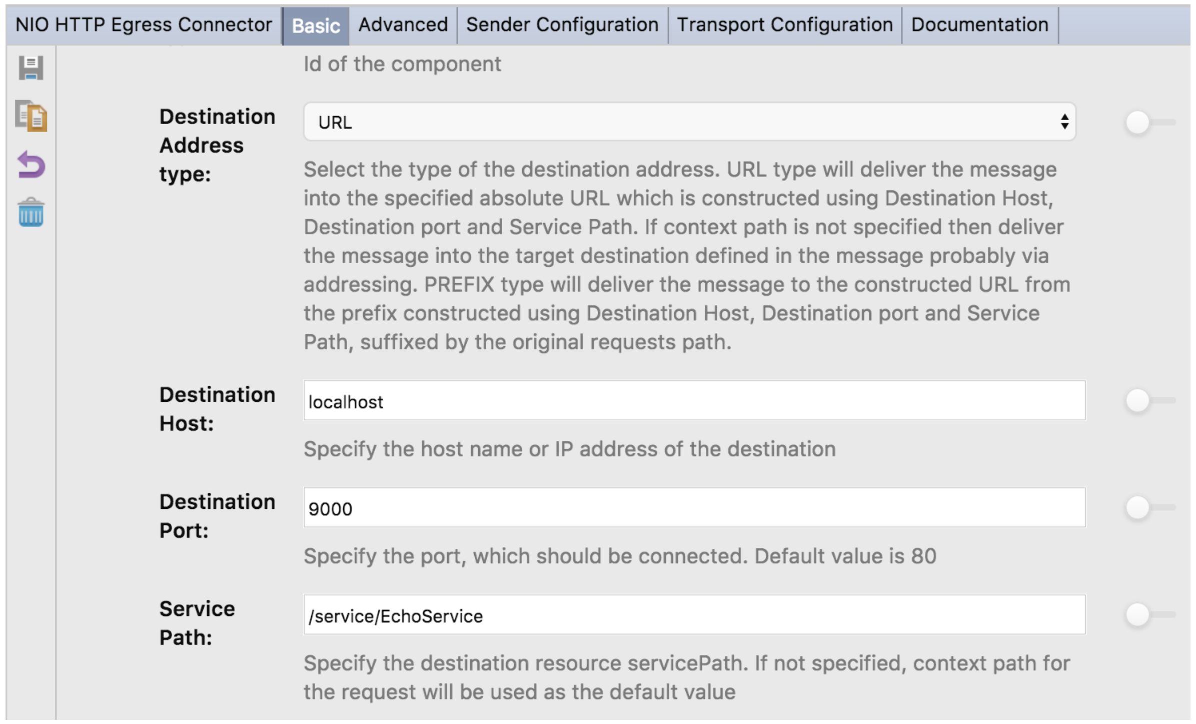
Destination Address Type - The type of the destination address. For this example, select `URL`.

Destination Host - Host name of the destination endpoint. For this example, specify `localhost`.

Destination Port - The remote port we want to connect. Specify `9000` as the value.

Service Path - The remote service path we want to connect. Specify `/service/EchoService` for that

After that save the configuration by clicking on the Save Button.



Id of the component

Destination Address type: URL
Select the type of the destination address. URL type will deliver the message into the specified absolute URL which is constructed using Destination Host, Destination port and Service Path. If context path is not specified then deliver the message into the target destination defined in the message probably via addressing. PREFIX type will deliver the message to the constructed URL from the prefix constructed using Destination Host, Destination port and Service Path, suffixed by the original requests path.

Destination Host: localhost
Specify the host name or IP address of the destination

Destination Port: 9000
Specify the port, which should be connected. Default value is 80

Service Path: /service/EchoService
Specify the destination resource servicePath. If not specified, context path for the request will be used as the default value

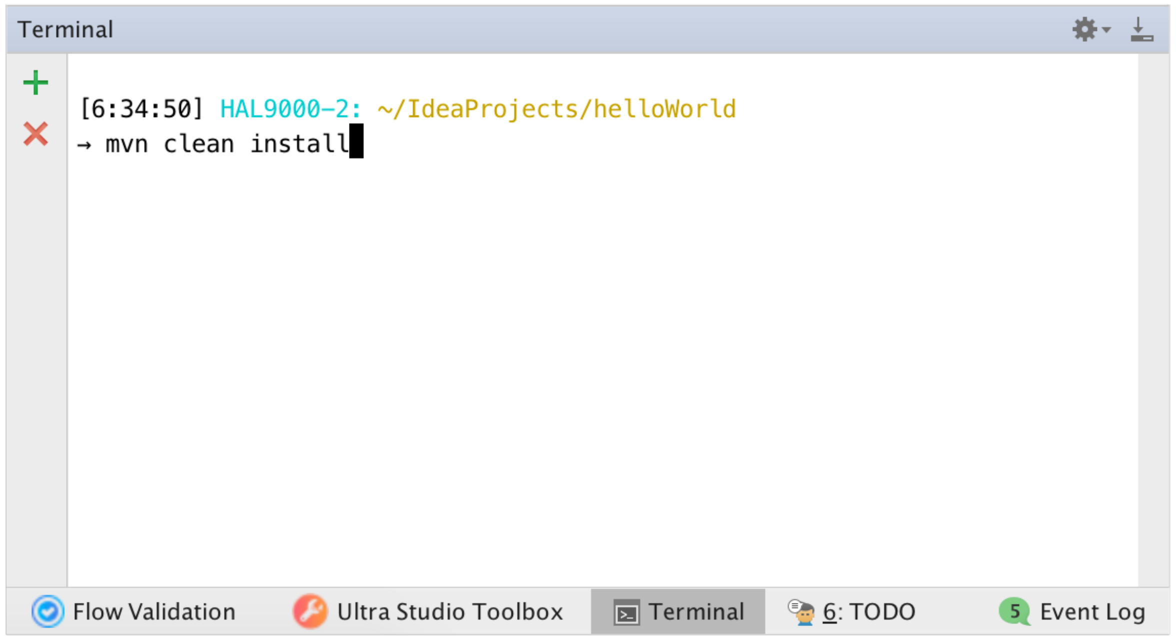
Figure 5.6: Property configuration for NIO HTTP Egress Connector

Now if you run the flow validator again, you will be able to see that flow validation has passed successfully.

Building and Running the Project

First let's compile the project. Note that it is not mandatory to compile the project each time you want to run the project (it is done automatically before running the project). But for the moment, let's see how we can compile and build our project bundle.

First open the terminal tab and execute `mvn clean install` command as shown in the figure 5.7



```
[6:34:50] HAL9000-2: ~/IdeaProjects/helloWorld
→ mvn clean install
```

Figure 5.7: Terminal Tab

After successfully building the project, you will see an output as shown below on in the terminal.

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.952 s
[INFO] Finished at: 2016-10-07T10:47:50+05:30
[INFO] Final Memory: 28M/402M
[INFO] ------------------------------------------------------------------------
```

Figure 5.8: Terminal output

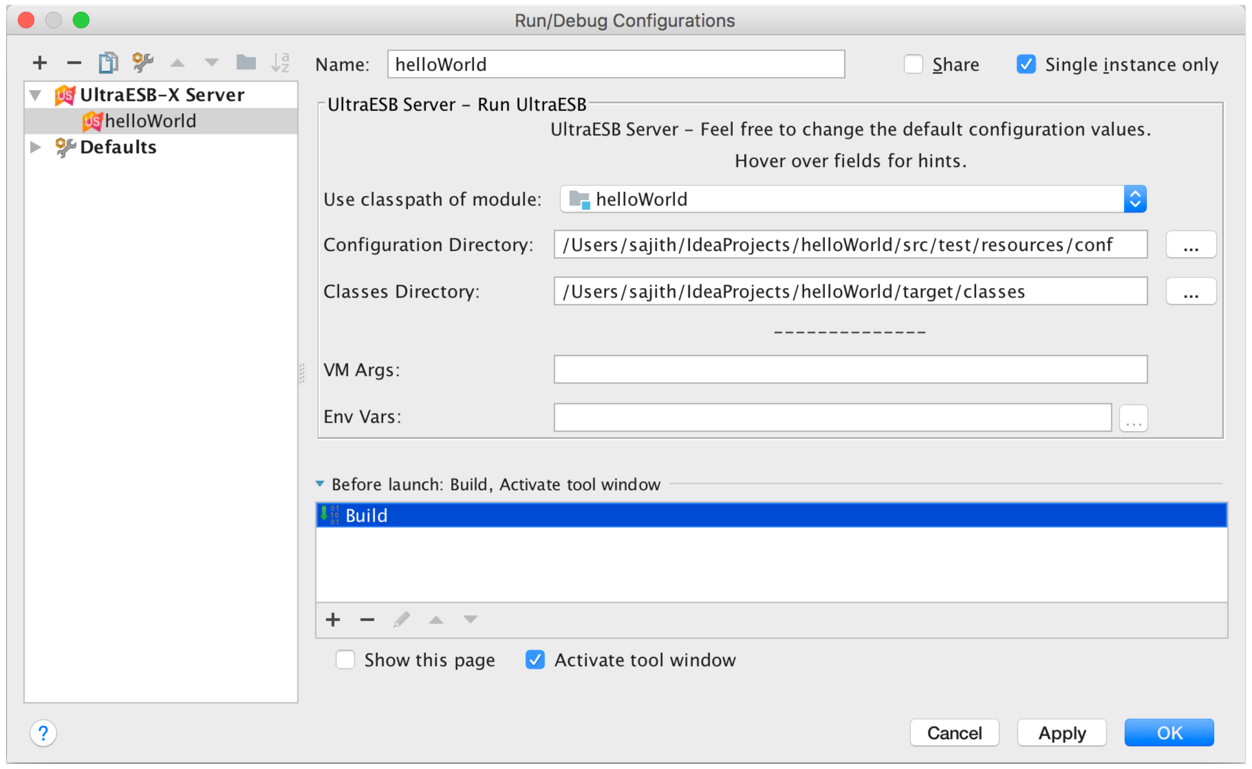
NOTE: When building the project, maven will fetch the dependencies from the remote repository. Although if those dependencies are already cached in the M2 home, maven will use the dependencies in the local cache instead of downloading them from the remote repository

Now our project is bundled into a file with the extension **.xpr** and if you inspect the **target** directory within the project you will be able to see it.

Now we need to create the run configuration. First click on

`Run → Edit Configurations…`

menu item and open the Run/Debug Configuration window. Next from the context menu shown after clicking on the + icon select UltraESB-X server, and specify a name for your configuration (figure 5.9) and click on OK button.



Run/Debug Configurations

Name: helloWorld

☐ Share ☑ Single instance only

UltraESB-X Server
helloWorld
Defaults

UltraESB Server - Run UltraESB

UltraESB Server - Feel free to change the default configuration values.
Hover over fields for hints.

Use classpath of module: helloWorld

Configuration Directory: /Users/sajith/IdeaProjects/helloWorld/src/test/resources/conf

Classes Directory: /Users/sajith/IdeaProjects/helloWorld/target/classes

VM Args:

Env Vars:

Before launch: Build, Activate tool window

Build

☐ Show this page ☑ Activate tool window

Cancel Apply OK

Figure 5.9: Run/Debug Configuration

After that click on the run button on toolbar (figure 5.10)

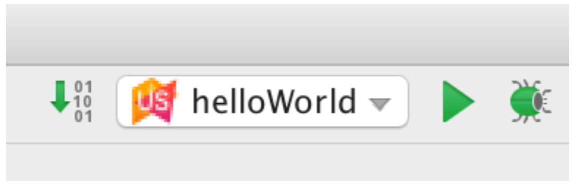


Figure 5.10: Run Button on Toolbar

Next, you will get a message as below

Client Key is Not Specified
Please specify the client key which you have received via email in UltraStudio Settings.

If you haven't received a client Key, please contact license@adroitlogic.com

When you have downloaded UltraStudio, you got an email which included a client Id and you need that Id, in-order to activate the UltraEBS-X server. Copy the clientId from the email and go to File → Settings (IntelliJ IDEA → Preference on MacOS) and go to Tools →Ultra Studio section. In there, under Client Key text box, paste the client key and save it.

After that, click on the run button again and you will see below output on run tab which signals that the UltraESB has started successfully.

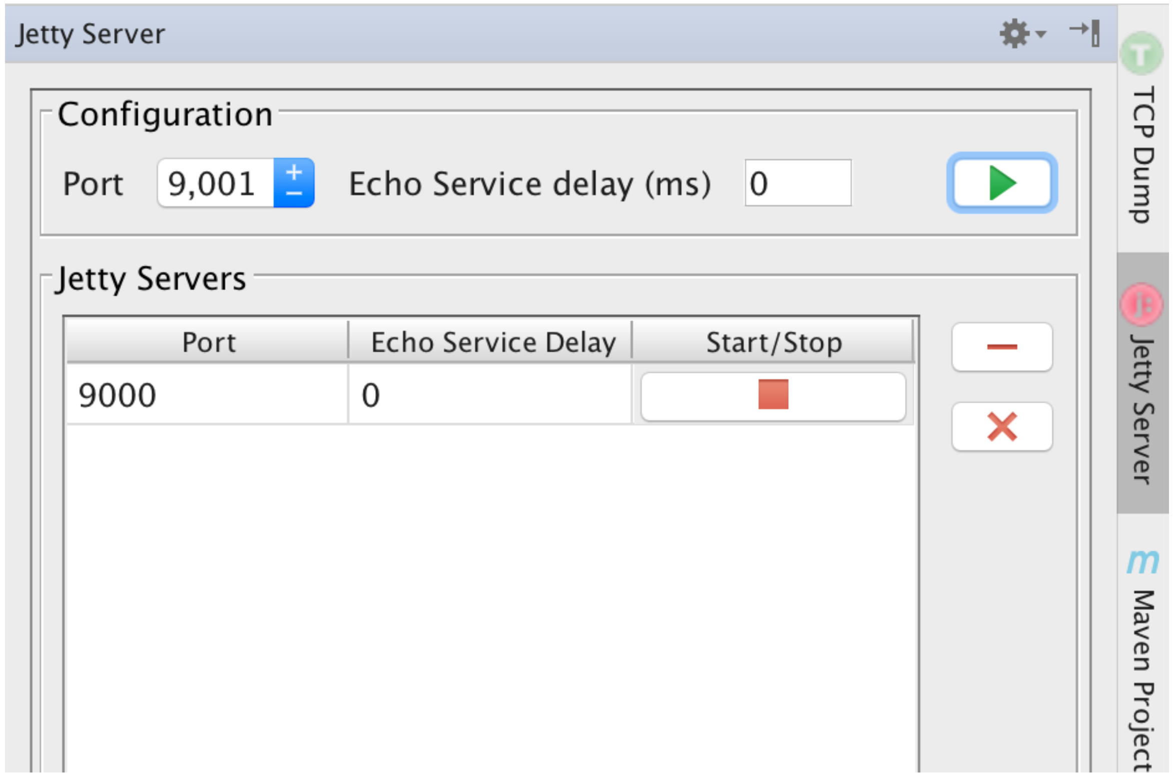
```
INFO XContainer AdroitLogic UltraStudio UltraESB-X
server started successfully in 2 seconds and 524
milliseconds
```

NOTE: If you haven't received a client key or you have misplaced it, you can obtain a new client key by dropping an email to license@adroitlogic.com

Sending Messages and Debugging Flow

In-order to send a message, first we need to start a backend echo service on `http://localhost:9000/service/EchoService`

Fear not, we provide a very basic jetty server for testing and debugging purposes within the Ultra Studio. As shown in figure 5.11 click on the start button (green play button) to start a Jetty server on port 9000.

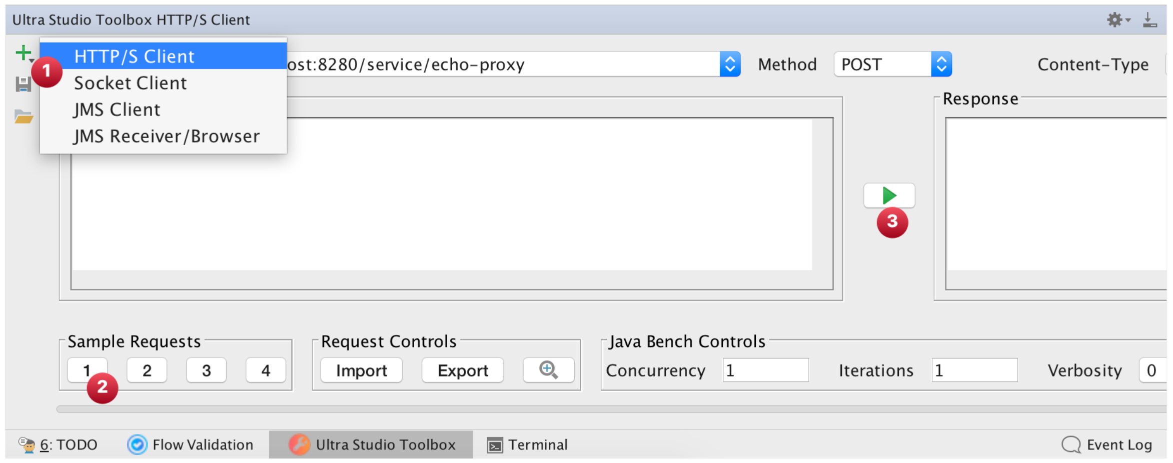


Port 9,001 Echo Service delay (ms) 0

Port	Echo Service Delay	Start/Stop
9000	0	🟥

Figure 5.11: Embedded Jetty Server Panel

After that in-order to send a sample message we can use the HTTP client integrated with the Ultra Studio. As shown in figure 5.12, create a new HTTP client (figure 5.12:1), select payload 1 (figure 5.12:2) and send it (figure 5.12:3) after changing the URL into `http://localhost:8280/service/echo-proxy`



ost:8280/service/echo-proxy Method POST

Concurrency 1 Iterations 1 Verbosity 0

Figure 5.12: HTTP Client Panel

Now in the response window, you can see the response, and the response payload is as same as the request payload since we sent the message to an echo server. (Make sure UltraESB-X server is up and running as described in *Building and running the project* section)

One other useful feature of Ultra Studio is that we can view the message execution path in the flow we have created. Click on the highlight message execution path icon on main toolbar and it will highlight the execution path as shown in figure 5.13.

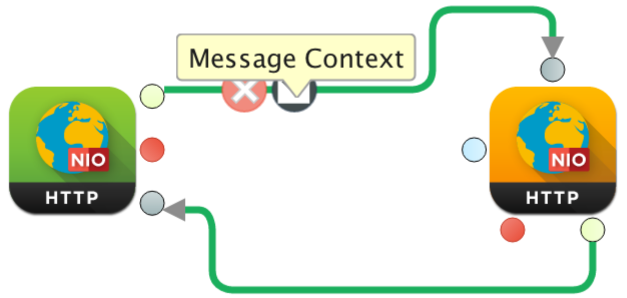
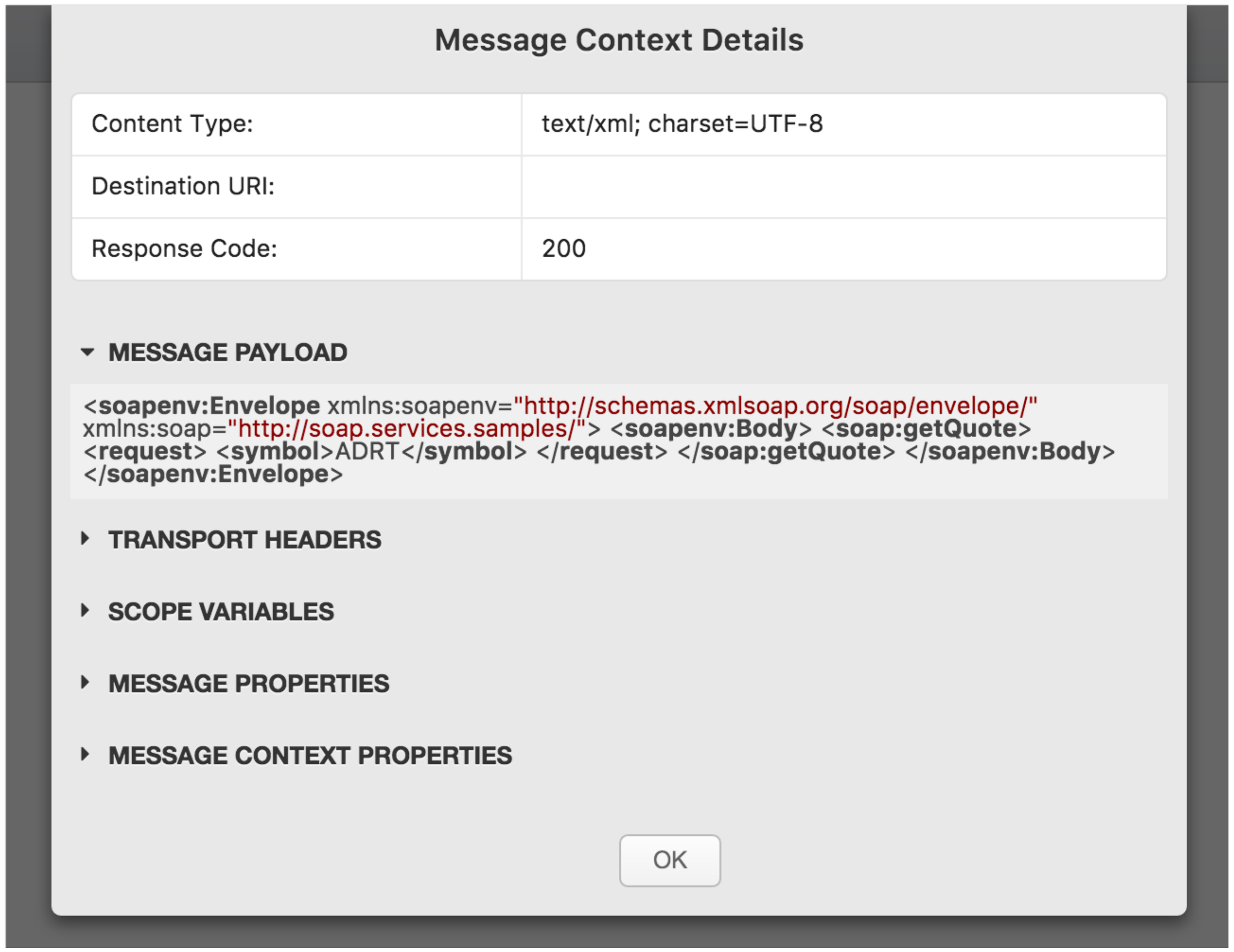


Figure 5.13: Highlighted Message Execution Path

This feature comes in handy when there is a complex message flow and you want to check which branches of the flow got executed while processing the message.

Further, if you click on the message icon for a particular link, you can see the full message context which passed through that particular link as shown in figure 5.14.



Message Context Details

Content Type:	text/xml; charset=UTF-8
Destination URI:	
Response Code:	200

▾ MESSAGE PAYLOAD

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soap="http://soap.services.samples/"> <soapenv:Body> <soap:getQuote>
<request> <symbol>ADRT</symbol> </request> </soap:getQuote> </soapenv:Body>
</soapenv:Envelope>
```

Figure 5.14: Message Context View

Section 6

Writing Custom Processing Elements

In Ultra Studio, you can write your own custom processing elements to execute your own logic as well. In this section let's see how we can write a very basic hello world processing element.

This element will obtain an input, i.e. the user name from the user as a property and write "`Hello ${username}`" to the console.

First within `src/main/java` directory in the `com.acme.esb` package create a new processing element by right clicking on the package name, and selecting `New → Processing Element`. Specify ConsoleLogger as the Class Name.

After that a Java class will be created as shown in figure 6.1. As you can see, `ConsoleLogger` is extended from the `AbstractProcessingElement` class. Whenever you write a custom processing element, you must extend from this class since it abstract out all the functionalities required by Project-X framework to use the custom processing element properly.

Further, there is an annotation named `Processor` associated with the Processing Element class. This annotation is used to obtain information about this processing element in the design view.

```
package com.acme.esb;

import org.adroitlogic.x.api.ExecutionResult;
import org.adroitlogic.x.api.XMessageContext;
import org.adroitlogic.x.annotation.config.Processor;
import org.adroitlogic.x.api.config.ProcessorType;
import org.adroitlogic.x.base.processor.AbstractProcessingElement;


@Processor(displayName = "", type = ProcessorType.CUSTOM,
           description = "")
public class ConsoleLogger extends AbstractProcessingElement {

    @Override
    public ExecutionResult process(XMessageContext
                                                  messageContext) {
        //TODO: execute any end processing logic
        return ExecutionResult.SUCCESS;
    }
}
```

Figure 6.1: Processing Element code template

As for the processor annotation, for now we only need to specify a display name (which will be shown in the component pallet) and a description.

Further, we need to set the *requireConfiguration* property to be *true* so that when our custom processing element is drag-and dropped on to the design canvas, the property pane for the component will be automatically shown. Below list briefly describe each property in the annotation

✦ **type** - This represent the subcategory the custom processing element belongs to under the processors category. You can specify any value (GENERIC, TRANSFORMER, FLOW_CONTROLLER, VALIDATOR, CUSTOM, EIP, SCOPE, ERROR_HANDLING) under org.adroitlogic.x.api.config.ProcessorType enum.

✦ **customType** - You can specify any value for this and this value will be used to categorize your processing element when shown in the component pallet.

✦ **displayName** - This value is showed in the component pallet to represent the processing element along with the icon of the element

✦ **iconFileName** - Name of the icon file to be assigned with this custom processing element.

NOTE: You need to specify the full URL of the icon file. It MUST be a resource which is accessible over HTTP/S protocol.

✦ **scope** - Mainly there are Integration Flows and Sub Flows. You can specify any value (ALL, NONE, INTEGRATION_FLOW, SUB_FLOW) from the `org.adroitlogic.x.api.config.ScopeType` enum. If you specify INTEGRATION_FLOW, then this element will be shown only in integration flow file's component pallet, and same goes for SUB_FLOW as well. Default value is ALL i.e. the particular component will be shown in all flow files.

★ **description** - A brief description to explain the functionality of your element. This will be shown under documentation tab in the property pane

★ **requireConfiguration** - If the value is true, when a user add this component to the design canvas, the property pane will be shown automatically and if the value is false, the property container will not be shown after adding this component to the design pane

★ **documentationURL** - URL of the resource which contains a comprehensive documentation for this processing element. This link will be shown under Documentation tab in the property pane.

For more information on annotation properties, refer the API documentation (https://developer.adroitlogic.org/project-x/api/17.07/apidocs/org/adroitlogic/x/annotation/config/Processor.html)

Now let's modify the annotation properties of our ConsoleLogger as shown in figure 6.2.

```
@Processor(displayName = "Console Logger",
           type = ProcessorType.CUSTOM,
           requireConfiguration = true,
           description = "Console Logger Processing element
                          adds a log line to the console")
```

Figure 6.2: Processor annotation

Now we can move on to writing our custom logic. We need to write our logic within

`process(XMessageContext messageContext)`

method because, when the framework execute the message flow, this method will get executed. Within the process() method, you will get the messageContext as a parameter. If you do not remember what message context is, refer core concepts of Proejct-X to refresh your memory.

First of all we need the user's name as an input for our custom logic. Hence, we can add a string class variable and annotate it with `Parameter` annotation as shown in figure 6.3.

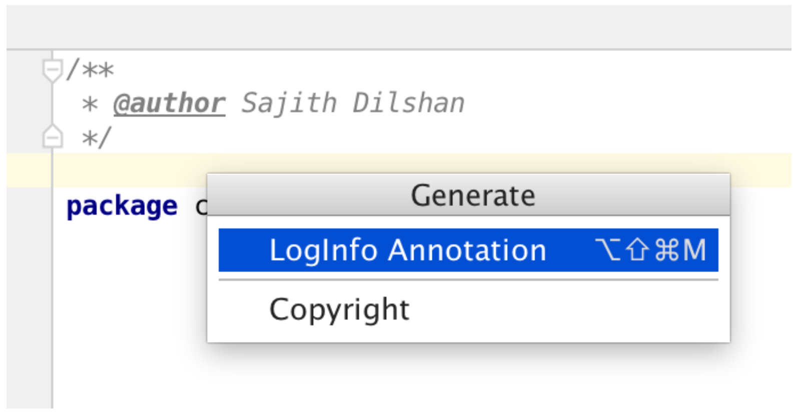
```
@Parameter(displayName = "User Name",
        inputType = InputType.TEXT_BOX,
        placeHolder = "Sajith",
        description = "Specify the username to be displayed on
                       the console")
private String username;
```

Figure 6.3: Parameter annotation

For a comprehensive documentation on the properties in the Parameter annotation, refer the API documentation. (https://developer.adroitlogic.org/project-x/api/17.07/apidocs/org/adroitlogic/x/annotation/config/Parameter.html)

After that we need to configure the logging annotations in-order to use the logging framework provided by the project-x framework. The most useful advantage of using this logging framework is that it will generate a unique code for each and every log line and it will easier to pin-point the any issue by analyzing the log file.

First, right click on the `com.acme.esb` package name directory and from the context menu, select `New → package-info.java`. Now a package-info file will be added. Next place the cursor one line before the package name and right click. Form the context menu select `Generate…` and you will be shown the code generation menu as shown in figure 6.4



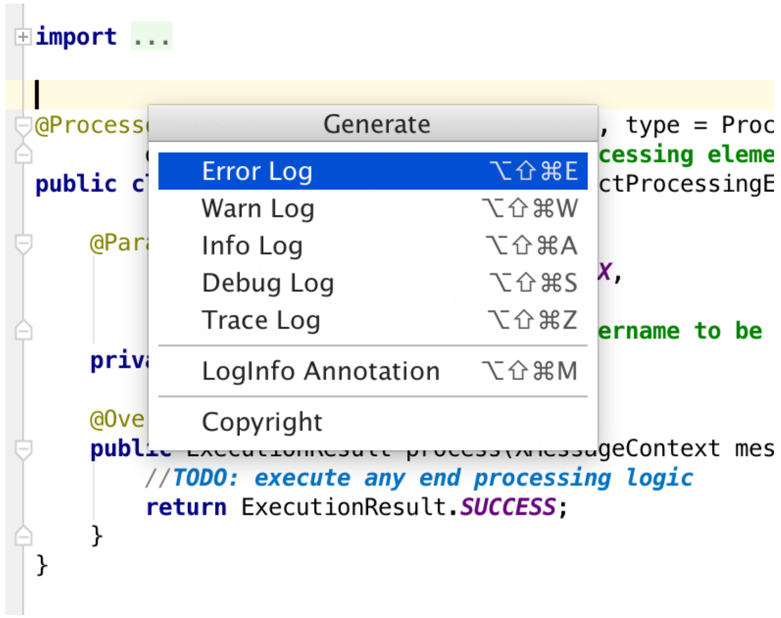
```
/**
 * @author Sajith Dilshan
 */

package c
```

Figure 6.4: Code generation menu

From that menu, select LogInfo Annotation. After that, you will be asked for a component type and for that specify `CTM`. Now your package-info file should look as below

```
@LogInfo(componentType = "CTM", moduleId = 0,
         nextModuleId = 1, maxModuleId = 1,
         nextLoggerId = 1)
package com.acme.esb;

import org.adroitlogic.x.logging.LogInfo;
```

Next, go to the Console Logger processing element and put the cursor one line before @Processor annotation and right click. From the context menu select `Generate…` and you will be shown the code generation menu as shown in figure 6.5



```
import ...

|
@Process      , type = Proc
              cessing eleme
public c      ctProcessingE

    @Par
              x,

              ername to be
    priv

    @Ove
    publ             geContext mes
        //TODO: execute any end processing logic
        return ExecutionResult.SUCCESS;
    }
}
```

Figure 6.5: Code generation menu

From that menu select LogInfo Annotation and a new @LogInfo annotation will be added to your class as below

```
@LogInfo(loggerId = 1, nextLogCode = 1)
```

Next, place the cursor inside the process() method and obtain the code generation menu as described previously. Now from that menu select `Info Log`. You can see that a logger statement is added to within the process() method and the `nextLogCode` value of the `@LogInfo` annotation has been incremented by one.

The logic in process() method is shown in Figure 6.6.

```
@Override
public ExecutionResult process(XMessageContext
                                     messageContext) {
    logger.info(1, "Hello {}", username);
    return ExecutionResult.SUCCESS;
}
```

Figure 6.6: Process() method logic

One more thing to note is that after executing our logic, we need to pass the message to the other processing element in the message flow. This step is optional. You can write an element which can be used as the last processing element of the flow as well. In-order to add a next element, you need to add a `XProcessingElement` class variable with `Outport` annotation as shown in figure 6.7

```
@OutPort(displayName = "Next Element",
        description = "Success output from the
                      ConsoleLogger")
private XProcessingElement nextElement;
```

Figure 6.6: Outport Annotation

The properties you need to modify in the outport annotation are

- ✦ **displayName** - The name to be displayed when user hover on the outport

- ✦ **description** - A brief description about under what circumstance an output will be emitted through this outport. This description will be shown in the documentation tab on the property pane

For more information on Outport annotation refer API documentation (https://developer.adroitlogic.org/project-x/api/17.07/apidocs/org/adroitlogic/x/annotation/config/OutPort.html)

Now we can modify the process method as below.

```
@Override
public ExecutionResult process(XMessageContext messageContext) {
    logger.info(1, "Hello {}", username);
    return nextElement.processMessage(messageContext);
}
```

Figure 6.7 shows the completed class code. Now Let's compile the project via `mvn clean install` command.

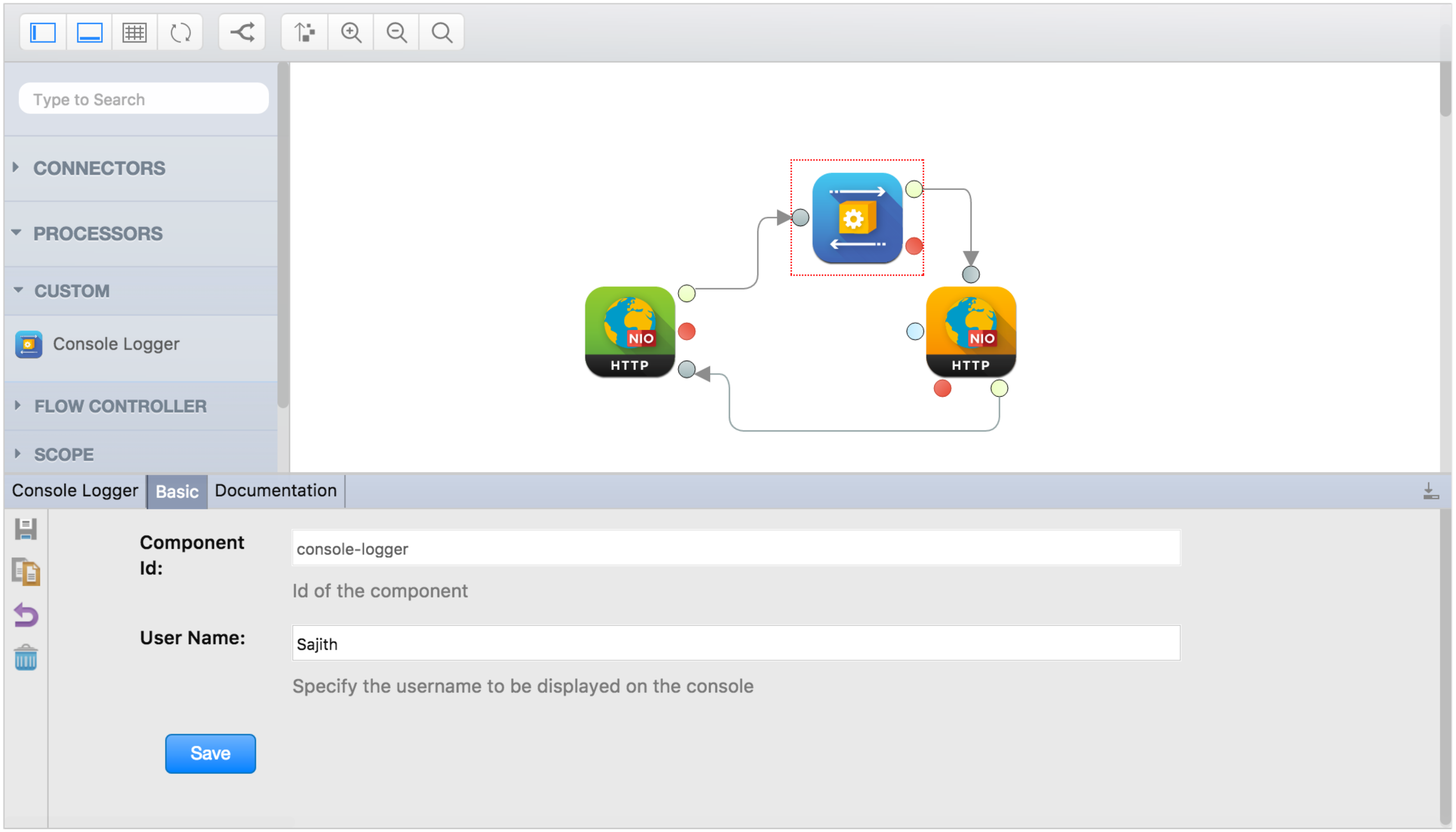
```
package com.acme.esb;

import org.adroitlogic.x.annotation.config.OutPort;
import org.adroitlogic.x.annotation.config.Parameter;
import org.adroitlogic.x.api.ExecutionResult;
import org.adroitlogic.x.api.XMessageContext;
import org.adroitlogic.x.annotation.config.Processor;
import org.adroitlogic.x.api.config.InputType;
import org.adroitlogic.x.api.config.ProcessorType;
import org.adroitlogic.x.api.processor.XProcessingElement;
import org.adroitlogic.x.base.processor.AbstractProcessingElement;

import org.adroitlogic.x.logging.LogInfo;

@LogInfo(loggerId = 1, nextLogCode = 2)
@Processor(displayName = "Console Logger", type = ProcessorType.CUSTOM,
           requireConfiguration = true,
           description = "Console Logger Processing element
                          adds a log line to the console")
public class ConsoleLogger extends AbstractProcessingElement {

    @OutPort(displayName = "Next Element",
            description = "Success output from the ConsoleLogger")
    private XProcessingElement nextElement;

    @Parameter(displayName = "User Name",
            inputType = InputType.TEXT_BOX,
            placeHolder = "Sajith",
            description = "Specify the username to be displayed
                           on the console")
    private String username;

    @Override
    public ExecutionResult process(XMessageContext messageContext) {
        logger.info(1, "Hello {}", username);
        return nextElement.processMessage(messageContext);
    }

    public XProcessingElement getNextElement() {
        return nextElement;
    }

    public void setNextElement(XProcessingElement nextElement) {
        this.nextElement = nextElement;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
}
```

Figure 6.7: Complete code of ConsoleLogger

After opening our helloWorldFlow (if the flow is already opened, click on the Refresh Current View button on the tool bar), in the component pallet, under processors/custom you can see our new processing element. When you add the new element to the design pane, you can see that the property pane is automatically opened and there is a property named User Name.

Specify your name and save the property and modify the flow as shown in figure 6.8

Now let's run the project and send a new message. If you inspect the console log, you can see our custom message is printed with the user specified property.



Component Id: console-logger

Id of the component

User Name: Sajith

Specify the username to be displayed on the console

Figure 6.8: Integration flow with Console Logger

Section 7

What is Project.xpml?

WHAT IS PROJECT.XPML?

In this section we will learn about the Project.xpml file and its content. From now on, I will refer to it as XPML file since there can be only one Project.xpml file within a single Ultra Project.

Basics of Project.xpml File

The XPML file is the beginning point of the project when it comes to the execution of the project. Sample XPML file is shown in Figure 7.1 and there are few key components in this file

✳ Project Id - This specifies the Id of the project (`id="mySampleProject`)

✳ Project Name - Name of the current Project (`name="mySampleProject"`)

✳ Project Version - Current version of the project (`version="1.0-SNAPSHOT"`)

✳ Description - A brief description about the current project

✳ Flows - This section contains the current Integration Flows in the project. Only the flows specified in this section will be deployed when the project is executed.

✳ Resources - This section contains spring beans which are used by the Integration Flows.

```
<x:project id="mySampleProject" name="mySampleProject"
version="1.0-SNAPSHOT"
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:x="http://www.adroitlogic.org/x/x-project"
xsi:schemaLocation="http://www.adroitlogic.org/x/x-project
http://schemas.adroitlogic.org/x/x-project-1.0.0.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.1.
xsd">

    <x:description>
     A sample project that demonstrates the projects concept
     </x:description>

    <x:flows>
        <x:flow id="sampleFlow" file="helloWorldFlow.xcml"/>
    </x:flows>

    <x:resources>
    </x:resources>

</x:project>
```

Figure 7.1: Sample project.xpml file

How to Add Resources

Under the resources section you can add Spring beans, Maps and Lists. There are keyboard shortcuts to add the boilerplate code.

Beans - Type `xbean` and press tab key and the code template for Spring bean resource will be injected to the XPML file.

Map - Type `xmap` and press tab key and the code template for a Map will be injected to the XPML file.

List - Type `xlist` and press tab key and the code template for a List will be injected to the XPML file.

Writing Custom Resource Templates

Suppose you want to write a custom processing element or a connector and within that component, a complex set of resources are used. In this case, the user has to add each resource one by one into the XPML file. This is an arduous and an unintuitive task. In-order to overcome this problem, you can write a resource template.

Suppose if I were to write a JMS ingress and egress connectors which uses ActiveMQ and in-order to use those connectors, following resources should be present in the XPML file

```
<x:resource id="activeMq-ConnectionFactory">
  <bean class="org.apache.activemq.ActiveMQConnectionFactory">
   <property name="brokerURL" value="tcp://localhost:61616"/>
  </bean>
</x:resource>
```

```
<x:resource id="activeMq-jmxTxnManager">
  <bean class=
  "org.springframework.jms.connection.JmsTransactionManager">
    <constructor-arg
       index="0"
       type="javax.jms.ConnectionFactory"
       ref="activeMq-springCachingConnectionFactory"/>
  </bean>
</x:resource>

<x:resource id="activeMq-ultraTxnmanager">
  <bean class=
"org.adroitlogic.x.base.trp.UltraPlatformTransactionManager">
   <property name="txnManager" ref="activeMq-jmxTxnManager"/>
  </bean>
</x:resource>


<x:resource id="activeMq-jmsTemplate">
  <bean class="org.springframework.jms.core.JmsTemplate">
     <constructor-arg
         index="0"
         type="javax.jms.ConnectionFactory"
         ref="activeMq-springCachingConnectionFactory"/>
    </bean>
</x:resource>

<x:resource id="activeMq-springCachingConnectionFactory">
  <bean class=
"org.springframework.jms.connection.CachingConnectionFactory">
     <constructor-arg index="0"
                type="javax.jms.ConnectionFactory"
                ref="activeMq-ConnectionFactory"/>
   </bean>
</x:resource>
```

Instead of asking the user to specify all the above resources, you could provide a template and obtain only the `brokerURL` from the user.

Figure 7.2 shows a sample Java class for the resource template. The `ResourceTemplate` annotation indicates that this class is a resource template and it will be picked up by the UltraStudio.

The class MUST implement the `XResourceTemplate` interface and override the `getXmlConfiguration()` method. This method will be invoked by UltraStudio to obtain the string which should be appended to the XPML file. Hence, this string MUST return a complete resource configuration.

If you need user input to build the resource configuration, as we did with custom processing elements, you can specify the inputs with `Parameter` annotation and add setters for the properties. UltraStudio will scan the class and show a model for the user to enter the inputs and the parameter annotated properties will be initialized with the user inputs prior to the invocation of `getXmlConfiguration()` method.

It is a **MUST** to add a parameter with the variable name **beanPrefix**. User can add a single resource template with multiple values to the project.xpml file. In that case all the resource Ids should be prefixed with the beanPrefix as shown in figure 7.2 (Complete code is available at https://gist.github.com/sajithdilshan/b0b4fab55bcc46c8795e9a1b9bcbdafd)

```java
import org.adroitlogic.x.annotation.config.Parameter;
import org.adroitlogic.x.api.config.InputType;
import org.adroitlogic.x.api.template.ResourceTemplate;
import org.adroitlogic.x.api.template.XResourceTemplate;

import java.util.UUID;

@ResourceTemplate(displayName = "ActiveMQ JMS")
public class ActiveMQJmsConfiguration implements XResourceTemplate {

    @Parameter(displayName = "Resource ID Prefix", description = "Specify the ID prefix
               for configuration resources",
               inputType = InputType.TEXT_BOX, placeHolder = "MQ1")
    private String beanPrefix;

    @Parameter(displayName = "Broker URL", description = "Specify the URL of the broker",
               inputType = InputType.TEXT_BOX,
               placeHolder = "tcp://localhost:61616", propertyName = "brokerURL")
    private String brokerURL = "localhost";

    @Parameter(displayName = "Username", description = "Specify the username for the
               broker", inputType = InputType.TEXT_BOX,
               isOptional = true, propertyName = "userName")
    private String userName;

    @Parameter(displayName = "Password", description = "Specify the password for the
               above mentioned user",
               inputType = InputType.TEXT_BOX, isOptional = true,
               propertyName = "password")
    private String password;

    @Override
    public String getXmlConfiguration() {
        if (beanPrefix == null) {
            beanPrefix = UUID.randomUUID().toString();
        }
        String newConfig = CONFIG.replaceAll("BEANPREFIX", beanPrefix);

        String auth = "";
        if (userName != null) {
            auth = String.format(AUTH, userName, password);
        }

        return String.format(newConfig, brokerURL, auth);
    }

    private static final String AUTH =
            "                <property name=\"userName\" value=\"%s\"/>\n" +
                    "                <property name=\"password\" value=\"%s\"/>\n";

    private static final String CONFIG = "\n" +
            "        <x:resource id=\"BEANPREFIX-activeMQ-ConnectionFactory\">\n" +
            "            <bean
class=\"org.apache.activemq.ActiveMQConnectionFactory\">\n" +
            "                <property name=\"brokerURL\" value=\"%s\"/>\n" +
            "%s" +
            "            </bean>\n" +
            "        </x:resource>\n" +
            "        <x:resource id=\"BEANPREFIX-activeMQ-jmxTxnManager\">\n" +
            "            <bean
class=\"org.springframework.jms.connection.JmsTransactionManager\">\n" +
            "                <constructor-arg index=\"0\"
type=\"javax.jms.ConnectionFactory\" ref=\"BEANPREFIX-activeMQ-
-springCachingConnectionFactory\"/>\n" +
            "            </bean>\n" +
            "        </x:resource>\n" +
            "        <x:resource id=\"BEANPREFIX-activeMQ-ultraTxnManager\">\n" +
            "            <bean
class=\"org.adroitlogic.x.base.trp.UltraPlatformTransactionManager\">\n" +
            "                <property name=\"txnManager\" ref=\"BEANPREFIX-activeMQ-
-jmxTxnManager\"/>\n" +
            "            </bean>\n" +
            "        </x:resource>\n" +
            "        <x:resource id=\"BEANPREFIX-activeMQ-jmsTemplate\">\n" +
            "            <bean class=\"org.springframework.jms.core.JmsTemplate\">\n" +
            "                <constructor-arg index=\"0\"
type=\"javax.jms.ConnectionFactory\" ref=\"BEANPREFIX-activeMQ-
-springCachingConnectionFactory\"/>\n" +
            "            </bean>\n" +
            "        </x:resource>\n" +
            "        <x:resource id=\"BEANPREFIX-activeMQ-
-springCachingConnectionFactory\">\n" +
            "            <bean
class=\"org.springframework.jms.connection.CachingConnectionFactory\">\n" +
            "                <constructor-arg index=\"0\"
type=\"javax.jms.ConnectionFactory\" ref=\"BEANPREFIX-activeMQ-ConnectionFactory\"/>\n" +
            "            </bean>\n" +
            "        </x:resource>";

    public String getBrokerURL() {
        return brokerURL;
    }

    public void setBrokerURL(String brokerURL) {
        this.brokerURL = brokerURL;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getBeanPrefix() {
        return beanPrefix;
    }

    public void setBeanPrefix(String beanPrefix) {
        this.beanPrefix = beanPrefix;
    }
}
```

Figure 7.2: Custom Resource Template Code

Adding a Resource From Template

After compiling the configuration template class, open the project.xpml file. Next right click within the XPML file and from the context menu select **Resource Template**.(Figure 7.3)

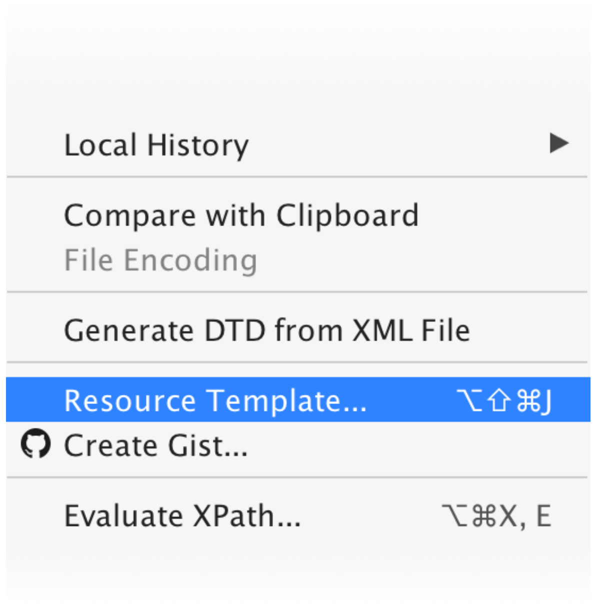
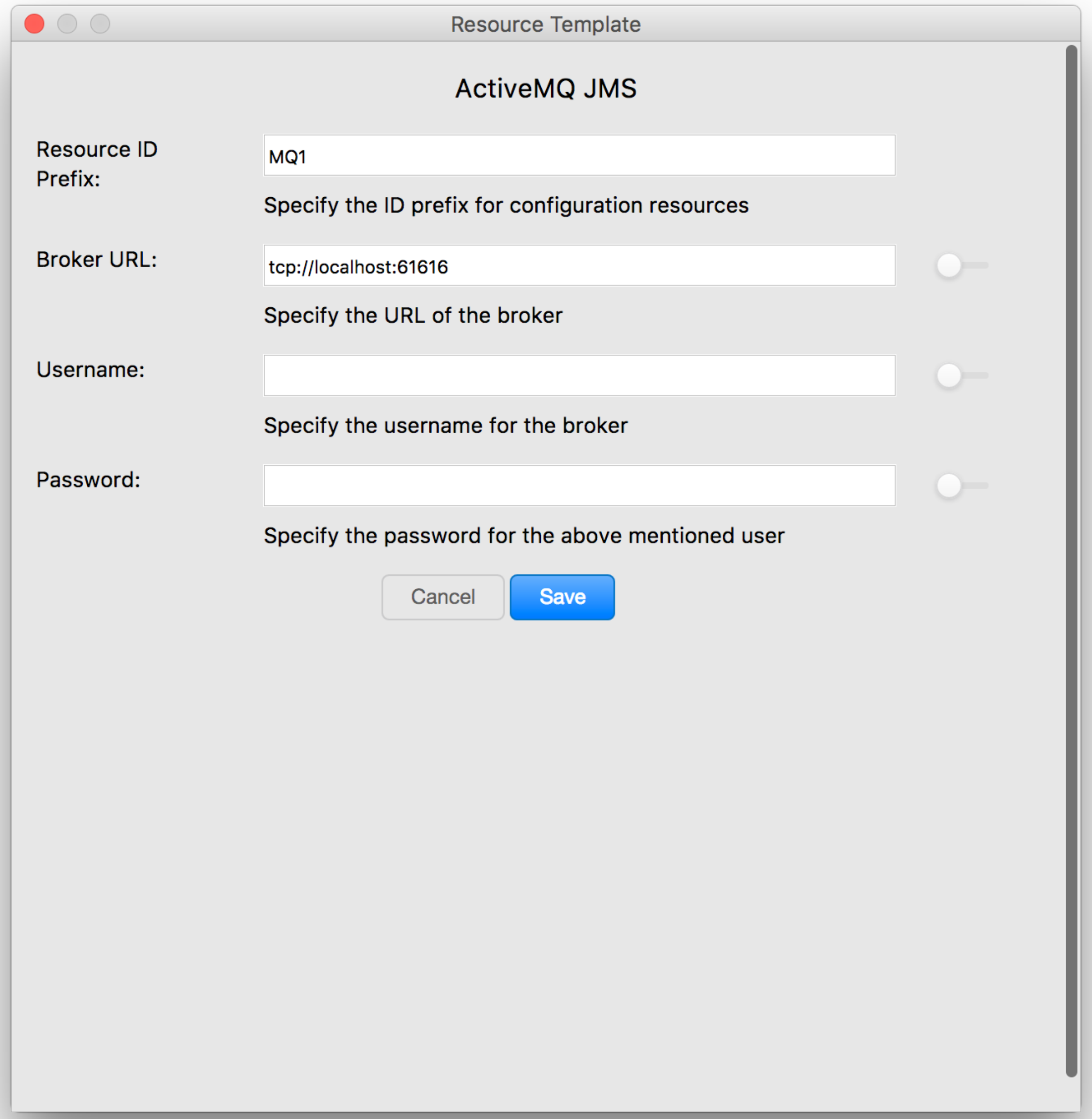


Figure 7.3: Context Menu

Next from the pop-up window, select the Resource template we have created (ActiveMQ JMS) and click continue button. Next specify the required properties in the modal dialog box (figure 7.4) and click on the save button to add the resource configuration into the project.xpml file.



Resource Template

ActiveMQ JMS

Resource ID Prefix: MQ1
Specify the ID prefix for configuration resources

Broker URL: tcp://localhost:61616
Specify the URL of the broker

Username:
Specify the username for the broker

Password:
Specify the password for the above mentioned user

Figure 7.4: Active MQ JMS Resource Template

Section 8

Miscellaneous Features

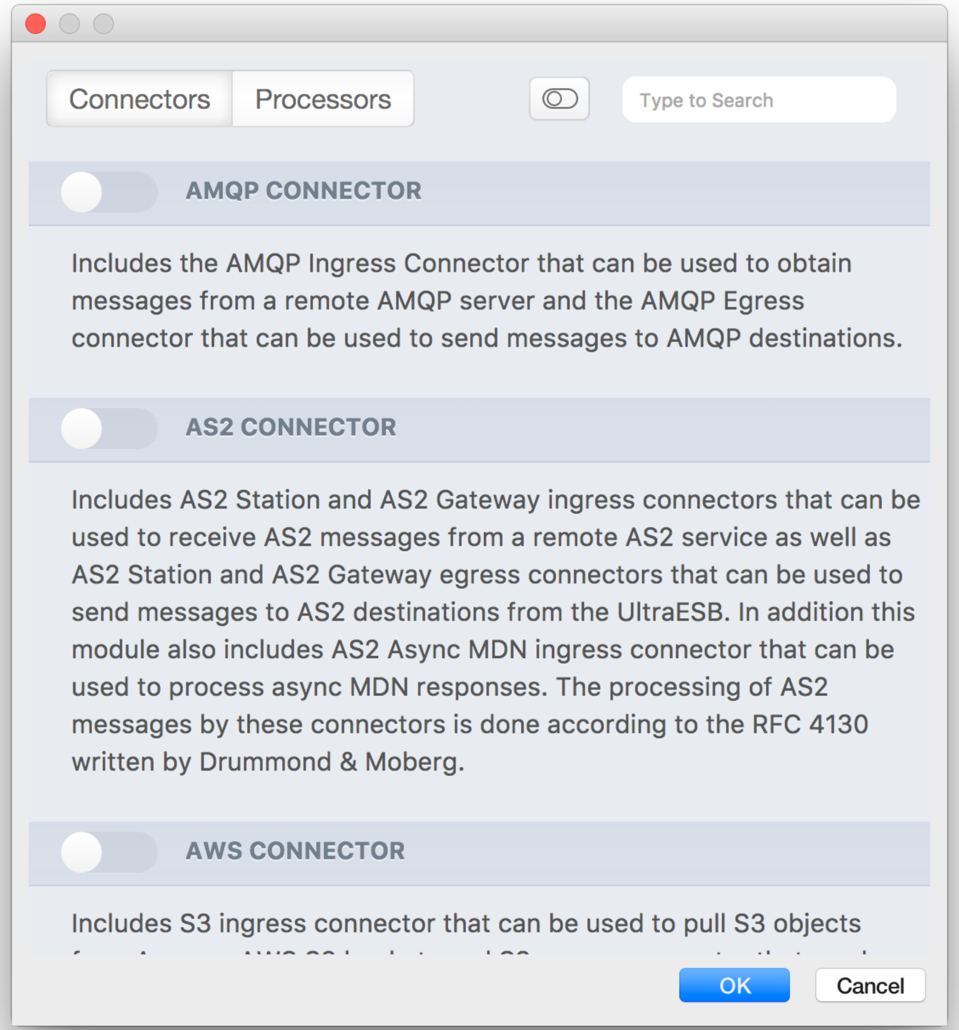
MISCELLANEOUS FEATURES

In this section we will look at few other components present in UltraStudio and those feature are

1. Component Registry
2. Password Encryptor
3. IPS Uploader

Component Registry

Component Registry can be used to add Processors and Connectors to the Integration Project after you have created the project. Component Registry can be opened via Tools → UltraStudio → Component Registry menu item. After that you will be shown a dialog box with the modules in the current project. Select the module you want to add new connectors or



AMQP CONNECTOR

Includes the AMQP Ingress Connector that can be used to obtain messages from a remote AMQP server and the AMQP Egress connector that can be used to send messages to AMQP destinations.

AS2 CONNECTOR

Includes AS2 Station and AS2 Gateway ingress connectors that can be used to receive AS2 messages from a remote AS2 service as well as AS2 Station and AS2 Gateway egress connectors that can be used to send messages to AS2 destinations from the UltraESB. In addition this module also includes AS2 Async MDN ingress connector that can be used to process async MDN responses. The processing of AS2 messages by these connectors is done according to the RFC 4130 written by Drummond & Moberg.

AWS CONNECTOR

Includes S3 ingress connector that can be used to pull S3 objects

Figure 8.1 Component Registry

processors and and click OK button. Next you will be shown the Component Registry as shown in Figure 8.1.

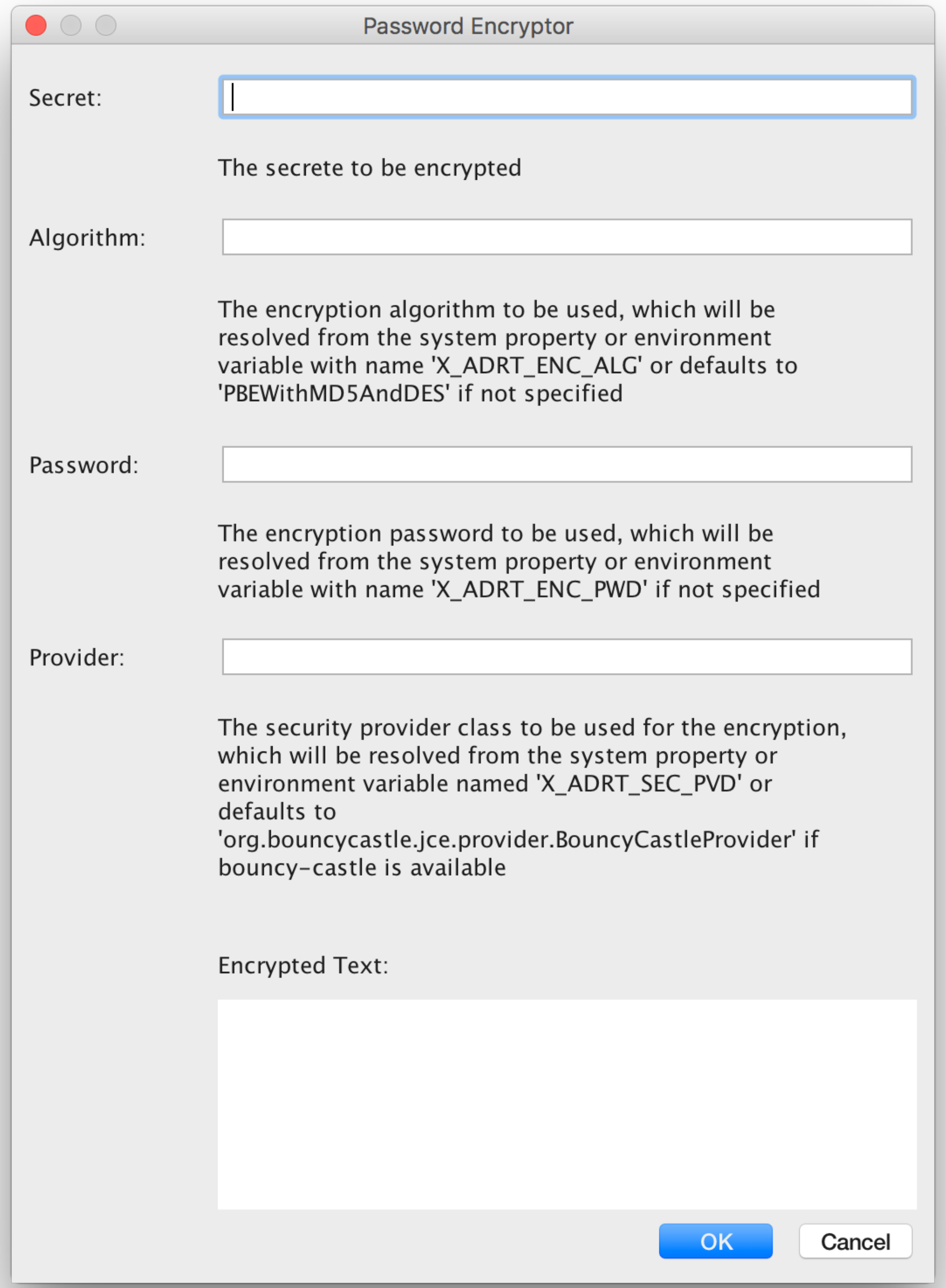
Note that there are two tabs (Connectors and Processors) in the Component Registry and from those tabs you can select the component you want to create your integration flow.

Password Encryptor

Password Encryptor can be accessed via Tools → UltraStudio → Password Encryptor menu item. In the dialog box there are couple of items you need to specify. These encrypted passwords can be directly used in your property configurations in integration flows.

★ **Secret** - Specify the secret you want to encrypt

★ **Algorithm** - Specify the algorithm you want to encrypt the secret. Note that the required Java library which includes the encryption algorithm mechanism MUST be included as an dependency in the pom.xml file. The default value for algorithm is **PBEWithMD5AndDES**. Further you can specify this algorithm via **X_ADRT_ENC_ALG** environment variable/System property as well



Secret:

The secrete to be encrypted

Algorithm:

The encryption algorithm to be used, which will be resolved from the system property or environment variable with name 'X_ADRT_ENC_ALG' or defaults to 'PBEWithMD5AndDES' if not specified

Password:

The encryption password to be used, which will be resolved from the system property or environment variable with name 'X_ADRT_ENC_PWD' if not specified

Provider:

The security provider class to be used for the encryption, which will be resolved from the system property or environment variable named 'X_ADRT_SEC_PVD' or defaults to 'org.bouncycastle.jce.provider.BouncyCastleProvider' if bouncy-castle is available

Encrypted Text:

Figure 8.2 Password Encryptor

★ **Password** - The password which is used to encrypt the secret. You can use **X_ADRT_ENC_PWD** environment variable/System property to specify this password as well.

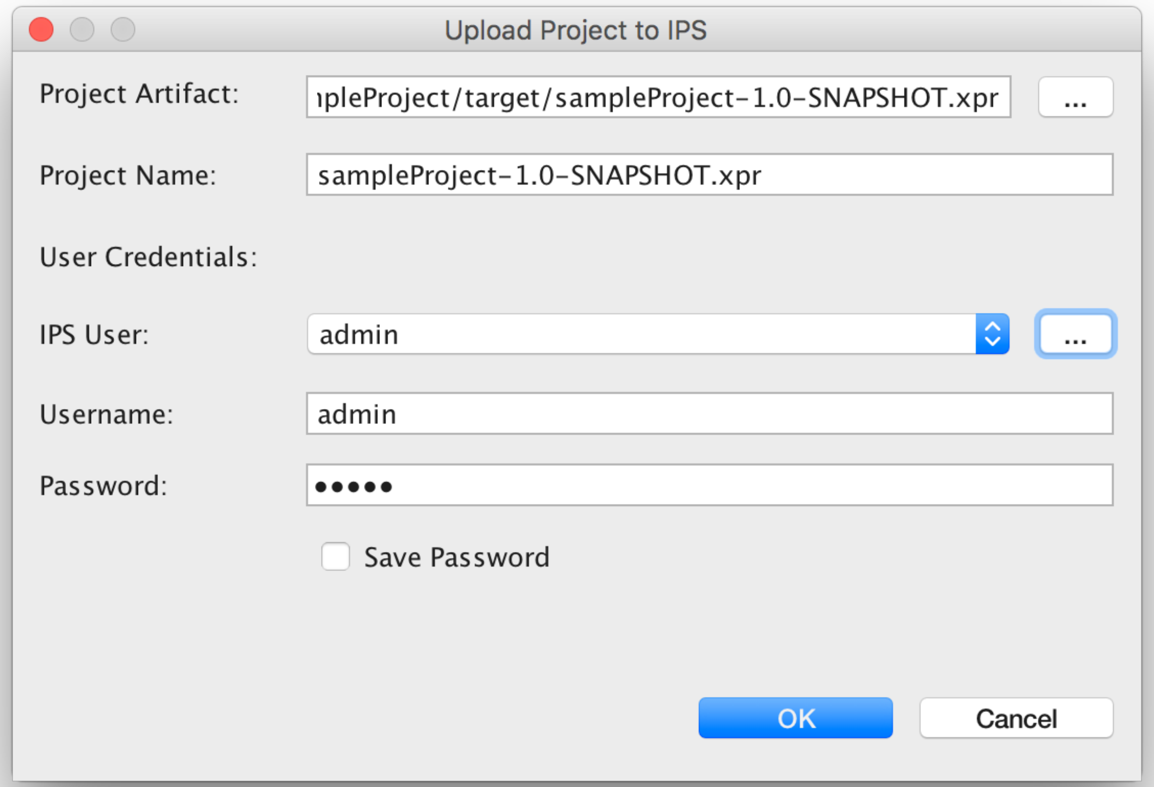
★ **Provider** - You need to specify the security provider in here. You can specify this value via **X_ADRT_SEC_PVD** environment variable/System property. The default value is **org.bouncycastle.jce.provider.BouncyCastleProvider.**

IPS Uploader

AdroitLogic Integration Platform as a Service (IPS) defines a new and convenient way to easily integrate your systems with minimum setup time, low maintenance cost and hassle free configuration updates. This is a stack designed to deploy, manage and monitor Adroitlogic's high performance UltraESB-X instances on a public or private cloud environments.

UltraStudio provided direct integration with IPS and all you have to do is specify the URL of the hosted IPS instance in the UltraStudio Preference.

First compile the project and build the .xpr bundle. After that click on Tools → UltraStudio → IPS Uploader menu item. Next, select the Project Artifact and specify the IPS user credentials (Figure 8.3.


Upload Project to IPS

Project Artifact: ıpleProject/target/sampleProject-1.0-SNAPSHOT.xpr

Project Name: sampleProject-1.0-SNAPSHOT.xpr

User Credentials:

IPS User: admin

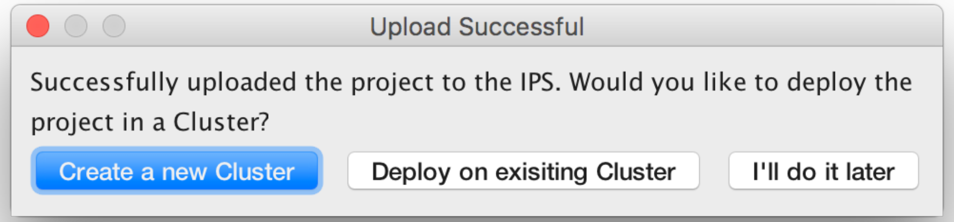
Username: admin

Password: •••••

Save Password

Figure 8.3 Project Uploader

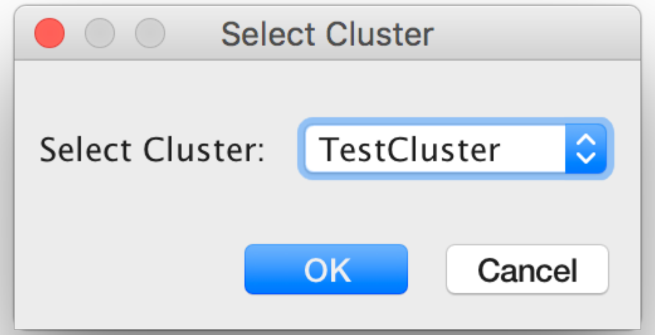
After specifying the above mentioned properties click on OK button and project will be uploaded into IPS. After the project is uploaded to IPS successfully, you will be given the option to deploy the project on a existing cluster or on a new cluster (Figure 8.4)


Upload Successful

Successfully uploaded the project to the IPS. Would you like to deploy the project in a Cluster?

Figure 8.4 Project upload message

If you want to deploy the project on an existing cluster, select the **Deploy on existing Cluster** option. After that you will be shown a list of existing clusters in the IPS instance (Figure 8.5). Select the cluster you want to deploy the project and click OK.


Select Cluster

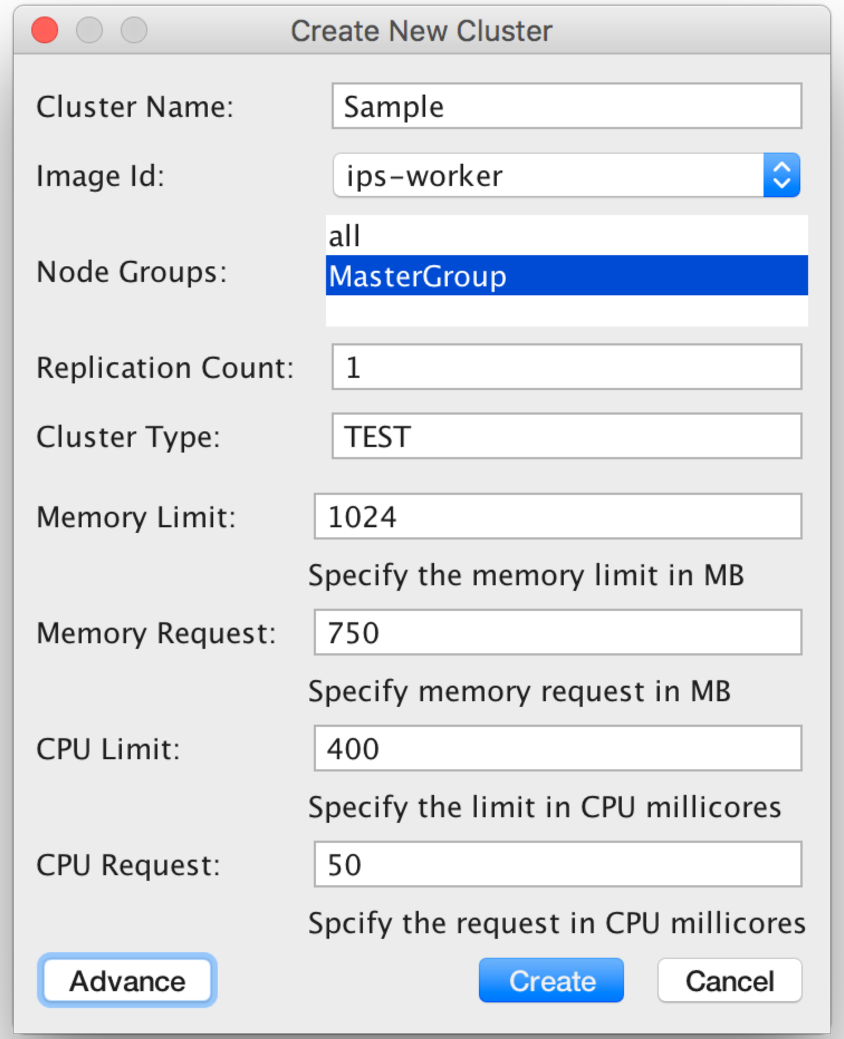
Select Cluster: TestCluster

Figure 8.5 Cluster Selection

After that you will get a message similar to the below one.

```
Uploaded the project and deployed on cluster. Server responded with: Cluster version deployment started successfully
```

On the other hand, if you want to deploy the project on a new cluster select **Create a new Cluster** from figure 8.4. In the new cluster creation dialog box specify Cluster Name, select an Image Id, Node Group as well. Specify the replication count and a Cluster Type as well. All the other properties are Advanced options.



Cluster Name: Sample

Image Id: ips-worker

Node Groups: all, MasterGroup

Replication Count: 1

Cluster Type: TEST

Memory Limit: 1024

Specify the memory limit in MB

Memory Request: 750

Specify memory request in MB

CPU Limit: 400

Specify the limit in CPU millicores

CPU Request: 50

Spcify the request in CPU millicores

Figure 8.5 Create New Cluster

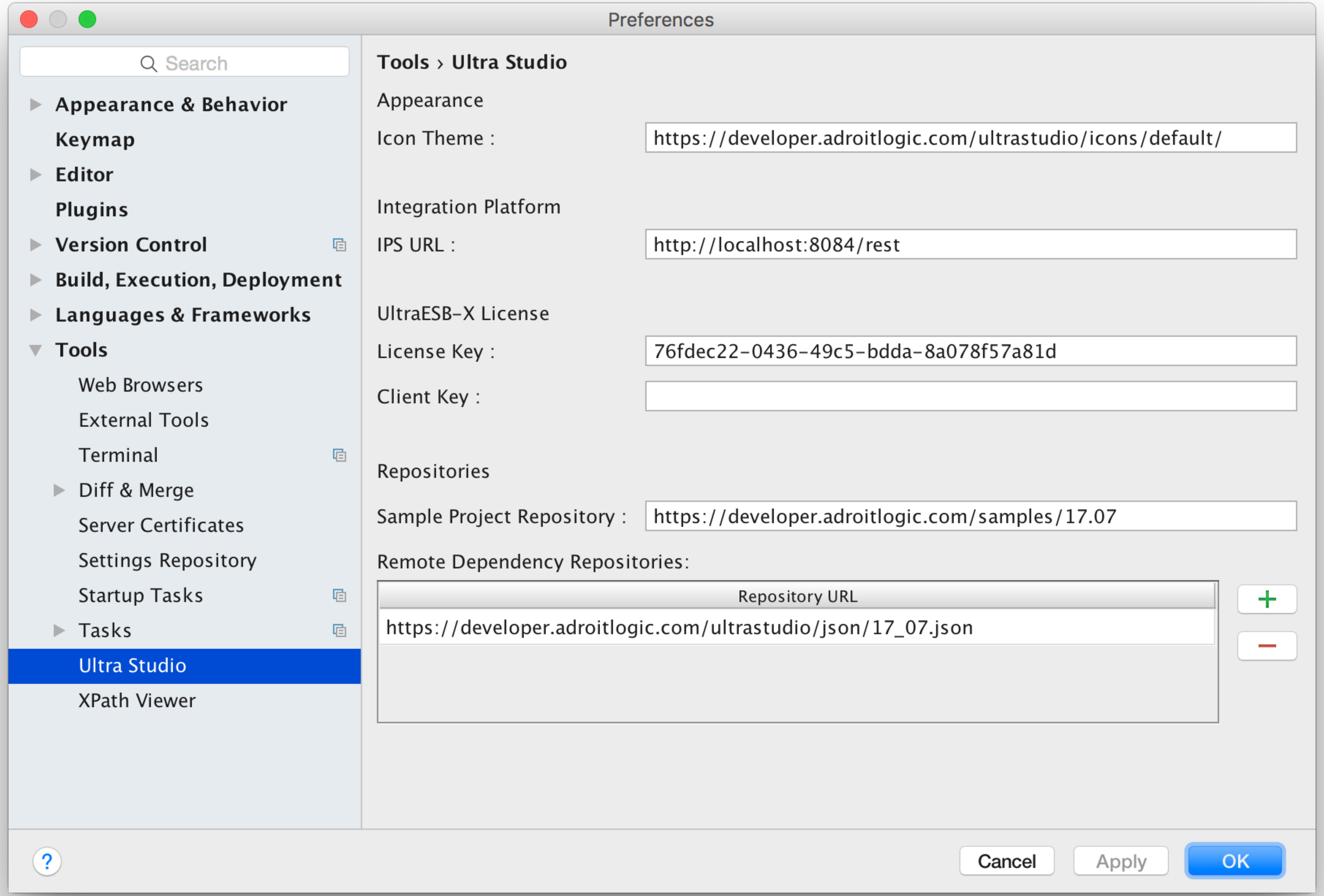
After specifying the above mentioned properties, and clicking on OK, you will get a message as below upon successful cluster deployment.

```
Created the cluster and uploaded the project. Server
responded with : Cluster version deployment started
successfully
```

Section 9

UltraStudio Preferences

UltraStudio provides the functionality to modify the UI of Editor pane as well as various other functionalities. This section elaborates on the Preferences provided by UltraStudio which can be modified by the end user.



Tools › Ultra Studio

Appearance

Icon Theme : https://developer.adroitlogic.com/ultrastudio/icons/default/

Integration Platform

IPS URL : http://localhost:8084/rest

UltraESB-X License

License Key : 76fdec22-0436-49c5-bdda-8a078f57a81d

Client Key :

Repositories

Sample Project Repository : https://developer.adroitlogic.com/samples/17.07

Remote Dependency Repositories:

Repository URL
https://developer.adroitlogic.com/ultrastudio/json/17_07.json

Figure 9.1: UltraStudio Settings Panel

✦ Icon Theme

Icon theme to be used for connectors and processors.

✦ IPS URL

UltraStudio provides integration with IPS and you can directly upload a project you have created in UltraStudio into IPS. For this property, you need to specify the URL of the hosted IPS instance.

✦ UltraESB-X License Key

You can start an UltraESB-X instance within UltraStudio and deploy, test the project you have developed. But in-order to do that, you need to obtain a license from the AdroitLogic to perpetually run the UltraESB-X runtime. After obtaining the license, specify it here. By default, a 30 day evaluation license is specified.

✦ UltraESB-X Client Key

A Client key is required to activate the license when running the UltraESB-X instance. The client key is sent to the user via an email when UltraStudio is downloaded. A new client key can be obtained by dropping an email to license@adroitlogic.com

✦ Sample Project Repository

Specify the URL of the Sample UltraProject repository. Default value is https://developer.adroitlogic.org/samples/17.07.

✦ Remote Dependency Repository

Specify the URL of the JSON file which contains the information of processors and connectors. The default value is https://developer.adroitlogic.com/ultrastudio/json/17_07.json.

Section 10

UltraStudio Toolbox

ULTRASTUDIO TOOLBOX

UltraStudio Comes with a number of utility components which helps the developer to test, debug integration flows. Mainly there are six components which can be used to ease your Integration Flow development tasks and those are

1. HTTP/S Client
2. Socket Client
3. JMS Client
4. JMS Receiver/Browser
5. Jetty Server
6. TCP Dump Tool

HTTP/S Client

Features and Capabilities

The HTTP/S client panel provides a full RFC 2616 compliant HTTP/S client based on the Apache HttpComponents/HttpClient project. The HTTP/S client panel is capable of

✦ Sending HTTP/S requests with POST, GET, PUT, DELETE, HEAD, OPTIONS & TRACE methods

✦ Supporting 2-way SSL, ignoring SSL, ignoring host name verification

✦ Validation and support BASIC, DIGEST or NTLM authentication

✦ Setting preset payloads

✦ Setting payload by typing in or selecting a payload file

✦ Using binary payloads - such as raw Hessian messages

✦ Sending requests using both HTTP 1.0 and 1.1 version

✦ Controlling the use of chunking, expectation control (i.e. waiting for a 100 continue reply before sending the body), keep-alive and response compression control (i.e. accepting Gzipped responses).

✦ Setting the socket timeout to allow the client to specify the maximum delay for a response to arrive.

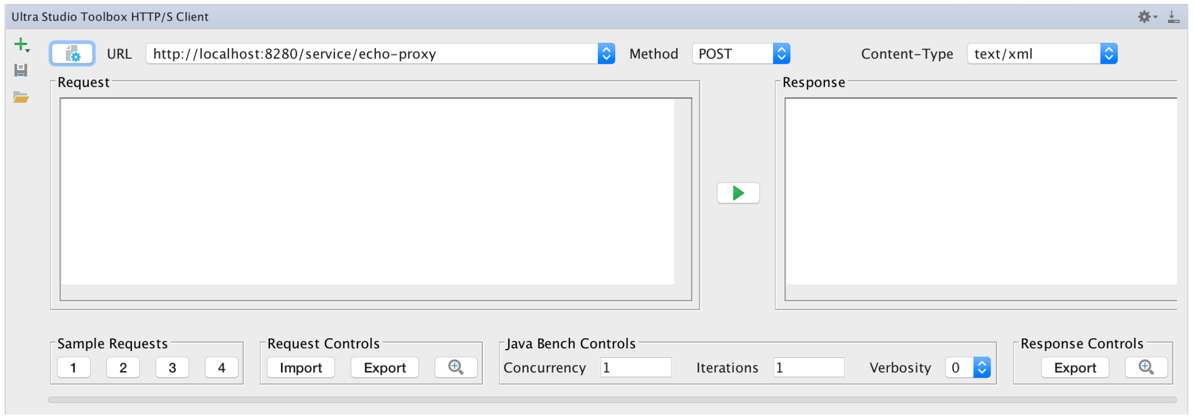
✦ Sending SOAP requests with custom SOAPActions

✦ Setting HTTP headers

✦ Unzipping a gzipped response payload for read.

✦ Sending a concurrent load specifying the concurrency level and number of iterations

Configuration Parameters

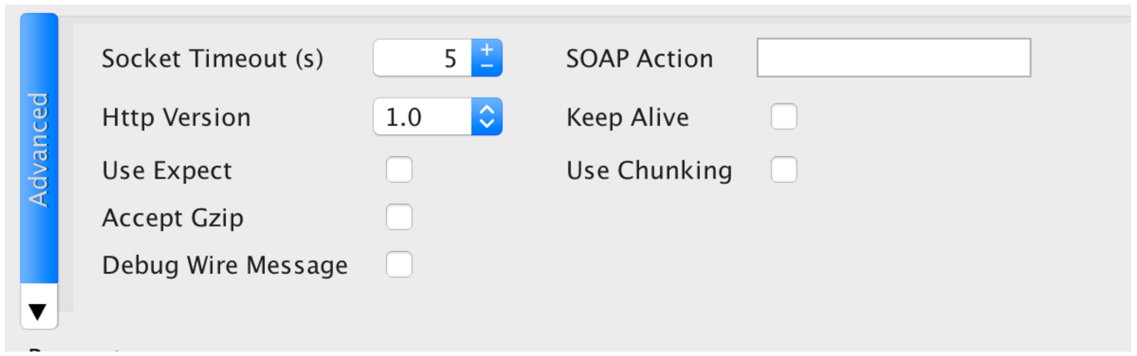


URL http://localhost:8280/service/echo-proxy
Method POST
Content-Type text/xml
Concurrency 1
Iterations 1
Verbosity 0

Configuration	Description
URL	A well formed URL which the request should send to. Once a new URL is entered it will be saved and can be selected from the drop down menu at the end of the URL field.
Method	Select one of POST, GET, PUT, DELETE, HEAD, OPTIONS & TRACE from the drop down menu to set as the requests HTTP method
Content Type	Select one of the given content type or a user can enter their own type, which will be set as Http header "Content-Type" for requests that has a payload

Request	Request payload to send with the request. This field becomes not accessible for HTTP methods that don't expect a payload (e.g : **GET**)
Sample Requests	A user can easily set a one of four preset payloads to send a test message
Request Controls	A request payload can be imported from a file, export to a file and view in full screen
Java Bench Controls	These input fields configure the load testing parameters. 1. Concurrency : Number of concurrent threads to be used when sending requests 2. Iterations : Number of requests send per a thread 3. Verbosity : Verbosity level of the output
Response Controls	A response message can be export to a file and view in full screen. If gzip compression is enabled, then only unzip payload checkbox is visible. If Unzip Checkbox is selected the response payload field will display the extracted payload.

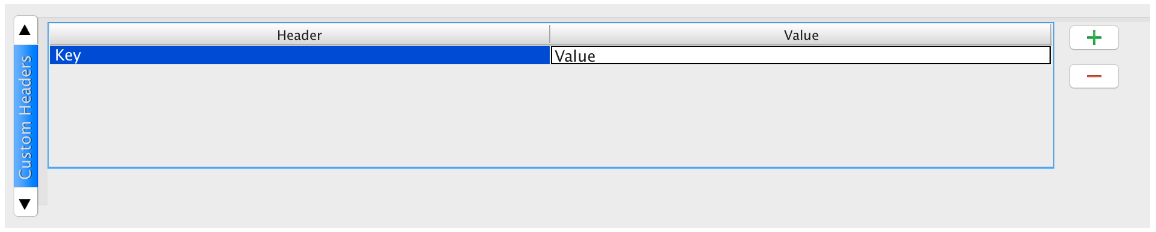
Advanced Configuration



Socket Timeout (s)	5	SOAP Action	
Http Version	1.0	Keep Alive	☐
Use Expect	☐	Use Chunking	☐
Accept Gzip	☐		
Debug Wire Message	☐		

Configuration	Description
Socket Timeout	Amount of time in seconds that the client waits for a response before time out
Http Version	A user can switch between Http 1.0 and Http 1.1 by choosing the relevant value
Use Expect	Select whether to use expect-100 header to indicate the capability to the server
Accept Gzip	Select whether client supports gzip for the response payload or not
SOAP Action	Set custom soap action if required
Keep Alive	Use Connection keep alive at the clients side
Use Chuncking	Use HTTP payload chunking for the client
DEBUG SSL Engine	Enable SSL level debugging, note that this could be set only before executing a request for the first time.

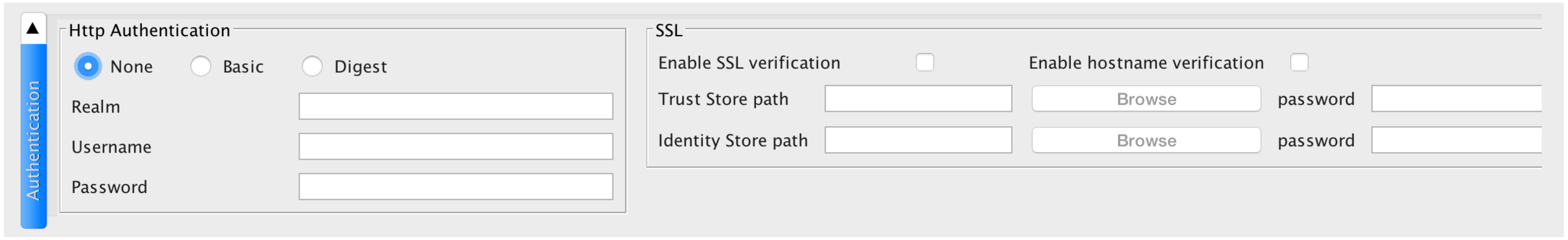
Custom Header Configuration



Header	Value
Key	Value

Header configuration can be added and removed from the table as header name header value pairs.

Authentication Configuration


Authentication

Http Authentication

[✓ None] Basic Digest

Realm	
Username	
Password	

SSL

Enable SSL verification	☐	Enable hostname verification	☐	
Trust Store path		Browse	password	
Identity Store path		Browse	password	

Authentication Configuration consist of two parts; HTTP Authentication Configuration and SSL Configuration

1. Configuration for HTTP Authentication

User can select one of Basic, Digest and NTLM types and provide required authentication details.

2. SSL Configuration

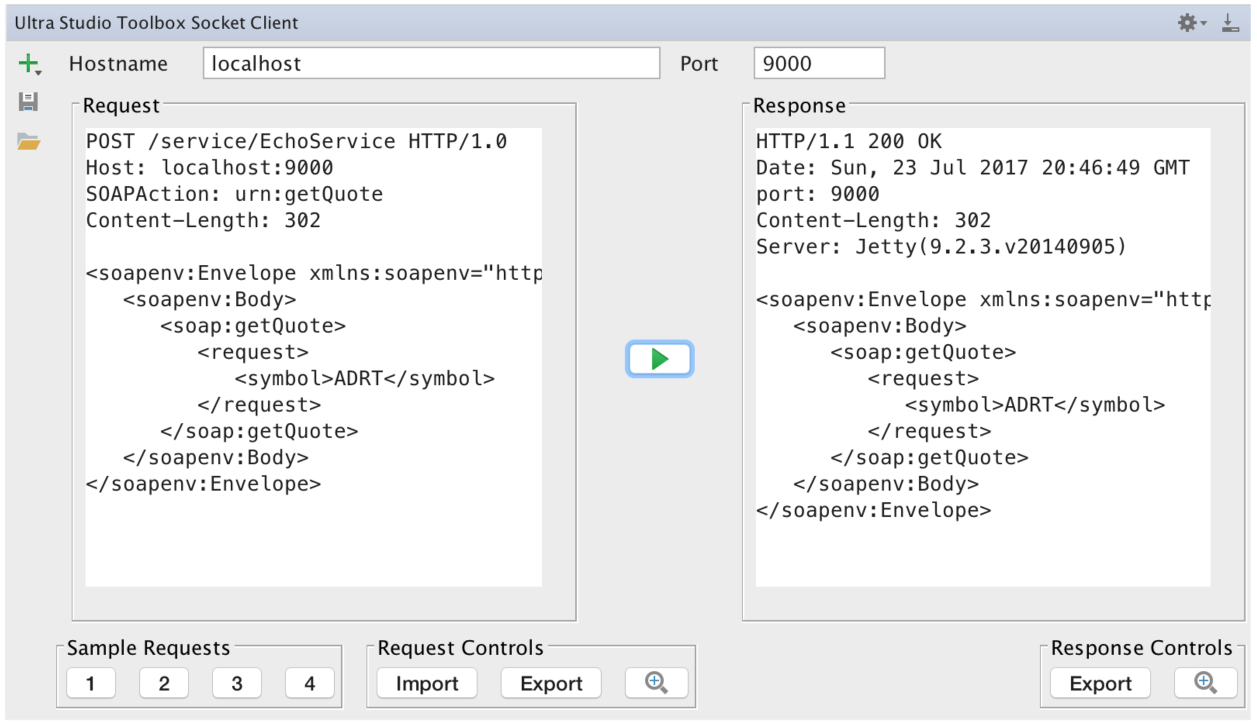
Even though the client talks to a HTTPS endpoint, HTTPS client can disable SSL verification for testing purposes. SSL verification will only work when **Enable SSL Verification** check box is selected. When SSL verification is enabled a user can optionally select whether to verify the server host name or not by selecting **Enable hostname verification** check box. When SSL verification is enabled, Trust store and Identity store information should be provided.

Socket Client

Features and Capabilities

The raw socket allows the user to send valid, invalid, malformed, corrupted or malicious payloads to the selected host over to the selected port. This is useful to test the negative test scenarios which are otherwise difficult to identify, reproduce and fix.

Some service hosting engines and ESBs may not be able to even accept malformed requests, even for logging purposes. Note that this is an advanced option and expects the user to be familiar with the HTTP RFC 2616 for operation. The most common mistake a novice user usually does is specifying a wrong content length - which should be the size of the payload in bytes. The payload must be separated from the header by a blank line as per the specification.



```
POST /service/EchoService HTTP/1.0
Host: localhost:9000
SOAPAction: urn:getQuote
Content-Length: 302

<soapenv:Envelope xmlns:soapenv="http
   <soapenv:Body>
      <soap:getQuote>
         <request>
            <symbol>ADRT</symbol>
         </request>
      </soap:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

```
HTTP/1.1 200 OK
Date: Sun, 23 Jul 2017 20:46:49 GMT
port: 9000
Content-Length: 302
Server: Jetty(9.2.3.v20140905)

<soapenv:Envelope xmlns:soapenv="http
   <soapenv:Body>
      <soap:getQuote>
         <request>
            <symbol>ADRT</symbol>
         </request>
      </soap:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

Configuration

The configuration is rather simple. **Hostname** and **Port** should be the hostname and port number of the server. There are four preset sample requests available in Sample requests panel. **Request control** allows to a user to import requests from a file as well as export the current requests to a file while **Response Control** allows a user to export the response from the server to a file.

JMS Client

Features and Capabilities

The JMS client panel provides a JMS client utility which can send JMS messages to ActiveMQ, HornetMQ and IBMMQ JMS queues. The JMS client panel is capable of

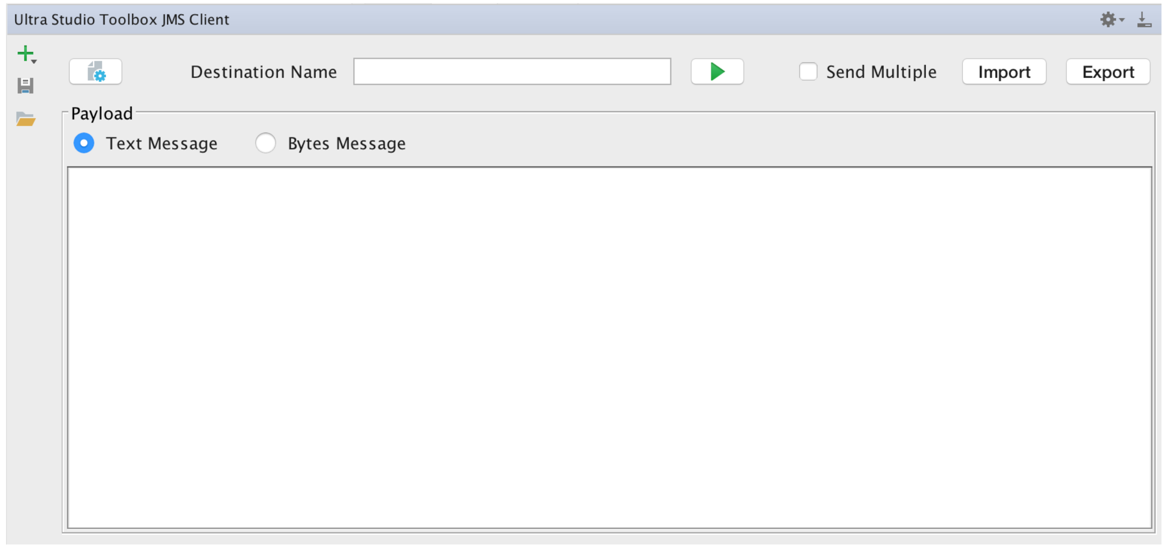
• Sending Text or Byte JMS messages

• Adding different type of JMS message headers

• Setting various JMS message properties including Correlation ID, Message ID, Expiration, Time Stamp, Priority etc.

• Importing requests from file locations

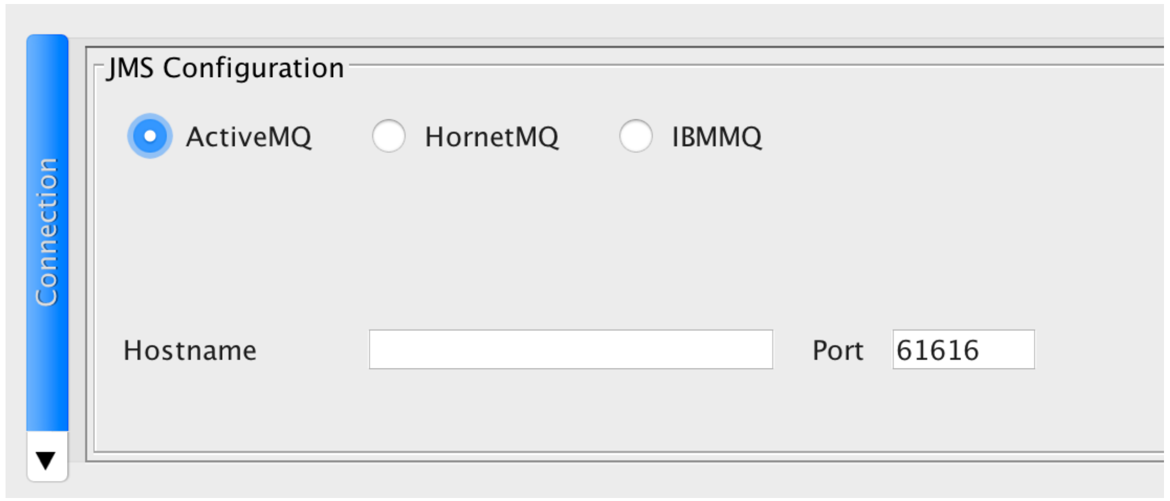
Main Configuration



Payload: [✓ Text Message] Bytes Message

Configuration	Description
Destination Name	Destination queue name that the message should sent to.
Import and Export	Import file content to be sent out or export content in the payload field.
Payload	A user can set either Text Message or Byte Message as the request payload.

Connection Configuration



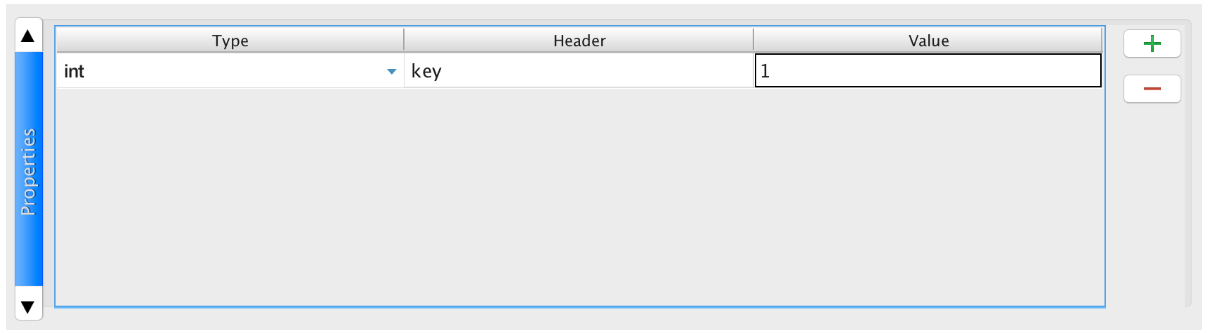
[✓ ActiveMQ] HornetMQ IBMMQ

Port 61616

A user can set the type of JMS server from this configuration panel. For Active MQ and HornetMQ servers **hostname** of the server and **port** number should be provided as the connection configuration where IBM MQ required two additional configurations, **Channel** and **Queue Manager**.

For the case of IBM MQ, the IBM MQ client libraries should be added using the file picker at the right most side of the panel since there is a license limitation to ship those libraries.

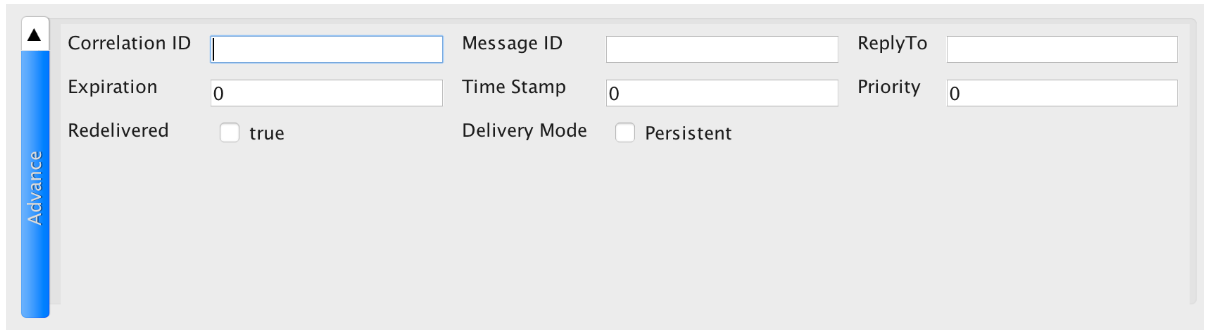
JMS Header Configuration



Type	Header	Value
int	key	1

Header configuration can be added and removed from the table as header name, header value pairs with the header type. Integer, Long, Boolean and String type headers can be added.

Advanced Configuration



Correlation ID
Message ID
ReplyTo
Expiration 0
Time Stamp 0
Priority 0
Redelivered ☐ true
Delivery Mode ☐ Persistent

Configuration	Description
Correlation ID	The JMSCorrelationID header field is used for linking one message with another. It typically links a reply message with its requesting message
Message ID	The JMSMessageID header field contains a value that uniquely identifies each message sent by a provider.
Reply To	Destination to which to send a response to this message
Expiration	The time the message expires, which is the sum of the time-to-live value specified by the client and the GMT at the time of the send
Time Stamp	The JMSTimestamp header field contains the time a message was handed off to a provider to be sent. It is not the time the message was actually transmitted, because the actual send may occur later due to transactions or other client-side queueing of messages.
Priority	The JMS API defines ten levels of priority value, with 0 as the lowest priority and 9 as the highest. In addition, clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority
Redelivered	If set Specifies whether this message is being redelivered. This field is set at the time the message is delivered.
Delivery mode	If selected sets the DeliveryMode value for this message as Persistent, else set it as non persistence

JMS Receiver/Browser

Features and Capabilities

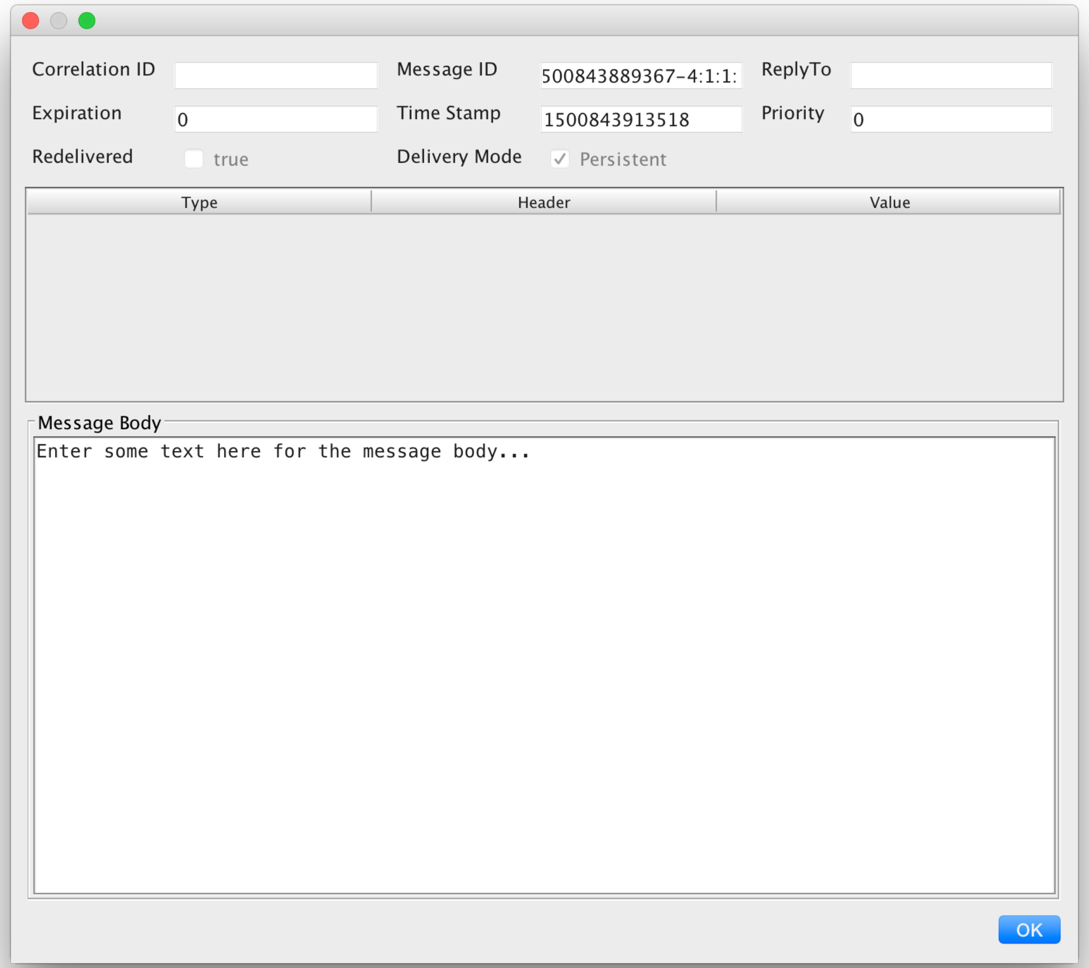
The JMS Receiver/Browser panel provides a JMS client utility which can obtain/browse JMS messages from ActiveMQ, HornetMQ and IBMMQ JMS queues. The JMS Receiver/Browser panel is capable of

• Receiving Text or Byte JMS messages

• Showing content of individual messages

The difference between **Receive** and **Browse** modes is that in Receive mode, the message will be removed from the source queue once it is fetched by the JMS client panel. In the Browse mode, the original message will not be removed from the source queue, and the user can view the content of the message from the JMS client panel.

The configuration is similar to the JMS Client and once a message has been obtained/browsed from the queue, you can view the content of the message by double clicking on it



Correlation ID		Message ID	500843889367-4:1:1:	ReplyTo	
Expiration	0	Time Stamp	1500843913518	Priority	0
Redelivered	true	Delivery Mode	✓ Persistent		

Type	Header	Value

Message Body

```
Enter some text here for the message body...
```

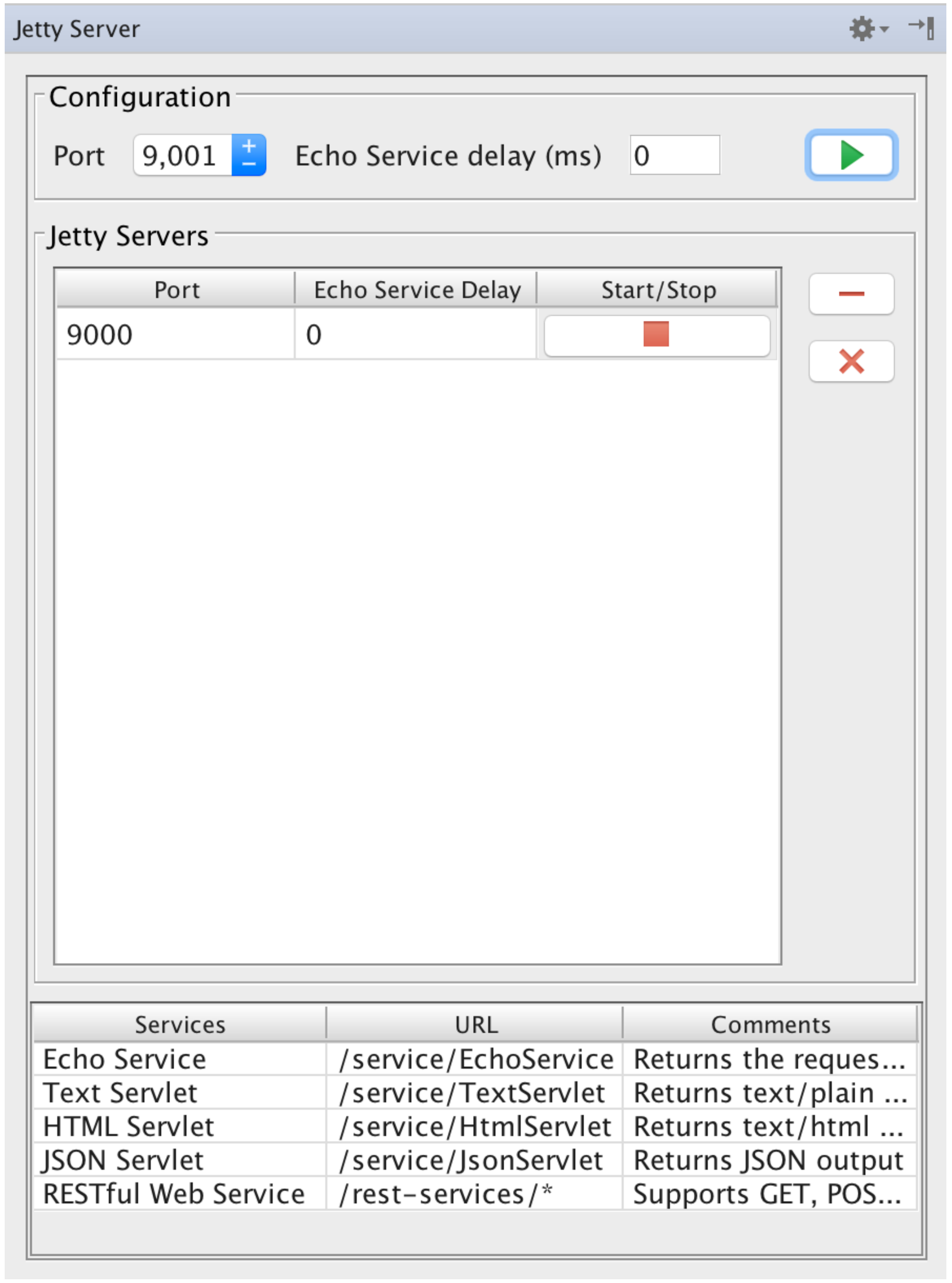
Jetty Server

Features and Capabilities

The Jetty server contains an implementation of the SimpleStockQuoteService (copied with modifications from the Apache Synapse project). And EchoService which is a high performance service which will echo back the request received. Thus it can be used to echo back various message sizes - say 1K, 5K, 10K etc for load and performance testing. Specifying an Echo service delay, causes the responses to be delayed by the specified number of milliseconds. This is useful to analyze how an ESB not using a Non-Blocking IO approach can easily block, when handling a large number of concurrent connections - which must be kept open until the delayed responses are received. The sample service also contains a servlet that will output a plain text response and an HTML response, to analyze how the UltraESB can handle these types of responses.

Configuration

The configuration is as simple as it could be, just specify the port number which the server should run on and hit the start button. Additionally you can select **Echo service delay** in **Milli seconds.** You can see the started servers in the below Jetty Servers panel where you can stop/start again or change the delay as you desired. Controls on your right allows you to stop and remove servers one by one or all at once.



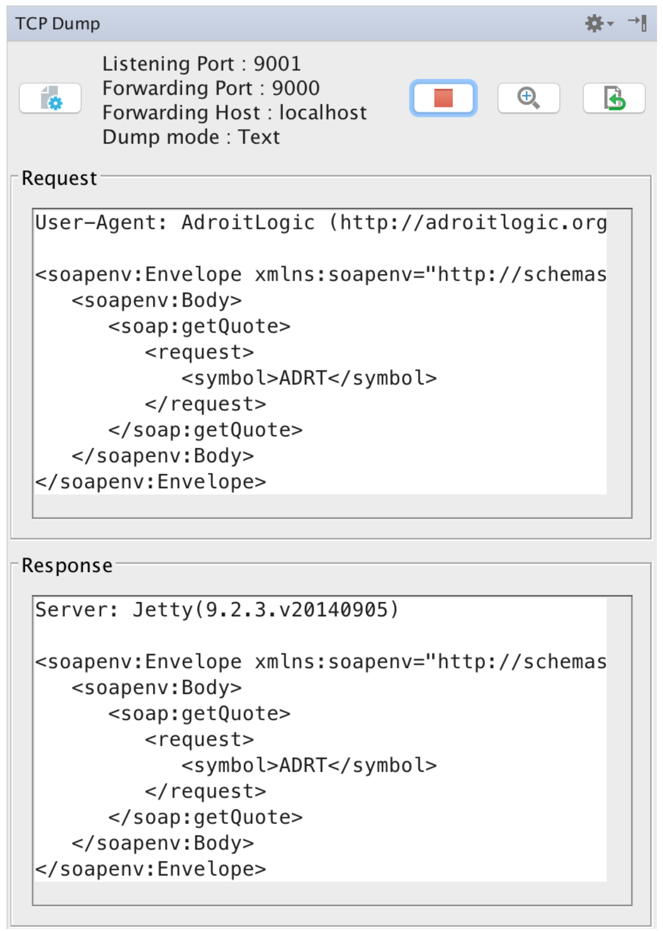
Port 9,001 Echo Service delay (ms) 0

Port	Echo Service Delay	Start/Stop
9000	0	🟥

Services	URL	Comments
Echo Service	/service/EchoService	Returns the reques...
Text Servlet	/service/TextServlet	Returns text/plain ...
HTML Servlet	/service/HtmlServlet	Returns text/html ...
JSON Servlet	/service/JsonServlet	Returns JSON output
RESTful Web Service	/rest-services/*	Supports GET, POS...

TCP Dump Tool

Features and Capabilities



TCP Dump

Listening Port : 9001
Forwarding Port : 9000
Forwarding Host : localhost
Dump mode : Text

Request

```
User-Agent: AdroitLogic (http://adroitlogic.org

<soapenv:Envelope xmlns:soapenv="http://schemas
   <soapenv:Body>
      <soap:getQuote>
         <request>
            <symbol>ADRT</symbol>
         </request>
      </soap:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

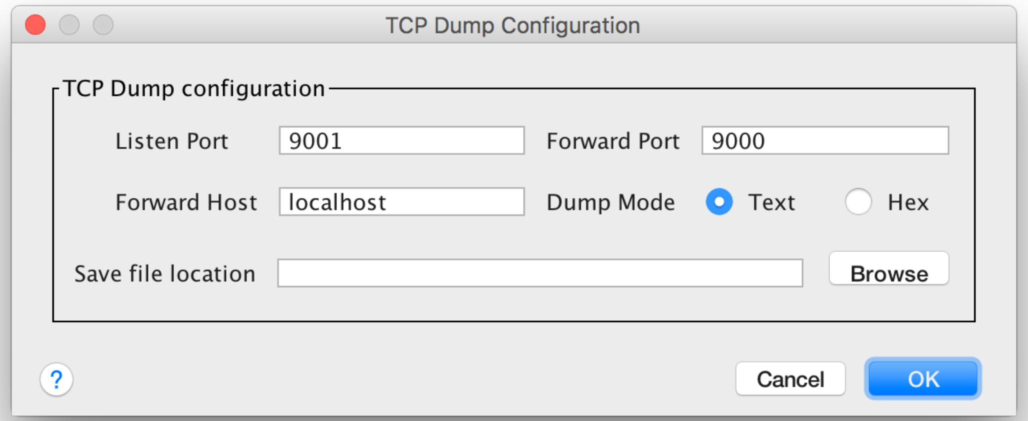
Response

```
Server: Jetty(9.2.3.v20140905)

<soapenv:Envelope xmlns:soapenv="http://schemas
   <soapenv:Body>
      <soap:getQuote>
         <request>
            <symbol>ADRT</symbol>
         </request>
      </soap:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

The TCP Dump Tool allows one to easily monitor, and optionally capture the messages sent at the wire level for debugging, or capturing (e.g. for a subsequent load test) purposes. The messages can be captured as Text, or binary Hex. The Listen port accepts messages on the local machine where the TCP Dump Tool is executed, and forwards the message to the **Forward Host** over the **Forward Port** specified.

To save the request being sent, specify a file with the full path in the Save request to file option

Configuration



TCP Dump Configuration

TCP Dump configuration

Listen Port: 9001
Forward Port: 9000
Forward Host: localhost
Dump Mode: [✓ Text] Hex
Save file location:

Configuration	Description
Listen Port	The port number that TCP dump should listen to, TCP Dump Tool will will start a socket on this port, thus this port should not be already bound to another program.
Forward Port	The Forward port number which the back end server runs
Forward Host	The Forward Hostname which the back end server runs
Dump Mode	A user can dump either a text format or binary Hex format
Save File Location	If the user wants the TCP dump to be saved to a file in order to analyze it later give the file location here.

Section 11

Frequently Asked Questions

Q: Component and Connectors are not shown in Component Pallet

A: Make sure the project is added as a Maven project and all the specified dependencies in the pom.xml file are shown in **Maven Projects** tab

Q: pom.xml file is invalid

A: If you are getting an error saying *Failed to load the dependencies from the pom.xml file. Make sure Project SDK is specified*, then make sure you have specified a valid **Project SDK** under project structure (File → Project Structure…)

Q: Message Execution paths are not getting highlighted

A: You need to specify Oracle JDK as the IDEA's running JDK. In order to do that, press Shift key twice and type **Switch IDE boot JDK** and execute that action. Next, from the dialog window, select the Oracle JDK and restart IDEA.