Enterprise Integration Patterns with WSO2 ESB

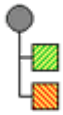
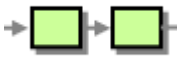
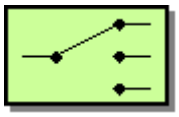
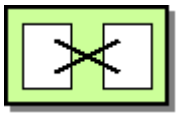
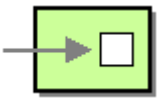
Table of Contents

Enterprise Integration Patterns with WSO2 ESB

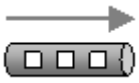
Enterprise Application Integration (EAI) is key to connecting business applications with heterogeneous systems. Over the years, architects of integration solutions have invented their own blend of patterns in a variety of ways. But most of these architectures have similarities, initiating a set of widely accepted standards in architecting integration patterns. Most of these standards are described in the **Enterprise Integration Patterns Catalog** available at: http://www.eaipatterns.com/toc.html.

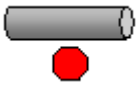
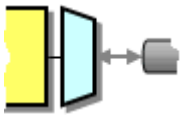
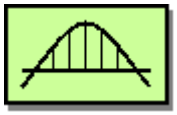
In this guide, we have shown how each pattern in the patterns catalog can be simulated using various constructs in WSO2 ESB. Click on a topic in the list below for details.

Messaging Systems

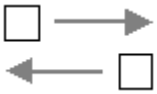
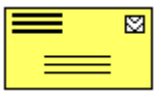
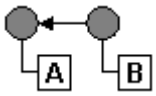
	Message Channels	How one application communicates with another using messaging.
	Message	How two applications connected by a message channel exchange a piece of information.
	Pipes and Filters	How to perform complex processing on a message while maintaining independence and flexibility.
	Message Router	How to decouple individual processing steps so that messages can be passed to different filters depending on conditions.
	Message Translator	How systems using different data formats communicate with each other using messaging.
	Message Endpoint	How an application connects to a messaging channel to send and receive messages.

Messaging Channels

	Point-to-Point Channel	How the caller can be sure that exactly one receiver will receive the document or perform the call.
	Publish-Subscribe Channel	How the sender broadcasts an event to all interested receivers.
	Datatype Channel	How the application sends a data item such that the receiver will know how to process it.

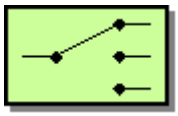
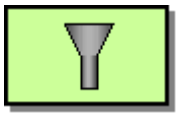
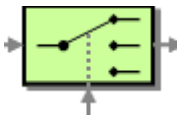
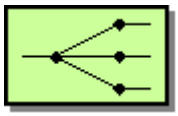
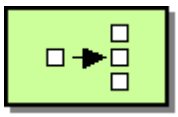
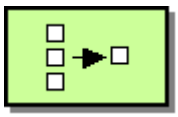
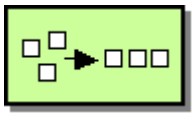
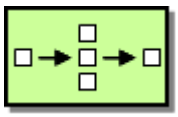
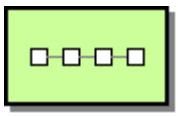
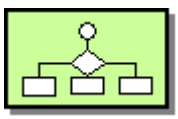
	Invalid Message Channel	How a messaging receiver gracefully handles a message that makes no sense.
	Dead Letter Channel	What the messaging system does with a message it cannot deliver.
	Guaranteed Delivery	How the sender ensures delivery of a message, even if the messaging system fails.
	Channel Adapter	How to connect an application to the messaging system to send/receive messages.
	Messaging Bridge	How multiple messaging systems can be connected so that messages available on one are also available on the others.
	Message Bus	An architecture enabling separate applications to work together in a decoupled fashion such that applications can be easily added or removed without affecting the others.

Message Construction

	Command Message	How messaging can be used to invoke a procedure in another application.
	Document Message	How messaging can be used to transfer data between applications.
	Event Message	How messaging can be used to transmit events from one application to another.
	Request-Reply	How an application that sends a message gets a response from the receiver.
	Return Address	How a replier knows where to send the reply.
	Correlation Identifier	How a requester that has received a reply knows which request the reply is for.
	Message Sequence	How messaging can transmit an arbitrarily large amount of data.

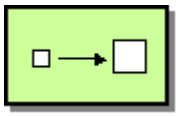
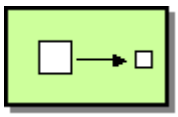
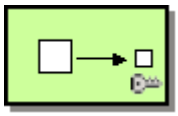
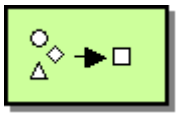
	Message Expiration	How a sender indicates when a message should be considered stale and therefore should not be processed.
	Format Indicator	How a message's data format can be designed to allow for possible future changes.

Message Routing

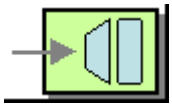
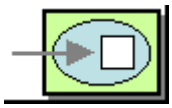
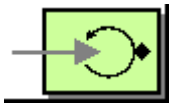
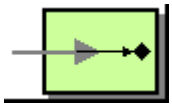
	Content-Based Router	How to handle a situation when the implementation of a single logical function (such as an inventory check) is spread across multiple physical systems.
	Message Filter	How a component avoids receiving uninteresting messages.
	Dynamic Router	How to avoid the dependency of a router in all possible destinations, while maintaining its efficiency.
	Recipient List	How to route a message to a list of dynamically specified recipients.
	Splitter	How to process a message if it contains multiple elements, each of which may have to be processed in a different way.
	Aggregator	How to combine the results of individual but related messages so that they can be processed as a whole.
	Resequencer	How to get a stream of related but out-of-sequence messages back into the correct order.
	Composed Msg. Processor	How to maintain the overall flow when processing a message consisting of multiple elements, each of which may require different processing.
	Scatter-Gather	How to maintain the overall flow when a message needs to be sent to multiple recipients, each of which may send a reply.
	Routing Slip	How to route a message consecutively through a series of steps when the sequence of the steps is not known at design time and may vary for each message.
	Process Manager	How to route a message through multiple processing steps, when the required steps may not be known at design time and may not be sequential.

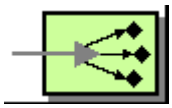
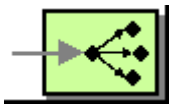
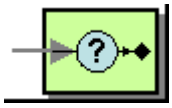
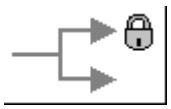
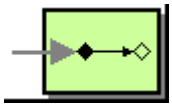
	Message Broker	How to decouple the destination of a message from the sender and maintain central control over the flow of messages.

Message Transformation

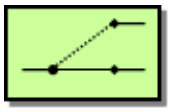
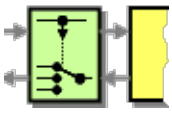
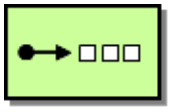
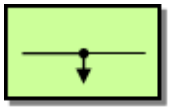
	Envelope Wrapper	How existing systems participate in a messaging exchange, which places specific requirements in the message format, such as message header fields or encryption.
	Content Enricher	How to communicate with another system if the message originator does not have all the required data items available.
	Content Filter	How to simplify dealing with a large message when you are interested only in a few data items.
	Claim Check	How to reduce the data volume of a message sent across the system without sacrificing information content.
	Normalizer	How to process messages that are semantically equivalent but arrive in a different format.
	Canonical Data Model	How to minimize dependencies when integrating applications that use different data formats.

Messaging Endpoints

	Messaging Gateway	How to encapsulate access to the messaging system from the rest of the application.
	Messaging Mapper	How to move data between domain objects and the messaging infrastructure, while keeping the two independent of each other.
	Transactional Client	How a client controls its transactions with the messaging system.
	Polling Consumer	How an application consumes a message when the application is ready.
	Event-Driven Consumer	How an application automatically consumes messages as they become available.

	Competing Consumers	How a messaging client processes multiple messages concurrently.
	Message Dispatcher	How multiple consumers on a single channel coordinate their message processing.
	Selective Consumer	How a message consumer selects which messages to receive.
	Durable Subscriber	How a subscriber avoids missing messages while it is not listening for them.
	Idempotent Receiver	How a message receiver deals with duplicate messages.
	Service Activator	How an application designs a service to be invoked via both messaging and non-messaging techniques.

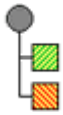
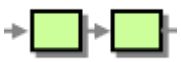
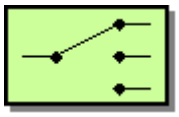
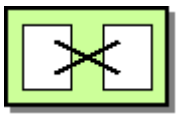
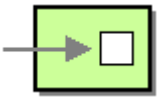
System Management

	Channel Purger	Removes unwanted messages, which can disturb tests or running systems, from a channel.
	Control Bus	Administers a messaging system that is distributed across multiple platforms and a wide geographic area.
	Detour	Routes a message through intermediate steps to perform validation, testing or debugging functions.
	Message History	Lists all applications that the message passed through since its origination.
	Message Store	Reports against message information without disturbing the loosely coupled and transient nature of a messaging system.
	Smart Proxy	Tracks messages on a service that publishes reply messages to the Return Address specified by the requester.
	Test Message	Ensures the health of message processing components by preventing situations such as garbling outgoing messages due to an internal fault.
	Wire Tap	Inspects messages that travel on a Point-to-Point Channel.

Messaging Systems

Messaging is one integration style out of many, used for connecting various applications in a loosely coupled, asynchronous manner. Messaging decouples the applications from data transferring, so that applications can concentrate on data and related logic while the messaging system handles the transferring of data.

This chapter introduces the basic patterns used when implementing enterprise integration using messaging and how they are simulated using the WSO2 ESB. These patterns are the fundamentals on which the rest of the chapters in this guide are built.

	Message Channels	How one application communicates with another using messaging.
	Message	How two applications connected by a message channel exchange a piece of information.
	Pipes and Filters	How to perform complex processing on a message while maintaining independence and flexibility.
	Message Router	How to decouple individual processing steps so that messages can be passed to different filters depending on conditions.
	Message Translator	How systems using different data formats communicate with each other using messaging.
	Message Endpoint	How an application connects to a messaging channel to send and receive messages.

Message Channels

This section explains, through an example scenario, how the Message Channels EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Message Channels
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Message Channels

Message Channels facilitate communication between applications. A sender adds a message to a particular channel, which a receiver can read. Channels allow the sender and receiver to synchronize.

For more information on Message Channels, refer to http://www.eaipatterns.com/MessageChannel.html.

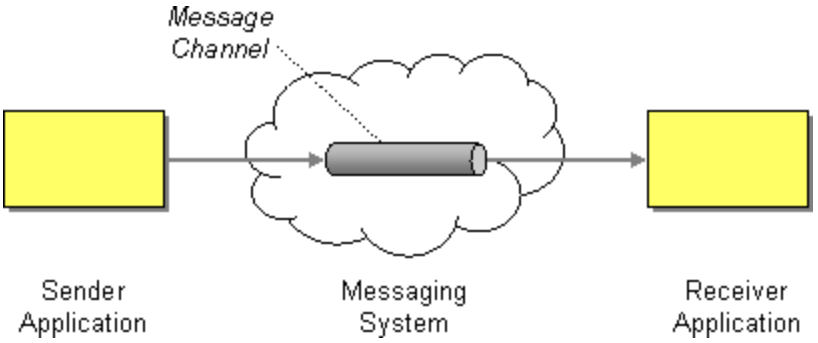


Figure 1: Message Channel EIP

Example scenario

The example scenario depicts how a stock inventory is made from a sender application to the receiver application through a Message Channel. The message channel retrieves message content from the sender, and it allows the receiver to read the content through the channel. The diagram below depicts how to simulate the example scenario using WSO2 ESB.

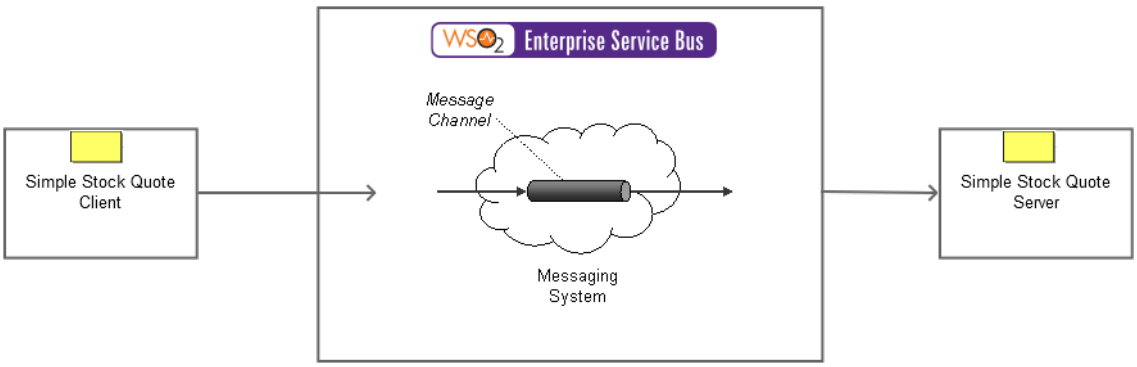


Figure 2: Example Scenario of the Message Channel EIP

Before digging into implementation details, let's look at how the core components in the example scenario map to the Message Channel EIP:

Message Channel EIP (Figure 1)	Message Channel Example Scenario (Figure 2)
Sender Application	Stock Quote Client
Message System	WSO2 ESB
Receiver Application	Stock Quote Server

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start an instance of Sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB Documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Message Channels -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
<!-- External sequence acts as a message channel -->
<sequence name="MessageChannel">
    <in>
     <send>
    <endpoint>
              <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
          </endpoint>
         </send>
     </in>
     <out>
         <send />
     </out>
</sequence>
<!-- Sender will invoke the following Sequence -->
 <sequence name="main">
    <!-- The request will first trigger to the following -->
    <in>
   <!-- Allows calling of the following sequence defined through the key -->
   <sequence key="MessageChannel" />
    </in>
    <out>
         <!-- The response given out through the message channel will be sent back
to the sender -->
        <send />
    </out>
</sequence>
</definitions>
```

Simulating the sample scenario

1. Send a request using the `Stock Quote Client` to WSO2 ESB in the following manner. Information on the `Stock Quote Client` and its operation modes are discussed in the Sample Clients section in the WSO2 ESB documentation.

   ```
   ant stockquote -Dtrpurl=http://localhost:8280
   ```
2. When you execute the command above, the request will be sent to the `Stock Quote` service. Notice the processed server log in the `Stock Quote` service console.

How the implementation works

Let's investigate in detail the elements of the ESB configuration. The line numbers below refer to the ESB configuration shown above.

- **main sequence** [line 18 in ESB config] - The default sequence that is triggered when the user invokes the ESB server.

- **in** [line 6 in ESB config] - The message is directed to the `in` mediator when it is received by the `main` sequence.
- **out** [line 13 in ESB config] - Triggered after execution of the steps defined through the `in` mediator.
- **sequence** [line 5 in ESB config] - An external sequence specified by `key` where requests are directed.
- **send** [line 7 in ESB config] - When a matching case is found, the `send` mediator routes the message to the endpoint indicated by the address URI.

Message

This section explains how the Message EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Message
- How WSO2 ESB supports the Message EIP

Introduction to Message

In an enterprise that uses Message Channels to connect between two different applications, you must package information as a data set. Data sets facilitate information transmission from one application to another. The Message EIP is used as the transfer medium in this case. A message has two parts: a **header** and a **body**. A header holds information about the data being transmitted, its origin, and its destination. A body holds the actual data.

For more information on the Message EIP, refer to http://www.eaipatterns.com/Message.html.

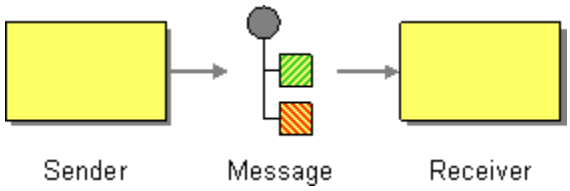


Figure 1: Message EIP

How WSO2 ESB supports the Message EIP

As described in Messaging Bridge, irrespective of the format of a request you send from the client application, WSO2 ESB transforms the message into a SOAP envelope, which adheres to the Message EIP. The following code segment illustrates the transformed message of a typical request. Notice that it has a header and a body.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">

  <soapenv:Header />
   <soapenv:Body>

      <ser:getQuote>
         <ser:request>
             <ser:symbol>Foo</ser:symbol>
         </ser:request>
      </ser:getQuote>

    </soapenv:Body>

</soapenv:Envelope>
```

Pipes and Filters

This section explains, through an example scenario, how the Pipes and Filters EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Pipes and Filters
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Pipes and Filters

The Pipes and Filters EIP breaks down a large task into smaller sub sets of independent steps that are chained together. It is useful when integrating applications requiring a sequence of processing steps to perform a single event. The main use case of the EIP is to maintain independence and flexibility between each processing step.

For more information on Pipes and Filters, refer to http://www.eaipatterns.com/PipesAndFilters.html.

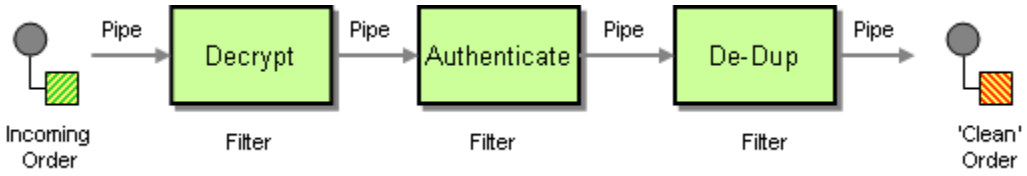


Figure 1: Pipes and Filters EIP

Example scenario

The example scenario is a stock quote service, which transmits a client request through a set of filtering steps such as:

- Verification of the user name
- Verification of the user ID

When the filtering criteria mentioned above is met, the stock quote request will be sent to the stock quote server for processing. The diagram below depicts how to simulate the example scenario using WSO2 ESB.

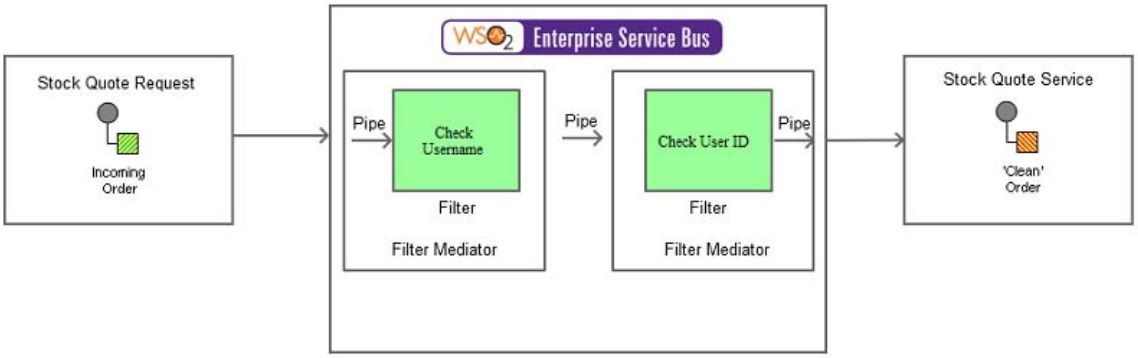


Figure 2: Example Scenario of the Pipes and Filters EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Pipes and Filters EIP by comparing their core components.

Pipes and Filters EIP (Figure 1)	Example Scenario of the EIP (Figure 2)
Incoming Order	Stock Quote Request
Filters	We use Filter Mediators to check the user name and user ID to verify the validity of the message. If the message meets the criteria of the first filter, we forward it to the next filter, and then to the stock quote service.
Clean Order	Stock Quote Service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start an instance of the sample Axis2 server. For instructions, refer to ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to the **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
<!-- Defines the Sequence of Processing Steps -->
   <sequence name="PipesAndFilters">
        <log level="full" xmlns="http://ws.apache.org/ns/synapse"/>
  <!-- Checks For the User Name -->
        <filter  xmlns:m0="http://services.samples"
source="//m0:credentials/m0:name" regex="UserName">
     <!-- If the filtered condition is satisfied -->
     <then>
      <!-- Checks for the User ID -->
         <filter xmlns:m1="http://services.samples" source="//m1:credentials/m1:id"
regex="001">
                <then>
                    <!-- The filtered message will be routed to the end point -->
                    <send>
                        <endpoint>
                          <address
uri="http://localhost:9000/services/SimpleStockQuoteService?wsdl"/>
                        </endpoint>
                    </send>
                    </then>
                <else>
                    <drop/>
                </else>
            </filter>

             </then>
             <!-- If the condition was not satisfied -->
      <else>
                 <drop/>
             </else>
        </filter>
     </sequence>
     <!-- Will be triggered first -->
  <sequence name="main">
        <in>
    <!-- Will direct an incoming request to the specified sequence -->
    <sequence key="PipesAndFilters"/>
        </in>
        <out>
            <send/>
        </out>
    </sequence>

</definitions>
```

Simulating the sample scenario

1. Send a request using a SOAP client (such as SoapUI) to WSO2 ESB in the following manner,

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header>
     <ser:credentials>
          <ser:name>UserName</ser:name>
          <ser:id>001</ser:id>
      </ser:credentials>
   </soapenv:Header>
   <soapenv:Body>
        <ser:getQuote>
         <ser:request>
             <ser:symbol>Foo</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

2. After sending the request to the ESB through the client, the client should be able to notice that the request is successfully generated in the stock quote server. If the name or ID is changed, request generation will fail.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **main sequence** [line 34 in ESB config] - The default sequence that gets triggered when the user invokes the ESB server.
- **in** [line 35 in ESB config] - The message is directed to the `in` mediator when it is received by the `main` sequence.
- **out** [line 39 in ESB config] - Triggered when a response is received for the message that was sent by the `in` mediator.
- **sequence** [line 37 in ESB config] - An external sequence specified by `key` (in this example, the `PipesAndFilters` sequence) where requests are directed.
- **filter** [line 7 in ESB config] - Filters an incoming message based on the constraints specified through `regex`.
- **then** [line 9 in ESB config] - Actions to take if the conditions are satisfied through the filtering criteria.
- **else** [line 28 in ESB config] - Actions to take if the conditions are not satisfied.

Message Router

This section explains, through an example scenario, how the Message Router EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Message Router

The Message Router EIP reads the content of a message and routes it to a specific recipient based on its content.

When the implementation of a specific logical function is distributed across multiple physical systems, an incoming request needs to be passed on to the correct service based on the request's content. A Message Router is useful in handling such scenarios.

The following diagram depicts the Message Router's behavior where the router performs a logical function (such as an inventory check). It receives a request message (new order), reads it, and routes the request to one of the two recipients according to the message's content. The router is also defined as a special type of a filter.

For more information on the Message Router, refer to http://www.eaipatterns.com/MessageRouter.html.

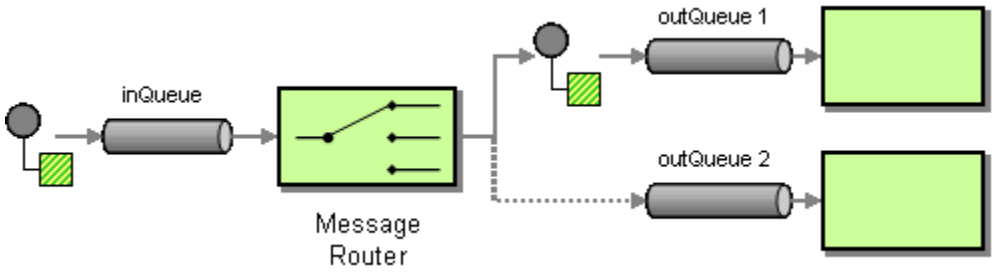


Figure 1: Message Router EIP

Example scenario

The example scenario depicts an inventory for stocks, and how Message Router EIP routes a message to a different service based on the message's content. When the router receives a stock request, it reads the content of the request. If the request is made to **Foo**, it is routed to **FooOutQueue**. If the request is for **Bar**, it is routed to **BarOutQueue**.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

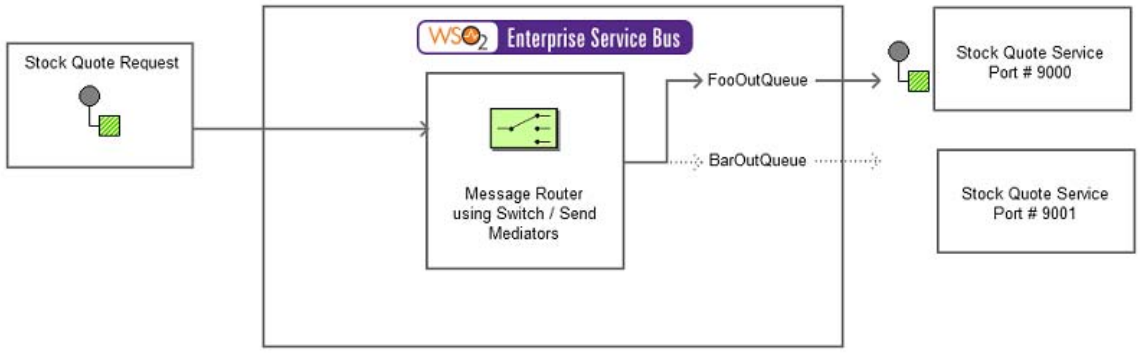


Figure 2: Example Scenario of the Message Router EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Message Router EIP by comparing their core components.

Message Router EIP (Figure 1)	Message Router Example Scenario (Figure 2)

Incoming message	Stock Quote Request
Message router	The **Switch** and **Send** mediators of the WSO2 ESB simulate the **Message Router** EIP. The Switch Mediator depicts the **Router** and observes the content of the message, while the Send Mediator sends the message to a selected recipient. Each case defined should decide on routing the message to the appropriate service.
Outgoing queues	**FooOutQueue** and **BarOutQueue** act as two separate services in the example scenario.

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances in ports 9000 and 9002. For instructions, refer to ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to the **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
      <parameter name="cachableDuration">15000</parameter>
    </registry>
    <!-- Receiving sequence which will be the message router -->
    <sequence name="MessageRoute">
      <in>
        <!-- Would analyze the data for filtering -->
        <switch xmlns:m0="http://services.samples"
source="//m0:getQuote/m0:request/m0:symbol">
            <!-- If the content is "Foo" -->
            <case regex="Foo">
                <!-- Sends the information to the FooOutQueue -->
                <send>
                  <endpoint>
                      <address
uri="http://localhost:9000/services/SimpleStockQuoteService?wsdl"/>
                  </endpoint>
                </send>
            </case>
            <!-- If the content is "Bar" -->
            <case regex="Bar">
                <!-- Sends the information to the BarOutQueue -->
                <send>
                  <endpoint>
                      <address
uri="http://localhost:9001/services/SimpleStockQuoteService?wsdl"/>
```

```
                  </endpoint>
                </send>
            </case>
            <default>
                <property name="symbol" expression="fn:concat('Normal Stock - ',
//m0:getQuote/m0:request/m0:symbol)"/>
            </default>
         </switch>
      </in>
      <out>
          <send/>
      </out>
    </sequence>
    <sequence name="fault">
        <log level="full">
          <property name="MESSAGE" value="Executing default 'fault' sequence"/>
          <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
          <property name="ERROR_MESSAGE"
expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
    </sequence>
    <sequence name="main">
      <in>
         <!-- Will call the message router -->
         <sequence key="MessageRoute"/>
      </in>
      <out>
         <send/>
      </out>
```

```
        <description>Message Router</description>
    </sequence>
</definitions>
```

Simulating the sample scenario

1. Next, we use a sample SOAP client by the name `Stock Quote` client to send a request using WSO2 ESB in the following manner. For more information on the `Stock Quote` client, refer to the Sample Clients section in the WSO2 ESB documentation.

   ```
   ant stockquote -Dtrpurl=http://localhost:8280 -Dsymbol=Foo
   ```

2. After you execute the above command through the client, observe that the request is transferred to `Foo inventory service`. If the `-Dsymbol` parameter is changed to `Bar`, the request goes to `Bar inventory service`. The structure of the request is as follows:

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

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **switch** [line 10 in ESB config] - Observes the message and filters out the content to the given xPath expression.
- **case** [line 12 in ESB config] - The filtered content will be matched with the specified regular expression.
- **send** [line 14 in ESB config] - When a matching case for `Foo` is found, the message is routed to the specified endpoint indicated in the address URI.
- **send** [line 23 in ESB config] - When a matching case for `Bar` is found, the message is routed to the specified endpoint indicated in the address URI.
- **default** [line 29 in ESB config] - If a matching condition is not found, the message will be diverted to the default case.
- **main sequence** [line 46 in ESB config] - The default sequence that is triggered when the user invokes the ESB server.
- **in** [line 47 in ESB config] - After the message is received by the main sequence, it is diverted to the `in` media tor.
- **out** [line 51 in ESB config] - Triggered after execution of steps defined through the `in` mediator.
- **sequence** [line 49 in ESB config] - An external sequence specified by `key` (in this example, the `MessageRoute` sequence) where requests are directed.
- **send** [line 52 in ESB config] - Routes the message back to the requesting client.

Message Translator

This section explains, through an example scenario, how the Message Translator EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Message Translator

Different applications typically use different data types. Therefore, for two applications to successfully communicate, we should intermediately translate the messages that pass from one application to the data type compatible with the receiving application. A translator changes the context of a message from one interface to another, allowing messages to adhere to message context rules of the back-end service.

The Message Translator EIP is responsible for message translating to ensure compatibility between applications supporting different data types. For more information, refer to http://www.eaipatterns.com/MessageTranslator.html.

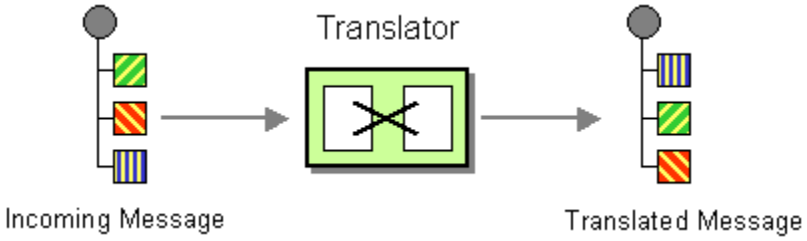


Figure 1: Message Translator EIP

Example scenario

The example scenario is an inventory for stocks. It illustrates how the sender sends a request in one format, which is then transformed into another format compatible with the receiver. The format of the request is as follows:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header>
   </soapenv:Header>
   <soapenv:Body>
        <ser:Code>Foo</ser:Code>
   </soapenv:Body>
</soapenv:Envelope>
```

The message format compatible with the receiver is as follows:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <ser:symbol>Foo</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

All requests in the first format should be translated to the second by WSO2 ESB. The diagram below depicts how to simulate this scenario using WSO2 ESB.

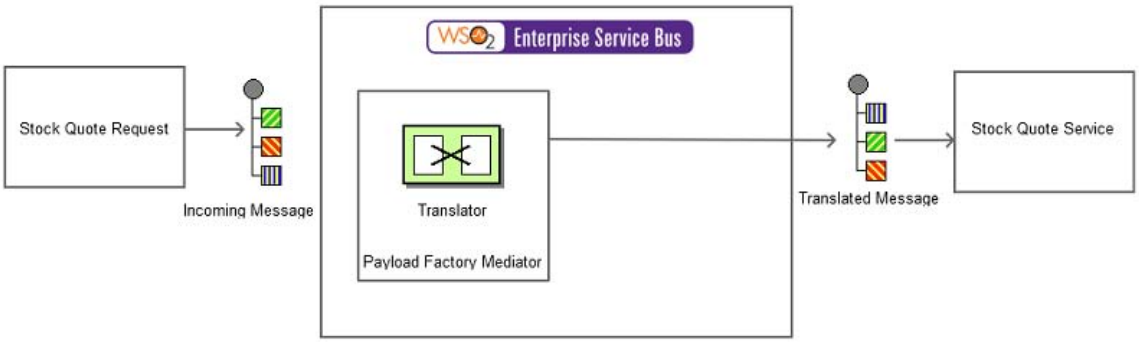


Figure 2: Example Scenario of the Message Translator EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Message Translator EIP by comparing their core components.

Message Translator EIP (Figure 1)	Message Translator Example Scenario (Figure 2)
Incoming Message	Stock Quote Request
Translator	The translation is done through the Payload Factory Mediator

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start an instance of Axis2 server. For instructions, refer to ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to the **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.


```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <!-- Will trigger when a request is sent to the ESB -->
   <sequence name="main">
      <in>
   <!-- Will transform the incoming message to the format specified below -->
   <payloadFactory>
            <format>
               <m:getQuote xmlns:m="http://services.samples">
                  <m:request>
                     <m:symbol>$1</m:symbol>
                  </m:request>
               </m:getQuote>
            </format>
            <args>
               <arg xmlns:m0="http://services.samples" expression="//m0:Code"/>
            </args>
         </payloadFactory>
         <send>
            <endpoint>
               <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
            </endpoint>
         </send>
      </in>
      <out>
       <send/>
   </out>
   </sequence>
</definitions>
```

Simulating the sample scenario

1. Send a request using a SOAP request client (such as SoapUI) to WSO2 ESB in the following manner.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header>
   </soapenv:Header>
   <soapenv:Body>
        <ser:Code>Foo</ser:Code>
   </soapenv:Body>
</soapenv:Envelope>
```

2. After sending the request to the ESB through the client, notice that the request is successfully generated in

the stock quote server.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **main sequence** [line 12 in ESB config] - The default sequence that gets triggered when the user invokes the ESB server.
- **in** [line 13 in ESB config] - The message is directed to the `in` mediator when it is received by the `main` sequence.
- **out** [line 33 in ESB config] - Triggered when a response is received for the message that was sent by the `in` mediator.
- **payload factory** [line 15 in ESB config] - Will transform the message to the format denoted within the mediator.

Message Endpoint

This section explains, through an example scenario, how the Message Endpoint EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Message Endpoint

The Message Endpoint EIP encapsulates the messaging system from the rest of the application and customizes a general messaging API for a specific application and task. Therefore, you can change the message API just by changing the endpoint code. This improves the maintainability of applications.

For more information, refer to http://www.eaipatterns.com/MessageEndpoint.html

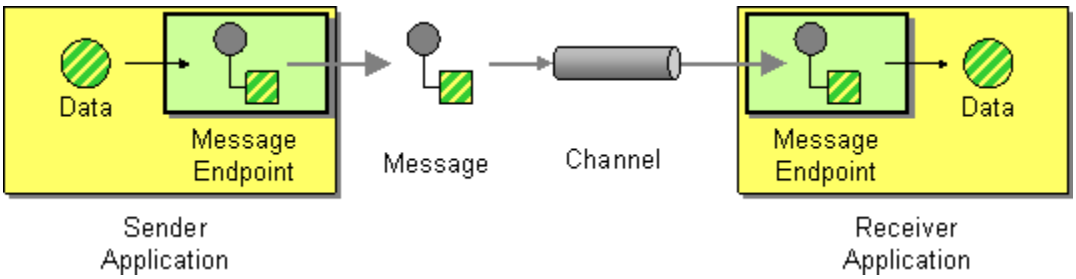


Figure 1: Message Endpoint EIP

Example scenario

The example scenario is an inventory for stocks. It illustrates how a stock quote is generated when a request is sent to the ESB. The sender sends the request to the WSO2 ESB, where the request is then diverted to the stock quote service. The diagram below depicts how to simulate the example scenario using WSO2 ESB.

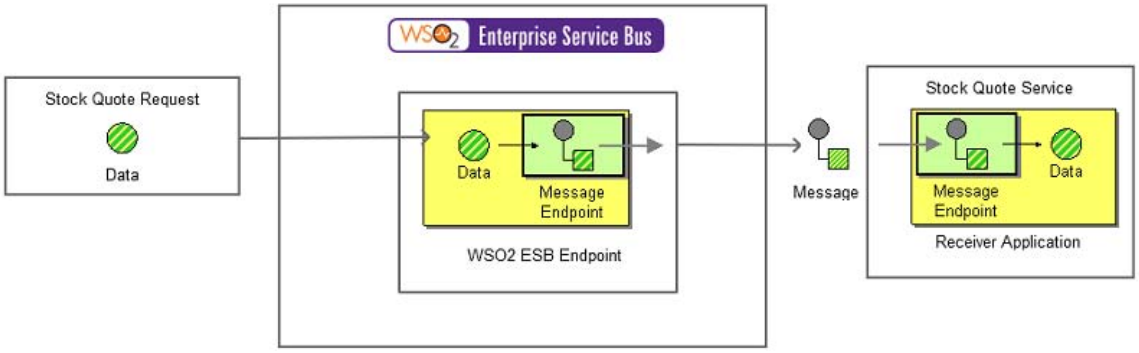


Figure 2: Example Scenario of the Message Endpoint EIP

Before digging into the implementation details, let's take a look at the relationship between the example scenario and the Message Endpoint EIP by comparing their core components.

Message Endpoint EIP (Figure 1)	Message Endpoint Example Scenario (Figure 2)
Data	Stock Quote Request
Message Endpoint	WSO2 ESB Endpoint
Receiver Application	Stock Quote service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances in ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to the **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <!-- Will trigger the following sequence when ESB is invoked -->
   <sequence name="main">
      <in>
   <!-- Sends the message to the specified service -->
         <send>
            <endpoint>
               <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
            </endpoint>
         </send>
      </in>
      <out>
         <send/>
      </out>
   </sequence>
</definitions>
```

Simulating the sample scenario

1. Send a request using a SOAP request client (such as SoapUI) to WSO2 ESB in the following manner:

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

2. After sending the request to the ESB through the client, notice that the Stock Quote service has received the inventory and logged the message.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **main sequence** [line 12 in ESB config] - The default sequence that is triggered when the user invokes the

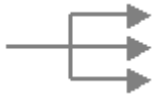
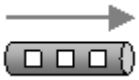
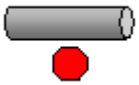
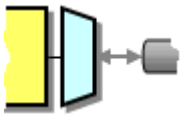
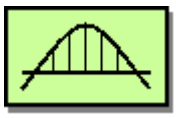
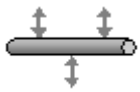
ESB server.

- **in** [line 13 in ESB config] - The message is directed to the `in` mediator when it is received by the `main` sequence.
- out [line 21 in ESB config] - Triggered after execution of steps defined through the `in` mediator.
- sequence [line in ESB config] - An external sequence specified by `key` where requests are directed.
- **endpoint** [line 16 in ESB config] - The destination where the message will be sent.

Messaging Channels

A Message channel is a basic architectural pattern of a messaging system and is used fundamentally for exchanging data between applications. An application can use a channel to make a specific type of data available to any receiver applications that need to consume that type of data.

This chapter introduces different types of channels used in a messaging system, their behaviors, and how each of them can be simulated using WSO2 ESB.

	Point-to-Point Channel	How the caller can be sure that exactly one receiver will receive the document or perform the call.
	Publish-Subscribe Channel	How the sender broadcasts an event to all interested receivers.
	Datatype Channel	How the application sends a data item such that the receiver will know how to process it.
	Invalid Message Channel	How a messaging receiver gracefully handles a message that makes no sense.
	Dead Letter Channel	What the messaging system does with a message it cannot deliver.
	Guaranteed Delivery	How the sender ensures delivery of a message, even if the messaging system fails.
	Channel Adapter	How to connect an application to the messaging system to send/receive messages.
	Messaging Bridge	How multiple messaging systems can be connected so that messages available on one are also available on the others.
	Message Bus	An architecture enabling separate applications to work together in a decoupled fashion such that applications can be easily added or removed without affecting the others.

Point-to-Point Channel

This section explains, through an example scenario, how the Point-to-Point Channel EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Point-to-Point Channel
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario

- How the implementation works

Introduction to Point-to-Point Channel

The Point-to-Point Channel EIP pattern allows only a single receiver to consume a sent message when there are multiple receivers waiting to consume it. For more information, refer to http://www.eaipatterns.com/PointToPointChannel.html

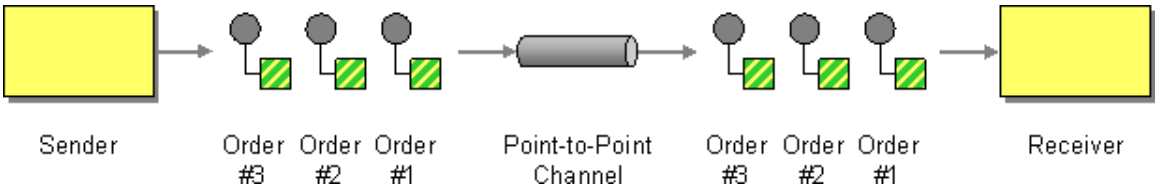


Figure 1 : Point-to-Point Channel EIP

Example scenario

The example scenario is an inventory for stocks. It illustrates how a stock quote is generated, which only a single consumer receives at a given time. The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

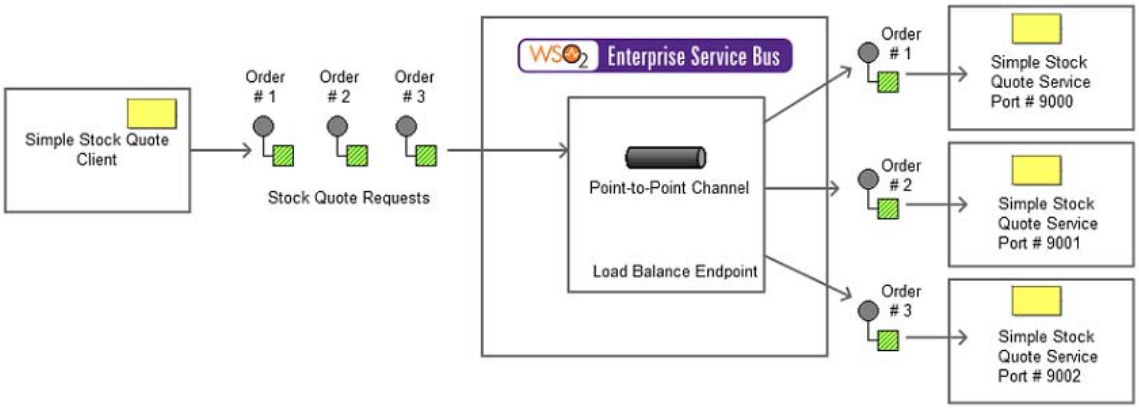


Figure 2: Example Scenario of the Point-to-Point Channel EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Point-to-Point Channel EIP by comparing their core components.

Point-to-Point EIP (Figure 1)	Point-to-Point Channel Example Scenario (Figure 2)
Sender	Stock Quote Client
Order	Stock Quote Requests
Point to Point Channel	Load-balance Endpoint
Receiver	Three instances of the Stock Quote service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances in ports 9000, 9001 and 9002. For instructions, refer to ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB Documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- Point to Point Channel-->
<definitions xmlns="http://ws.apache.org/ns/synapse">
  <proxy xmlns="http://ws.apache.org/ns/synapse" name="PointToPointProxy"
transports="http https" startOnLoad="true" >
    <target>
         <inSequence>
      <send>
        <endpoint>
        <!-- Channel With Multiple Enpoints Load Balancer Will Ensure that only one
will receive it -->
         <loadbalance>
                         <endpoint>
                               <address
uri="http://localhost:9000/services/SimpleStockQuoteService/" />
                         </endpoint>
                         <endpoint>
                             <address
uri="http://localhost:9001/services/SimpleStockQuoteService/" />

                         </endpoint>
                         <endpoint>
                              <address
uri="http://localhost:9002/services/SimpleStockQuoteService/" />
                         </endpoint>
         </loadbalance>
        </endpoint>
      </send>
        </inSequence>
        <outSequence>
            <send/>
        </outSequence>
    </target>
  </proxy>
</definitions>
```

Simulating the sample scenario

1. Send a request using the `Stock Quote` client to WSO2 ESB in the following manner. Information about the `Stock Quote` client and its operation modes are discussed in the Sample Clients section in the WSO2 ESB documentation.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/PointToPointProxy
-Dsymbol=Foo
```

2. The structure of the request is as follows:

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

Note that out of all three instances of the `Stock Quote` service (Axis2 server), only one server acquires the sent request at a given time.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **Proxy Service** [line 3 in ESB config] - Abstracts the routing logic from the client. Whatever the request is, the client sends it only to the exposed service.
- **inSequence** [line 5 in ESB config] - When the service is invoked through the client, the message is picked up by the inSequence and sent according to the routing logic.
- **send** [line 6 in ESB config] - When a matching case is found, the `send` mediator will route the message to an endpoint specified in the address URI.
- **loadbalance** [line 9 in ESB config] - Manages the array of services defined within this endpoint. It streams the request only to one instance, which is selected using the given algorithm.
- **outSequence** [line 23 in ESB config] - Receives the response from an endpoint and sends the response back to the client.

Publish-Subscribe Channel

This section explains, through an example scenario, how the Publish-Subscribe Channel EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Publish-Subscribe Channel

The Publish-Subscribe Channel EIP receives messages from the input channel, and then splits and transmits them among its subscribers through the output channel. Each subscriber has only one output channel. For more information, refer to http://www.eaipatterns.com/PublishSubscribeChannel.html

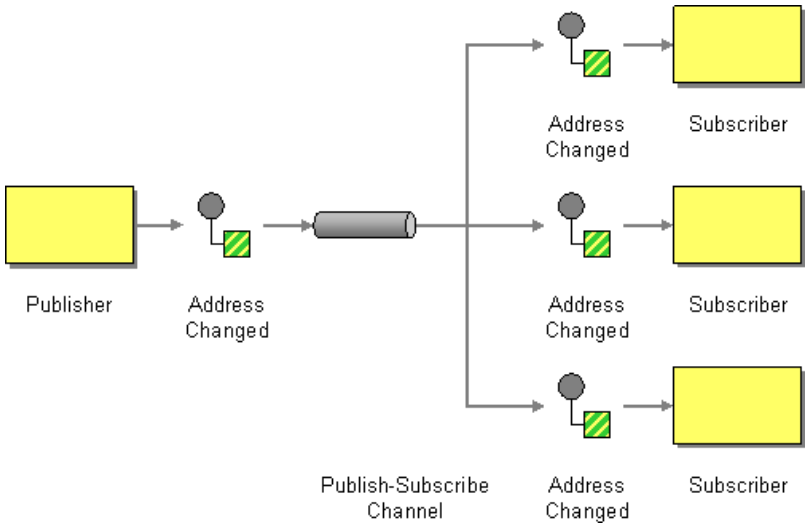


Figure 1 : Publisher-Subscribe Channel EIP

Example scenario

The example scenario depicts an inventory for stocks, and how the EIP distributes a sent message among several subscribers. It has several Stock Quote (Axis2) server instances. When a message arrives to the ESB, it is transmitted to these server instances, each of which acts as a subscriber through the event mediator.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

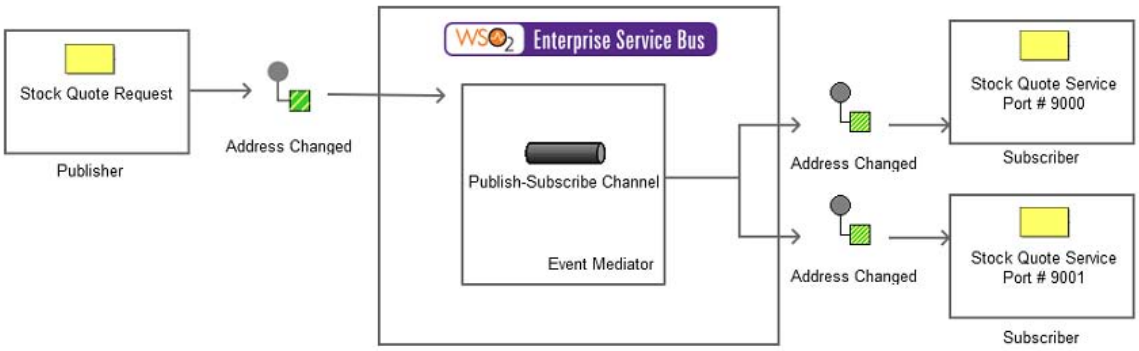


Figure 2: Example Scenario of the Publish-Subscribe Channel EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Publish-Subscribe Channel EIP by comparing their core components.

Publish-Subscribe Chanel EIP (Figure 1)	**Publish-Subscribe Channel Example Scenario (Figure 2)**
Publisher	Stock Quote Request
Publisher Subscriber Channel	Event Mediator
Subscriber	Stock Quote server instance

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two sample Axis2 server instances in ports 9000 and 9001. For instructions, refer to ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.
3. Following the steps below to create an event.
    - Sign in to the ESB management console (`https://localhost:9443/carbon`), click **Topics** from the **Manage** menu, and then click **Add**.
    - Enter the name `PublisherSubsciber` for the topic and click **Add Topic**.
    - You will be directed to the **Topic Browser** tree view where the newly created topic will be shown. Click on the new topic and select the **Subscribe** option to create a static subscription. Enter the value `http://localhost:9000/services/SimpleStockQuoteService` in the **Event Sink URL** field and click **Subscribe**.
    - Repeat these steps to add another subscriber in port 9001.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
      <event topic="PublisherSubsciber"/>
   </sequence>
</definitions>
```

Simulating the sample scenario

1. Send a request using the `Stock Quote` client to WSO2 ESB in the following manner. For information about the `Stock Quote` client, refer to the section Sample Clients in the WSO2 ESB documentation.

   ```
   ant stockquote -Dtrpurl=http://localhost:8280 -Dsymbol=Foo
   ```

2. After executing the above command, note that both `Stock Quote` service instances log a message accepting the request. The structure of the request is as follows:

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

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **main sequence** [line 10 in ESB config] - The default sequence that gets triggered when the user invokes the ESB server.
- **event** [line 12 in ESB config] - Allows you to define a set of receivers. For more information, refer to Eventing.

Datatype Channel

This section explains, through an example scenario, how the Datatype Channel EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Datatype Channel
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Datatype Channel

This EIP creates a separate channel for each type of data so that all the messages on a given channel will contain the same data type. The sender, who knows the data type, should select the appropriate channel on which to send the message. The receiver knows which type of data a message contains based on the channel in which it is received. For more information, refer to http://www.eaipatterns.com/DatatypeChannel.html.

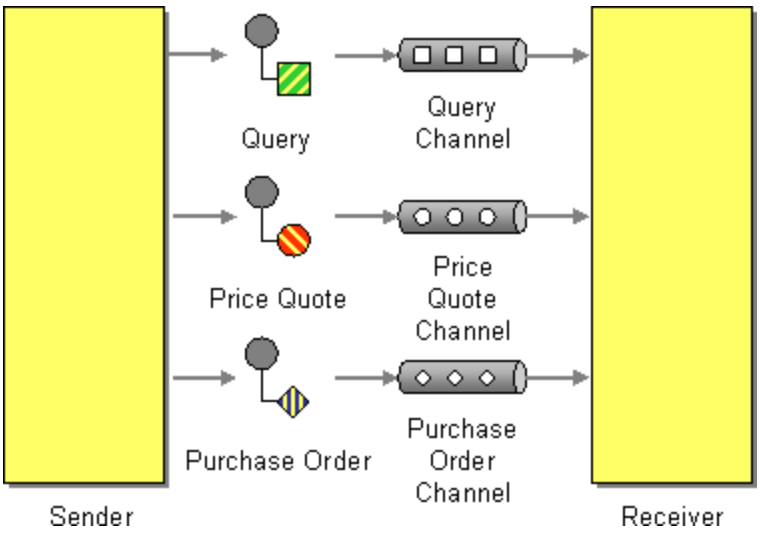


Figure 1: Datatype Channel EIP

Example scenario

The example scenario depicts a `Stock Quote` service deployed in Axis2 server. It offers several service operations to the user. WSO2 ESB uses the filter mediator to identify each action that is specified by the sender and diverts the request into the appropriate sequence. Each sequence acts as a separate channel. The sender experiences the decomposition of channels through a log message indicated by the ESB. There will be a different log message for each operation the sender requests.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

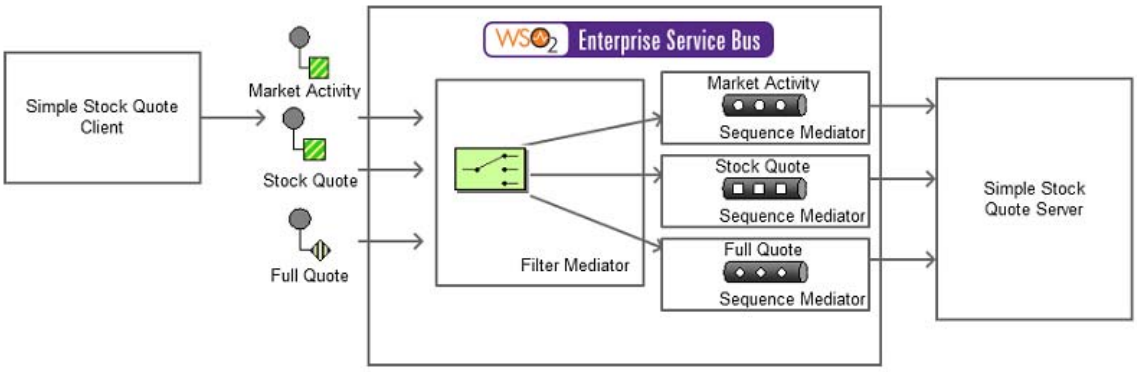


Figure 2: Example Scenario of the Datatype Channel EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Datatype Channel EIP by comparing their core components.

Datatype Channel EIP (Figure 1)	**Datatype Channel Example Scenario (Figure 2)**
Sender	Client
Datatype Channel	Filter and Sequence mediators of the ESB. The Filter Mediator specifies which datatype channel to use, and the ESB defines each datatype channel as a Sequence Mediator.

Receiver	Stock Quote Server Instance

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Deploy the `SimpleStockQuoteService` by starting the Sample Axis2 server on port 9000. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <endpoint name="StockQuoteReceiver">
      <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
   </endpoint>
   <sequence name="MarketActivity">
      <in>
         <log level="custom">
            <property name="Messagin_Channel" value="MARKET_ACTIVITY"/>
         </log>
         <send>
            <endpoint key="StockQuoteReceiver"/>
         </send>
      </in>
      <out>
         <send/>
      </out>
   </sequence>
   <sequence name="FullQuote">
      <in>
         <log level="custom">
            <property name="Messagin_Channel" value="FULL_QUOTE"/>
         </log>
         <send>
            <endpoint key="StockQuoteReceiver"/>
         </send>
      </in>
      <out>
         <send/>
      </out>
   </sequence>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
```

```
<sequence name="StockQuote">
   <in>
      <log level="custom">
         <property name="Messagin_Channel" value="STOCK_QUOTE"/>
      </log>
      <send>
         <endpoint key="StockQuoteReceiver"/>
      </send>
   </in>
   <out>
      <send/>
   </out>
</sequence>
<sequence name="main">
   <log/>
   <in>
      <filter source="get-property('Action')" regex="/*urn:getQuote/*">
         <then>
            <sequence key="StockQuote"/>
         </then>
         <else/>
      </filter>
      <filter source="get-property('Action')" regex="/*urn:getFullQuote/*">
         <then>
            <sequence key="FullQuote"/>
         </then>
         <else/>
      </filter>
      <filter source="get-property('Action')" regex="/*urn:getMarketActivity/*">
         <then>
            <sequence key="MarketActivity"/>
         </then>
         <else/>
      </filter>
   </in>
   <out>
      <send/>
```

```
        </out>
    </sequence>
</definitions>
```

Simulating the sample scenario

1. Send a request using the `Stock Quote` client to WSO2 ESB in the following manner. For information about the Stock Quote client, refer to the section Sample Clients in the WSO2 ESB documentation.

   ```
   ant stockquote -Dtrpurl=http://localhost:8280/ -Dmode=quote
   ```

2. Also execute the command above with `-Dmode=[quote | marketactivity | fullquote]`, and observe the ESB's log of the corresponding values `STOCK_QUOTE`, `MARKET_ACTIVITY` and `FULL_QUOTE`.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **Endpoint** [line 2 of the ESB config] - Defines an endpoint referenced by a name that contains an `<address>` element with the endpoint address of a particular service.
- **Sequence** [line 5 of the ESB config] - The Sequence Mediator defines a sequence block, callable by its key (defined in the name attribute). Each sequence block has its own `<in>` and `<out>` blocks.
- **Sequence main** [line 52 of the ESB config] - The main sequence, which is always run first.
- **Filter** [line 55 of the ESB config] - A Filter mediator that uses the `get-property` XPath expression to find the `Action` field from the SOAP header. The regular expression that is defined in the regex attribute tries to match the value in the Action field. If successfully matched, it calls the relevant sequence by its key. In line 55, if `get-property('Action')` returns `urn:getQuote`, the `StockQuote` sequence defined in line 39 is called.

Invalid Message Channel

This section explains, through an example scenario, how the Invalid Message Channel EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Invalid Message Channel

The Invalid Message Channel EIP pattern allows administrators to define an error indication that appears when an endpoint fails to process a request. For more information, refer to http://www.eaipatterns.com/InvalidMessageChannel.html.

Figure 1: Invalid Message Channel EIP

Example scenario

The example scenario creates a deliberate error situation to demonstrate how the ESB handles errors on message failures. It requires a live Axis2 server instance to successfully provide a response to the sender, and the server instance will be shut down before sending a request. You will observe how the ESB directs the process to the `faultsequence` mediator, which indicates the message invalidity to the user.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

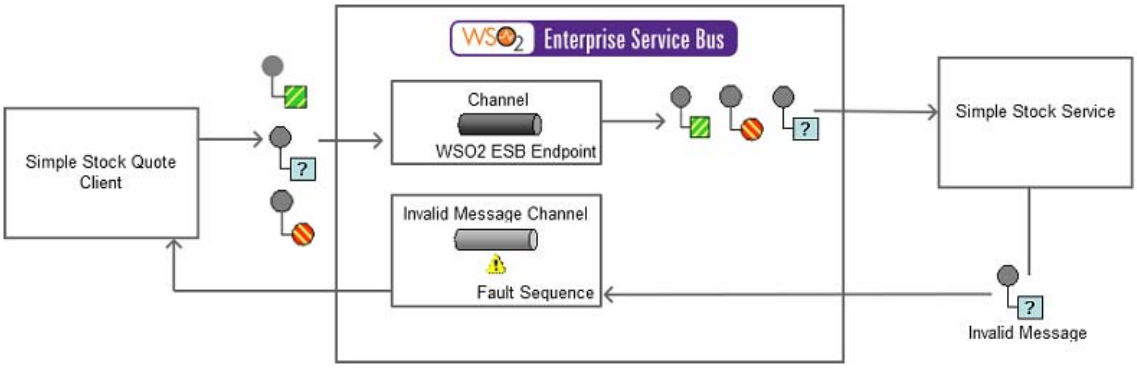


Figure 2: Example Scenario of the Invalid Message Channel EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Invalid Message Channel EIP by comparing their core components.

Invalid Message Channel EIP (Figure 1)	Invalid Message Channel Example Scenario (Figure 2)
Sender	Stock Quote Client
Channel	Target Endpoint
Receiver	Stock Quote Service Instance
Invalid Message Channel	FaultSequence

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- Invalid Message Chanel Proxy-->
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="InvalidMessageChannelProxy">
        <target>
      <endpoint>
                  <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
        </endpoint>
        <inSequence>
              <log level="full" />
        </inSequence>
        <outSequence>
                <log level="full" />
        </outSequence>
     <faultSequence>
          <log level="full">
                    <property name="MESSAGE" value="Failure Message..."/>
                    <property name="ERROR_CODE"
expression="get-property('ERROR_CODE')"/>
                    <property name="ERROR_MESSAGE"
expression="get-property('ERROR_MESSAGE')"/>
              </log>
          <drop />
       </faultSequence>
        </target>
    <publishWSDL uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
    </proxy>
</definitions>
```

When the server receives a request and the endpoint referred through the ESB is unavailable, the server triggers an error message. The ESB diverts the response to the invalid message channel.

Simulating the sample scenario

1. Pass the following message to the WSO2 ESB. You can use SoapUI or WSO2 ESB's Try It tool. To invoke the Try It tool, log into the ESB management console, navigate to the **Services** menu under the **Main** menu and select the **List** sub menu. Then select **InvalidMessageChannelProxy** and pass the following request to the ESB.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soap:Header/>
   <soap:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <xsd:symbol>Foo</xsd:symbol>
         </ser:request>
      </ser:getQuote>
   </soap:Body>
</soap:Envelope>
```

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **Proxy Service** [line 3 of ESB config] - Defines the proxy service `InvalidMessageChannelProxy` with a target endpoint to the back end service.
- **faultSequence** [line 14 of ESB config] - Defines a fault sequence to execute in the event of a fault. It acts as the Invalid Message Channel for this EIP. In this example configuration, we log the fault as an error, but you can place any of the usual mediators inside this sequence. For example, you could pass the invalid message to another service or back to the client.

Dead Letter Channel

This section explains, through an example scenario, how the Dead Letter Channel EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Dead Letter Channel

The Dead Letter Channel (DLC) EIP outlines how messaging systems can handle messages that cannot be delivered to the recipient. Due to system/network failures or failures at the recipient's end, messages sometimes do not get delivered to the target. In such cases, the messaging system can deliver the message to a DLC. Different mechanisms implemented in the DLC take care of delivering the dead message to the target. One method is periodically retrying to send the message to the recipient over a defined period of time. Persistence of the dead message is another option, which ensures that the dead messages are delivered to the receivers once the system is rebooted, even if the messaging system fails.

For more information, refer to http://www.eaipatterns.com/DeadLetterChannel.html.

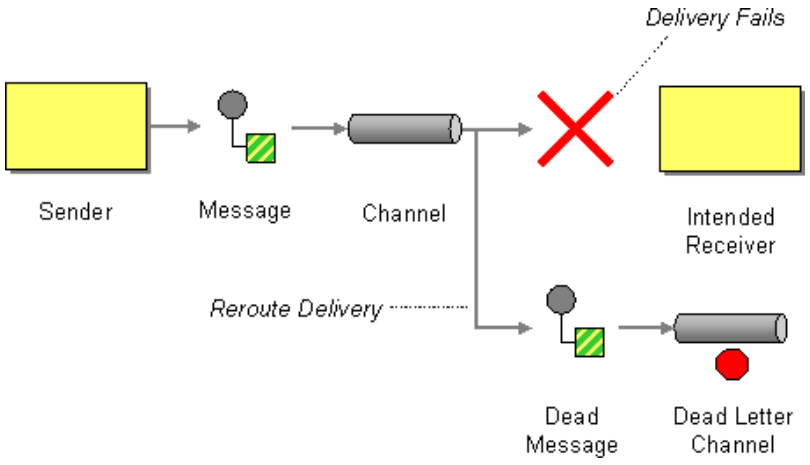


Figure 1: Dead Letter Channel EIP

Example scenario

This example takes a proxy service called `StockQuoteProxy`, which fronts a service by the name `SimpleStockQuoteService`. As long as the `SimpleStockQuoteService` is running, the clients calling `StockQuoteProxy` service get responses. But if the `SimpleStockQuoteService` is down or a failure occurs while trying to send the message to the `SimpleStockQuoteService`, the `faultSequence` of the `StockQuoteProxy` will get invoked, and the message will be forwarded to the dead letter channel.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

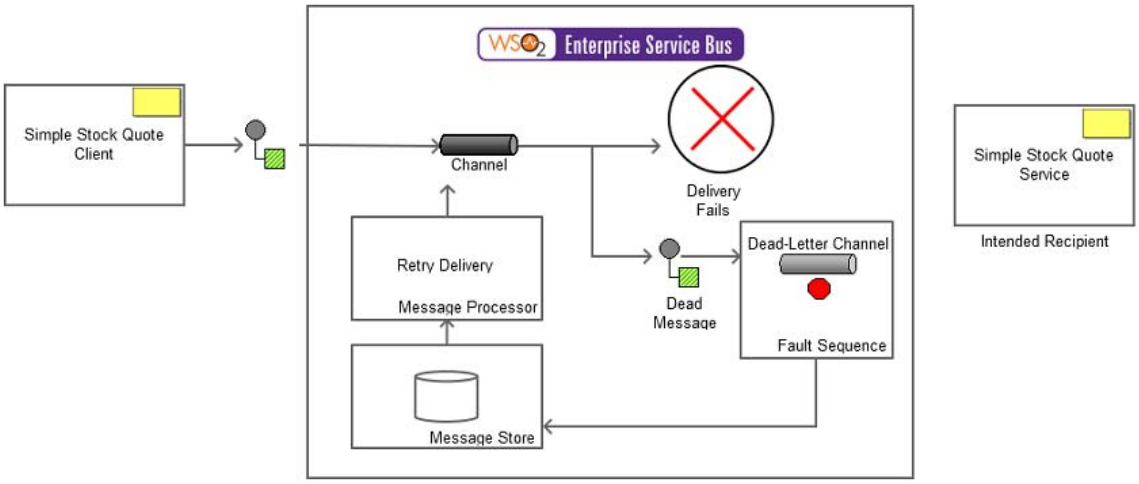


Figure 2: Example Scenario of the Dead Letter Channel EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Dead Letter Channel EIP by comparing their core components. We use three constructs of WSO2 ESB to implement the Dead Letter Channel EIP.

Dead Letter Channel EIP (Figure 1)	Dead Letter Channel Example Scenario (Figure 2)
Sender	Stock Quote Client
Intended Recipient	Stock Quote Service Instance
Dead Letter Channel	Store mediator, Message stores, Message processors
Intended Receiver	Simple Stock Quote Service

The diagram below depicts the DLC architecture in WSO2 ESB.

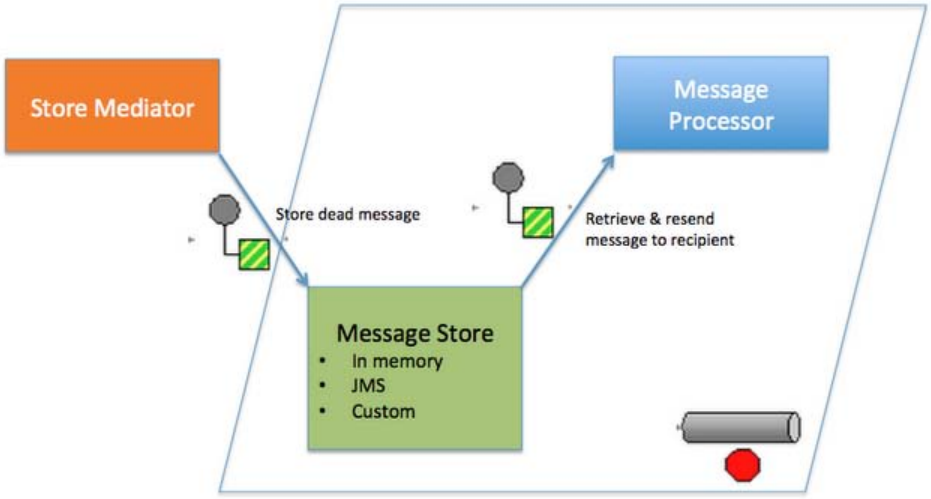


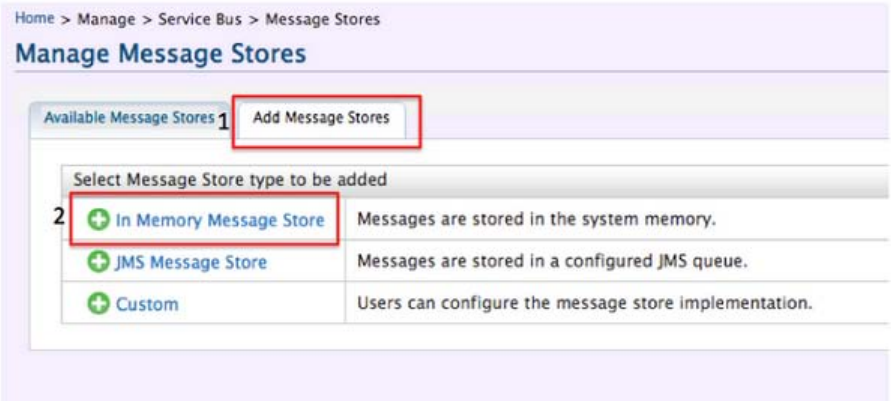
Figure 3: DLC architecture in WSO2 ESB

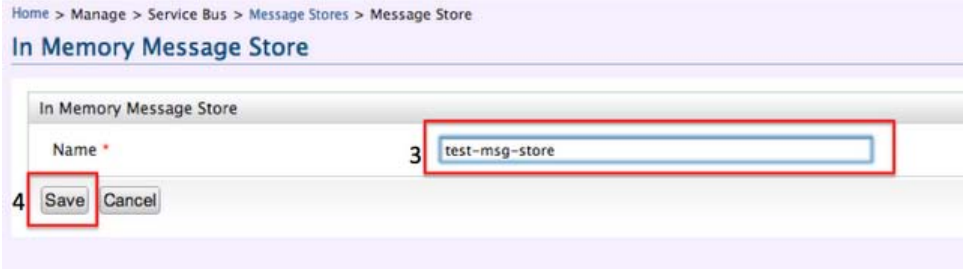
The store mediator stores the dead message in the specified message store. The message processor retrieves stored messages from its associated message store and tries to resend those messages to the target receiver. The message store and message processor combination acts as the dead letter channel.

Environment setup

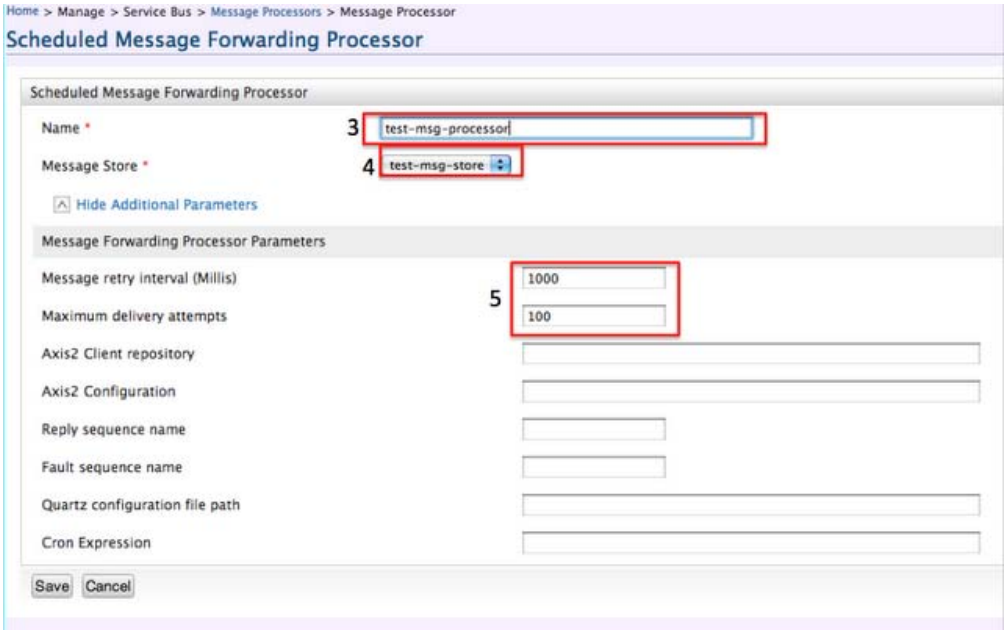
WSO2 ESB provides two message store types: in memory and JMS. Users can also define their own custom message store implementations.

1. Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Service Bus** menu in the **Main** menu. Then, select the **Message Stores** s ub menu.
2. Create a new message store. In this example, we create `test-msg-store`.

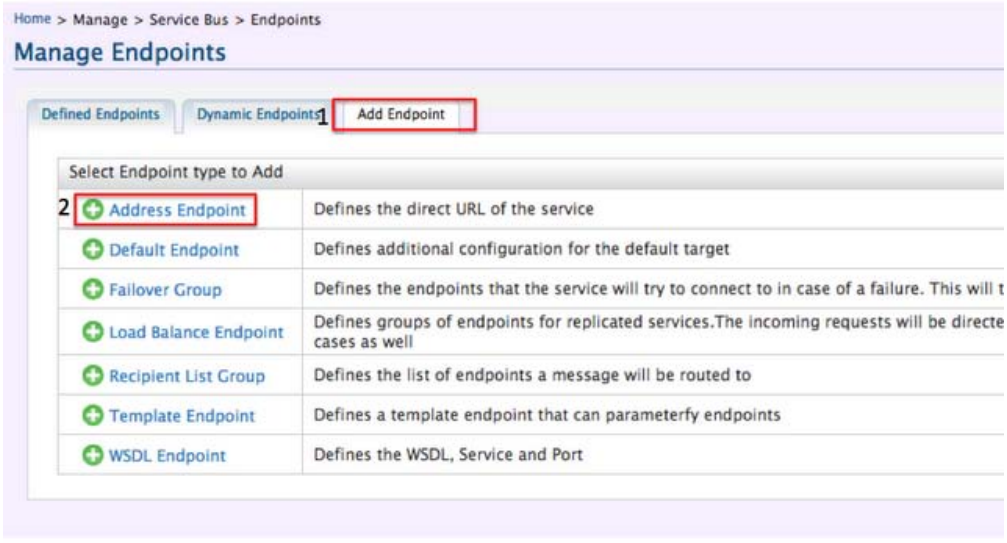


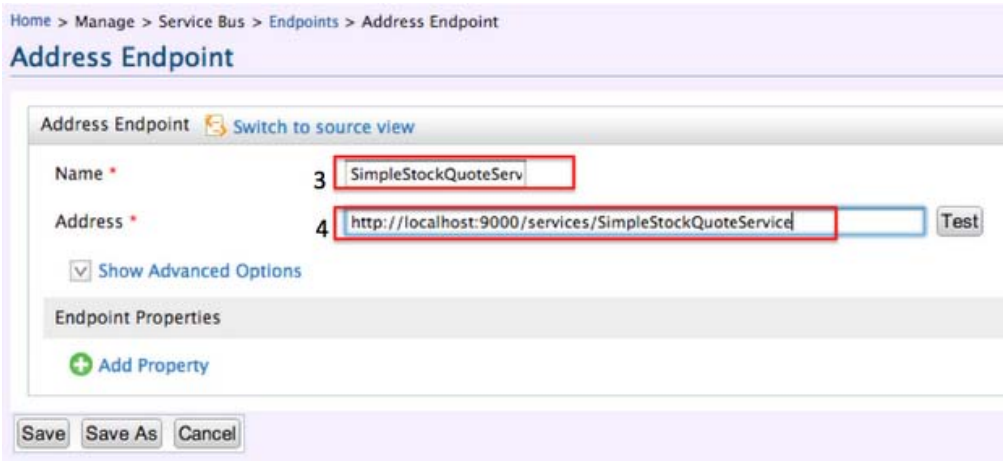


3. Define a message processor called `test-msg-processor` as follows:



4. Define an endpoint called `SimpleStockQuoteService`, and set it as the `target.endpoint` property. The store mediator uses it when storing the message into the message store.





ESB configuration

Navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockeQuoteProxy"
transports="https,http" statistics="disable" trace="disable" startOnLoad="true">
   <target>
      <inSequence>
         <log level="full" />
      </inSequence>
      <outSequence>
         <log level="full">
            <property name="MSG" value="Response...." />
         </log>
         <send />
      </outSequence>
      <faultSequence>
         <log level="full">
            <property name="MSG" value="++++++++++FAULT----------...." />
         </log>
         <property name="target.endpoint" value="SimpleStockQuoteService" />
         <store messageStore="test-msg-store" />
      </faultSequence>
      <endpoint>
         <address uri="http://localhost:9000/services/SimpleStockQuoteService" />
      </endpoint>
   </target>
   <publishWSDL uri="http://localhost:9000/services/SimpleStockQuoteService?wsdl" />

   <description></description>
</proxy>
```

The full WSO2 ESB configuration for this example is as follows.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
```

```
        <parameter name="cachableDuration">15000</parameter>
    </registry>
    <proxy name="StockQuoteProxy" transports="https http" startOnLoad="true"
trace="disable">
        <description/>
        <target>
            <endpoint>
                <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
            </endpoint>
            <inSequence>
                <log level="full"/>
            </inSequence>
            <outSequence>
                <log level="full">
                    <property name="MSG" value="Response...."/>
                </log>
                <send/>
            </outSequence>
            <faultSequence>
                <log level="full">
                    <property name="MSG" value="++++++++++FAULT----------...."/>
                </log>
                <property name="target.endpoint" value="SimpleStockQuoteService"/>
                <store messageStore="test-msg-store"/>
            </faultSequence>
        </target>
        <publishWSDL
uri="http://localhost:9000/services/SimpleStockQuoteService?wsdl"/>
    </proxy>
    <endpoint name="SimpleStockQuoteService">
        <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
    </endpoint>
    <sequence name="fault">
        <log level="full">
            <property name="MESSAGE" value="Executing default 'fault' sequence"/>
            <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
            <property name="ERROR_MESSAGE"
expression="get-property('ERROR_MESSAGE')"/>
        </log>
        <drop/>
    </sequence>
    <sequence name="main">
        <in>
            <log level="full"/>
            <filter source="get-property('To')" regex="http://localhost:9000.*">
                <send/>
            </filter>
        </in>
        <out>
            <send/>
        </out>
        <description>The main sequence for the message mediation</description>
    </sequence>
    <messageStore name="test-msg-store"/>
    <messageProcessor
class="org.apache.synapse.message.processors.forward.ScheduledMessageForwardingProce
ssor" name="test-msg-processor" messageStore="test-msg-store">
        <parameter name="interval">1000</parameter>
```

```
        <parameter name="max.deliver.attempts">100</parameter>
    </messageProcessor>
</definitions>
```

Simulating the sample scenario

1. Start the sample Axis2 server with `SimpleStockQuoteService` deployed. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.
2. Use SoapUI or WSO2 ESB's Try It tool to send the following request to the `StockQuoteProxy` service. For information about the Stock Quote client, refer to the section Sample Clients in the WSO2 ESB documentation.

3. A message similar to the one below appears in the simple Axis server.

```
Sun May 20 13:47:03 IST 2012 samples.services.SimpleStockQuoteService :: Generating quote for : IBM
```

4. Stop the Axis server, and resend the same request. Note that the message sending fails, and the message processor tries to resend the message every few seconds.
5. Restart the Axis server, and note that the message will be delivered to `SimpleStockQuoteService` once the server is running.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **faultSequence** [line 20 in ESB config] - The fault sequence. The proxy service `StockQuoteProxy` runs in the event of a fault or error.
- **store** [line 25 in ESB config] - The store mediator loads the message store defined in line 53.
- **messageStore** [line 53 in ESB config] - Defines the message store to use, which we created using the ESB management console in step 2 above.
- **messageProcessor** [line 54 in ESB config] - The `messageProcessor` defines the processing algorithm, the period of time to retry sending messages in case of a failure, and the maximum number of retry attempts.

Guaranteed Delivery

This section explains, through an example scenario, how the Guaranteed Delivery EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Guaranteed Delivery
- Example scenario
    - Environment setup
    - ESB configuration

- Simulating the sample scenario
- How the implementation works

Introduction to Guaranteed Delivery

The Guaranteed Delivery EIP ensures safe delivery of a message by storing it locally and transmitting it to the receiver's data store. Even when the receiver is offline, the EIP ensures that the message goes through when the receiver comes online. For more information, refer to http://www.eaipatterns.com/GuaranteedMessaging.html.

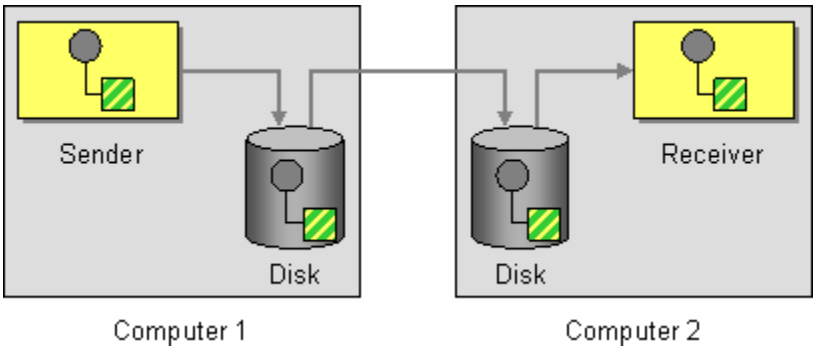


Figure 1: Guaranteed Delivery EIP

Example scenario

This example is a stock quote service where a stock quote request is sent to a specific endpoint when the receiver is offline. An Axis2 server acts as the receiver. The ESB stores the request message in a JMS message store provided by the ESB. The ESB periodically checks whether the receiver is online using a Message Forwarding Processor and delivers the message to the endpoint when the receiver comes online.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

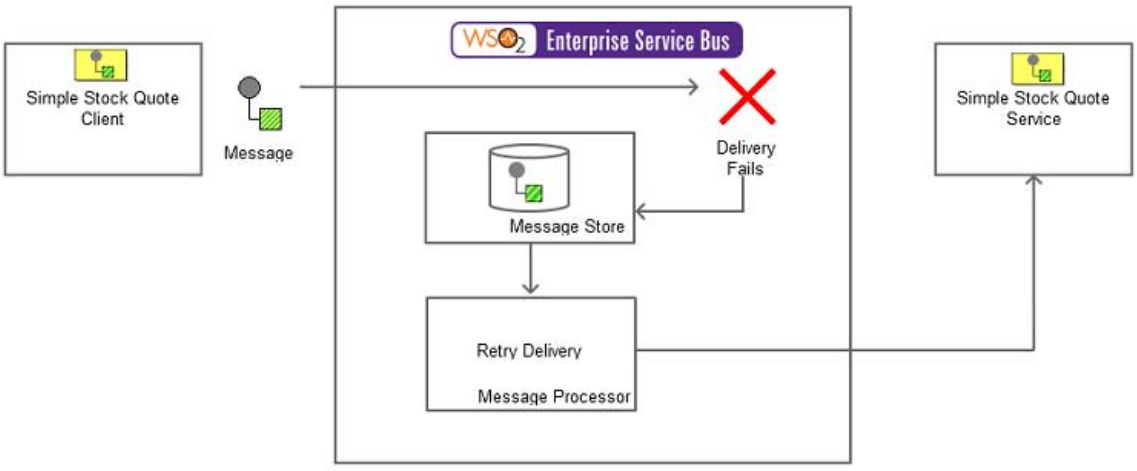


Figure 2: Example Scenario of the Guaranteed Delivery EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Guaranteed Delivery EIP by comparing their core components.

Guaranteed Delivery EIP (Figure 1)	Guaranteed Delivery Example Scenario (Figure 2)
Sender	Stock Quote Client
Store	Message Store
Receiver	Stock Quote Service Instance

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.
3. Start an instance of the ActiveMQ server. In this example, we use an ActiveMQ JMS service provider for message persistence.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="GuranteedDeliveryProxy"
          transports="http https"
          startOnLoad="true">
      <target>
         <inSequence>
            <sequence key="delivery_seq"/>
         </inSequence>
         <outSequence>
            <send/>
         </outSequence>
      </target>
   </proxy>
   <endpoint name="StockReqEndPoint">
      <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
   </endpoint>
   <sequence name="delivery_seq" onError="delivery_fail">
      <enrich>
         <source type="envelope" clone="true"/>
         <target type="property" property="mssg"/>
      </enrich>
      <send>
         <endpoint key="StockReqEndPoint"/>
      </send>
   </sequence>
   <sequence name="delivery_fail">
      <log level="full"/>
      <enrich>
         <source type="property" clone="true" property="mssg"/>
         <target type="envelope"/>
      </enrich>
      <property name="target.endpoint" value="StockReqEndPoint"/>
      <store messageStore="JMStore"/>
```

```
   </sequence>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
      <drop/>
   </sequence>
   <messageStore
class="org.wso2.carbon.message.store.persistence.jms.JMSMessageStore"
                 name="JMStore">
      <parameter
name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFa
ctory</parameter>
      <parameter name="store.jms.cache.connection">false</parameter>
      <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
      <parameter name="store.jms.JMSSpecVersion">1.1</parameter>
   </messageStore>
   <messageProcessor
class="org.apache.synapse.message.processors.forward.ScheduledMessageForwardingProce
ssor"
                    name="ScheduledProcessor"
                    messageStore="JMStore">
```

```
        <parameter name="interval">10000</parameter>
    </messageProcessor>
</definitions>
```

Simulating the sample scenario

1. Stop the Axis2 server instance, and send a message to the ESB requesting a stock quote with the following command.
   `ant stockquote -Dtrpurl=http://localhost:8280/services/GuranteedDeliveryProxy`
2. Start the Axis2 server instance. Observe in the server console that the message that you sent when it was offline is now successfully delivered.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration above.

- **proxy** [line 2 in ESB config] - This service allows you to abstract the routing logic from the client. Whatever the request is, the client sends it only to the exposed service.
- **inSequence** [line 6 in ESB config] - When the service is invoked through the client, this sequence receives the message and sends it to the routing logic.
- **sequence** [line 17 in ESB config] - The sequence mediator defines a sequence block, callable by its key (defined in the name attribute).
- **enrich** [line 18 in ESB config] - The enrich mediator processes messages based on the source configuration and performs the action on the target configuration.
- **store** [line 33 in ESB config] - Saves the message using the message store defined by the name JMStore.
- **messageStore** [line 47 in ESB config] - Defines the message store to use and the parameters used to connect to the message store. In this example, the connection is made to an external JMS store.
- **messageProcessor** [line 54 in ESB config] - Defines the message processing algorithm to use, the period of time to retry sending messages in case of a failure, and the maximum number of retry attempts.

Messaging Bridge

This section explains, through an example scenario, how the Guaranteed Delivery EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Messaging Bridge
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Messaging Bridge

The Messaging Bridge EIP facilitates the connection between messaging systems and replicates messages between them by transforming message formats from one system to the other. For more information, refer to http://www.eaipatterns.com/MessagingBridge.html.

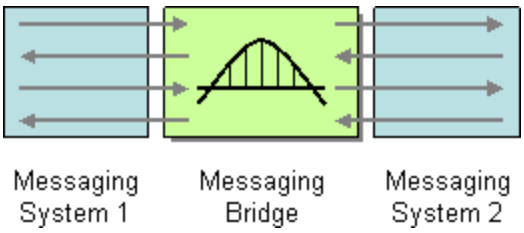


Figure 1: Messaging Bridge EIP

Example scenario

An enterprise might use more than one messaging system in communication. This example illustrates the ESB as a message bridge, which accepts a REST message from the client, transforms the REST message to SOAP format, and sends the SOAP message to the back-end Axis2 server. To transform the sent REST request to a SOAP message, we use the API functionality of the ESB. It restructures the message to a REST format using the WSO2 payload factory mediator.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

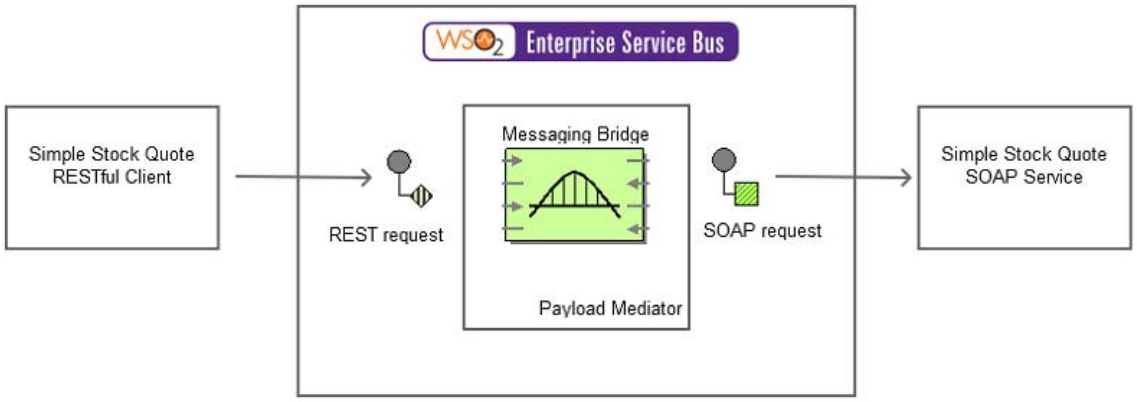


Figure 2: Example Scenario of the Messaging Bridge EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Messaging Bridge EIP by comparing their core components.

Messaging Bridge EIP (Figure 1)	Messaging Bridge Example Scenario (Figure 2)
Messaging System1	Stock Quote client
Messaging Bridge	PayloadFactory Mediator (You can add any transformation mediator here. Also see Message Translator)
Messaging System2	Stock Quote service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- Message Translator-->
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <api name="MessageTranslate" context="/stockquote">
      <resource methods="GET" uri-template="/view/{symbol}">
         <inSequence>
            <payloadFactory>
               <format>
                  <m0:getQuote xmlns:m0="http://services.samples">
                   <m0:request>
                      <m0:symbol>$1</m0:symbol>
                   </m0:request>
                </m0:getQuote>
               </format>
               <args>
                  <arg expression="get-property('uri.var.symbol')"/>
               </args>
            </payloadFactory>
            <send>
               <endpoint>
                  <address
uri="http://localhost:9000/services/SimpleStockQuoteService" format="soap11"/>
               </endpoint>
            </send>
         </inSequence>
         <outSequence>
            <send/>
         </outSequence>
      </resource>
   </api>
</definitions>
```

Simulating the sample scenario

1. Pass the following request to the ESB using the cURL client.

   ```
   curl -v http://127.0.0.1:8280/stockquote/view/IBM
   ```

2. You can use TCPMon to see the type of the message and its message format:

   ```
   GET /stockquote/view/IBM HTTP/1.1
   User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1
   zlib/1.2.3.4 libidn/1.23 librtmp/2.3
   Host: 127.0.0.1:8281
   Accept: */*
   ```

3. After sending the request, notice that the Axis2 server has logged the message and accepted the request.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **Payload Factory** [line 6 in ESB config] - Builds the necessary SOAP request for the back-end service. It uses the value of the HTTP GET variable (from the REST request made to the ESB).
- **args** - [line 14 in ESB config] A list of arguments that will be concatenated in the same order inside the format tags (in a printf style).

Message Bus

This section explains how the Message Bus EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Message Bus
- How WSO2 ESB implements the EIP

Introduction to Message Bus

The Message Bus EIP enables separate applications to work together in a decoupled manner so that applications can be easily added or removed without affecting each other. This approach makes maintenance and testing smoother, since editing or removing an application will not affect the functionality of any other application. For more information, refer to http://www.eaipatterns.com/MessageBus.html.

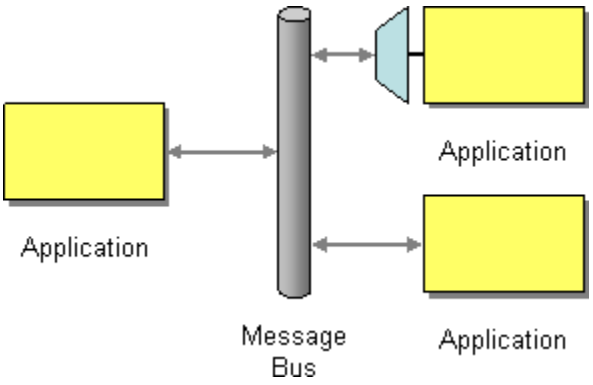


Figure 1: Message Bus EIP

How WSO2 ESB implements the EIP

The architecture of WSO2 ESB is discussed in ESB Architecture in the WSO2 ESB documentation. It illustrates how application logic is layered, and how each component of the application logic is separated as a mediator, allowing message processing to be executed in a decoupled manner. The mediation process is explained in the Message Mediation section of the WSO2 ESB documentation.

A sample ESB configuration is illustrated below:

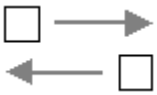
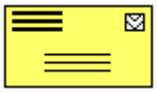
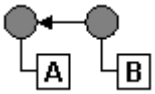
```
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <!-- filtering of messages with XPath and regex matches -->
    <filter source="get-property('To')" regex=".*/StockQuote.*">
        <send>
            <endpoint>
                <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
            </endpoint>
        </send>
        <drop/>
    </filter>
    <send/>
</definitions>
```

According to the configuration above, the ESB routes an incoming message to a back-end server if the conditions in the filter section are met. Notice how the application's logic is decoupled. It uses one component for filtering, and another to send a message to the endpoint. If you were to decide to remove the filtering step, you could remove the filter mediator segment from the XML without affecting the application's logic for sending the message to the back-end server.

Message Construction

Message construction involves the architectural patterns of various constructs, functions, and activities involved in creating and transforming a message between applications.

This chapter introduces message construction patterns and how each of them can be simulated using WSO2 ESB.

	Command Message	How messaging can be used to invoke a procedure in another application.
	Document Message	How messaging can be used to transfer data between applications.
	Event Message	How messaging can be used to transmit events from one application to another.
	Request-Reply	How an application that sends a message gets a response from the receiver.
	Return Address	How a replier knows where to send the reply.
	Correlation Identifier	How a requester that has received a reply knows which request the reply is for.
	Message Sequence	How messaging can transmit an arbitrarily large amount of data.
	Message Expiration	How a sender indicates when a message should be considered stale and therefore should not be processed.
	Format Indicator	How a message's data format can be designed to allow for possible future changes.

Command Message

This section explains, through an example scenario, how the Command Message EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Command Message
- Example scenario
    - Environment setup
    - ESB configuration
    - Simulating the sample scenario
    - How the implementation works

Introduction to Command Message

The Command Message EIP allows you to use messaging to invoke a procedure in another application. For more

information, refer to http://www.eaipatterns.com/CommandMessage.html.

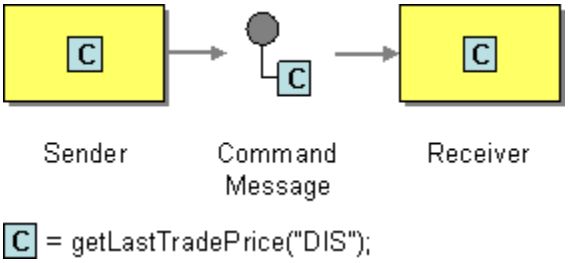


Figure 1: Command Message EIP

Example scenario

This example demonstrates how WSO2 ESB uses messaging to invoke functionality provided by an application, in this case a stock quote service. A command message can be in any form, including a JMS serialized object or a text message in the form of an XML or SOAP request. In this example, the ESB will pass the message as a document to a sample Axis2 server and invoke the operation directly using the callout mediator.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

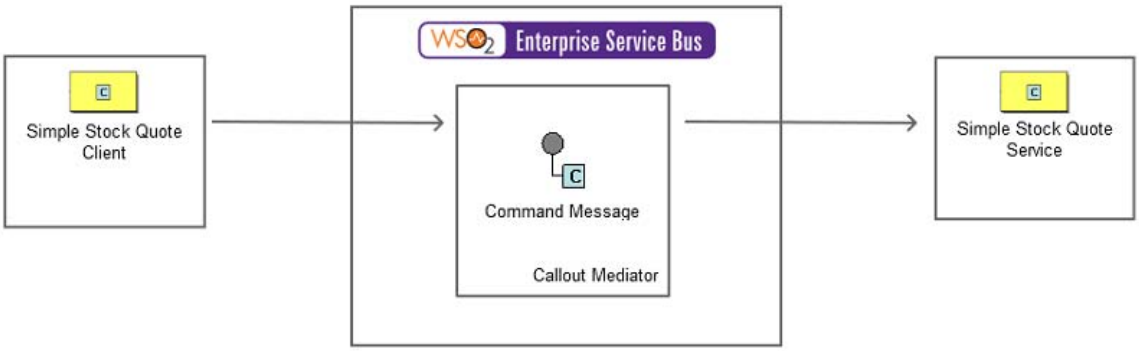


Figure 2: Example Scenario of the Command Message EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Command Message EIP by comparing their core components.

Command Message EIP (Figure 1)	Command Message Example Scenario (Figure 2)
Sender	Stock Quote Client
Command Message	Callout Mediator
Receiver	Stock Quote Service Instance

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <callout serviceURL="http://localhost:9000/services/SimpleStockQuoteService"
               action="urn:getQuote">
         <source xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
                 xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
                 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]"/>
         <target xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
                 xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
                 xpath="s11:Body/child::*[fn:position()=1] |
s12:Body/child::*[fn:position()=1]"/>
      </callout>
      <property name="RESPONSE" value="true"/>
      <header name="To" action="remove"/>
      <send/>
      <drop/>
   </sequence>
</definitions>
```

Simulating the sample scenario

1. Send a request using the Stock Quote client to WSO2 ESB in the following manner. Information about the `Stock Quote` client and its operation modes are discussed in the Sample Clients section in the WSO2 ESB documentation.

   ```
   ant stockquote -Dtrpurl=http://localhost:8280
   ```
2. The client receives the stock quote as the response.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **callout** [line 11 in ESB config] - The callout mediator specifies a particular method to invoke in the back-end service. This invocation is blocking.
- **source** [line 13 in ESB config] - The source specifies the payload for the method invocation using xPath expressions.
- **target** [line 16 in ESB config] - The target specifies the location where the response should be concatenated.

Document Message

This section explains, through an example scenario, how the Document Message EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Document Message

The Document Message EIP is used to reliably transfer a data structure between applications. The Command Message EIP allows you to invoke only a specific client through the ESB, while the Document Message EIP sends the entire data unit to the receiver. For more information, refer to http://www.eaipatterns.com/DocumentMessage.html.

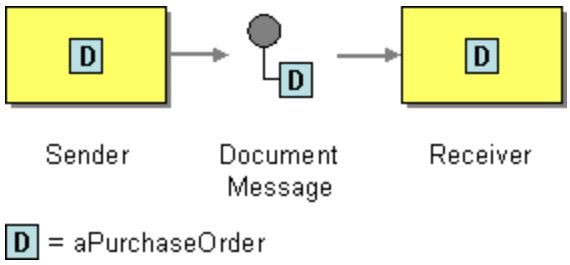


Figure 1: Document Message EIP

Example scenario

This example demonstrates WSO2 ESB transmitting an entire message from a client to a sample Axis2 server as a document message, which the Axis2 server processes so it can identify which operation to invoke.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

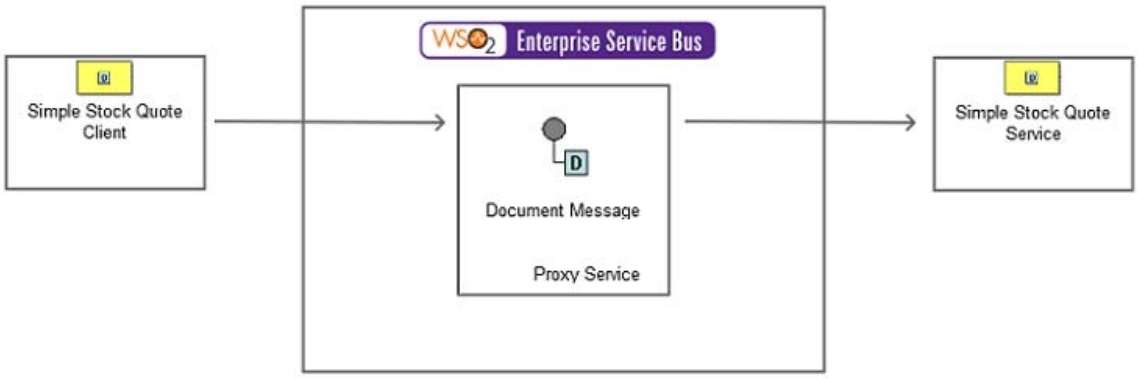


Figure 2: Example Scenario of the Document Message EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Document Message EIP by comparing their core components.

Document Message EIP (Figure 1)	Document Message Example Scenario (Figure 2)
Sender	Stock Quote Client

Document Message	Proxy Service
Receiver	Simple Stock Quote Service Instance

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB Configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="DocumentMessageProxy"
          transports="https http"
          startOnLoad="true"
          trace="disable">
      <target>
         <inSequence>
            <send>
               <endpoint>
                  <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
               </endpoint>
            </send>
         </inSequence>
         <outSequence>
            <send/>
         </outSequence>
      </target>
   </proxy>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
      <send/>
   </sequence>
</definitions>
```

Simulating the sample scenario

Using a SOAP client (such as SoapUI), send a request to the ESB server to invoke the `DocumentMessageProxy`. Note that the entire request will be passed to the back-end Axis2 Server, and the client will receive the response in return.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above .

- **Proxy Service** [line 3 of ESB config] - The proxy service takes requests and forwards them to the back-end service, abstracting the routing logic from the client. In this example scenario, the proxy service just forwards requests to the back-end service following the Document Message EIP style.

Event Message

This section explains, through an example scenario, how the Command Message EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Event Message
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Event Message

The Event Message EIP is used for reliable, asynchronous event notification between applications. For more information, refer to http://www.eaipatterns.com/EventMessage.html.

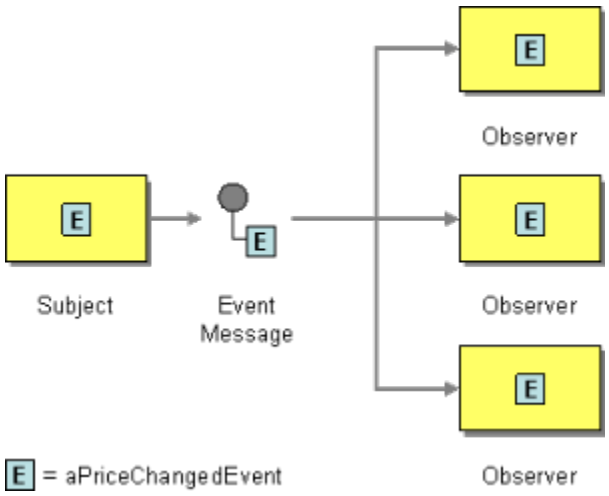


Figure 1: Event Message EIP

Example scenario

When a subject has an event to be announced, it will crate an event object, wrap it in a message, and send it to a set of subscribers. This example scenario depicts several Axis2 server instances as subscribers. When a message arrives to the ESB, it will be transmitted through the event mediator to each of these server instances that acts as a subscriber.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

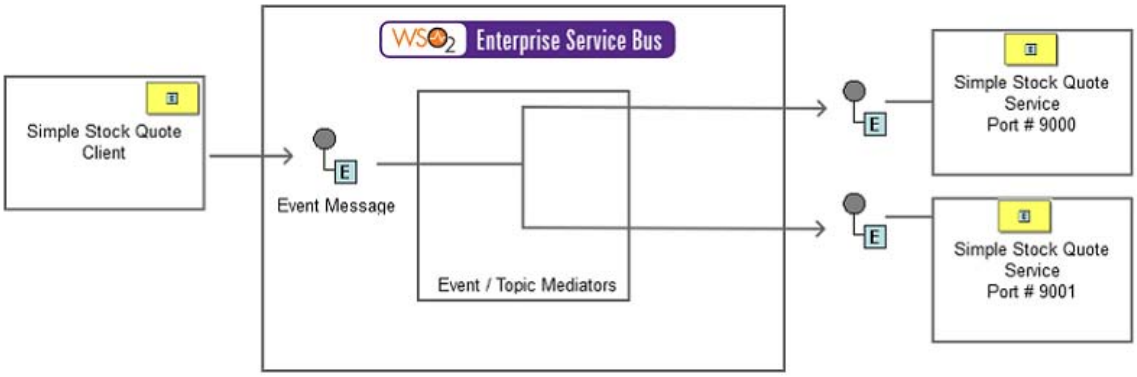


Figure 2: Example Scenario of the Event Message EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Event Message EIP by comparing their core components.

Event Message EIP (Figure 1)	Event Message Example Scenario (Figure 2)
Subject	Stock Quote Client
Event Message	Event, Topic
Observer	Stock Quote Service Instance

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two sample Axis2 server instances on port 9000 and 9001. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.
3. Follow the steps below to create an event.
    - Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`).
    - Select the **Topics** menu from the **Main** menu, and then select the **Add** sub menu.
    - Enter the name `stockquote` for the topic and then click **Add Topic**.
    - In the Topic Browser tree, click the newly created `stockquote` topic and then click **Subscribe** to create a static subscription.
    - Enter the value `http://localhost:9000/services/SimpleStockQuoteService` in the **Event Sink URL** field and click **Subscribe**.
    - Repeat these steps to add another subscriber in port 9001.

ESB configuration

In the ESB's Management Console, navigate to **Main** menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
      <event topic="EventMessage"/>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send the following message from a SOAP client like SoapUI to the ESB.

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

Observe the two Axis2 server instances. Both instances will receive the request, which was sent from the client.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **Event** [line 12 of ESB config] - Sends incoming events to the topics that you created earlier.

Request-Reply

This section explains, through an example scenario, how the Request-Reply Message EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Request-Reply
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario

- How the implementation works

Introduction to Request-Reply

The Request-Reply EIP facilitates two-way communication by ensuring that a sender gets a response or reply back from the receiver after sending a request message. This pattern sends a pair of Request-Reply messages, each on its own channel. For more information, refer to http://www.eaipatterns.com/RequestReply.html.

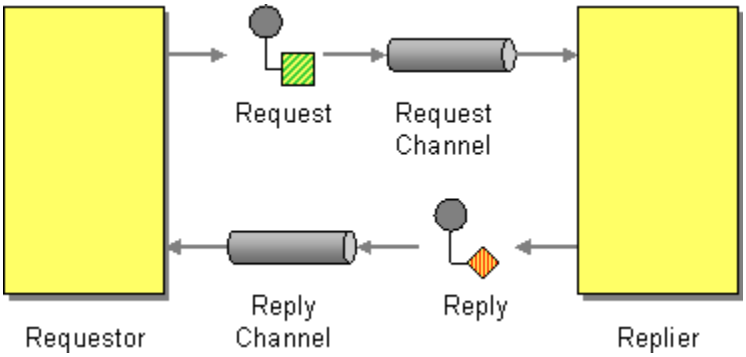


Figure 1: Request-Reply EIP

Example scenario

The example scenario illustrates how a request to a service is made through one channel, and the response from the service is returned to the requester on a separate channel.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

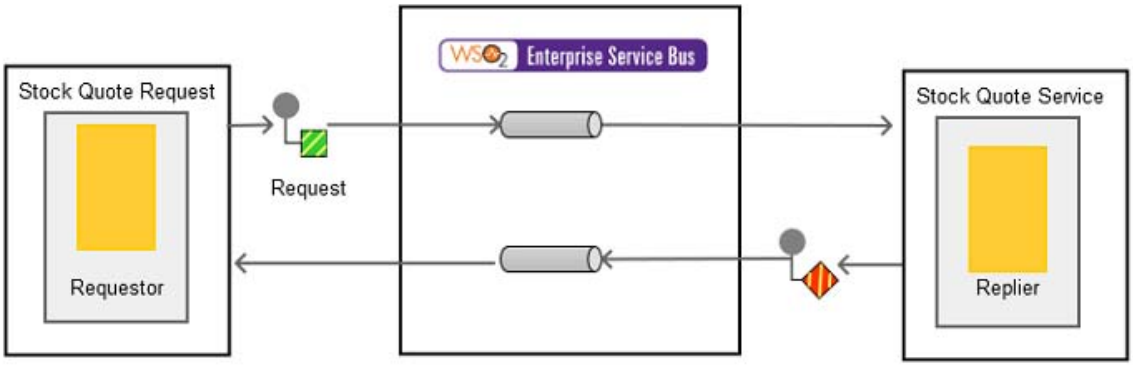


Figure 2: Example Scenario of the Request-Reply EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Request-Reply EIP by comparing their core components.

Request-Reply EIP (Figure 1)	Request-Reply Example Scenario (Figure 2)
Requestor	Simple Stock Quote Client
Request Channel	Send, Endpoint
Replier	Stock Quote Service Instance
Reply Channel	Send, Endpoint

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

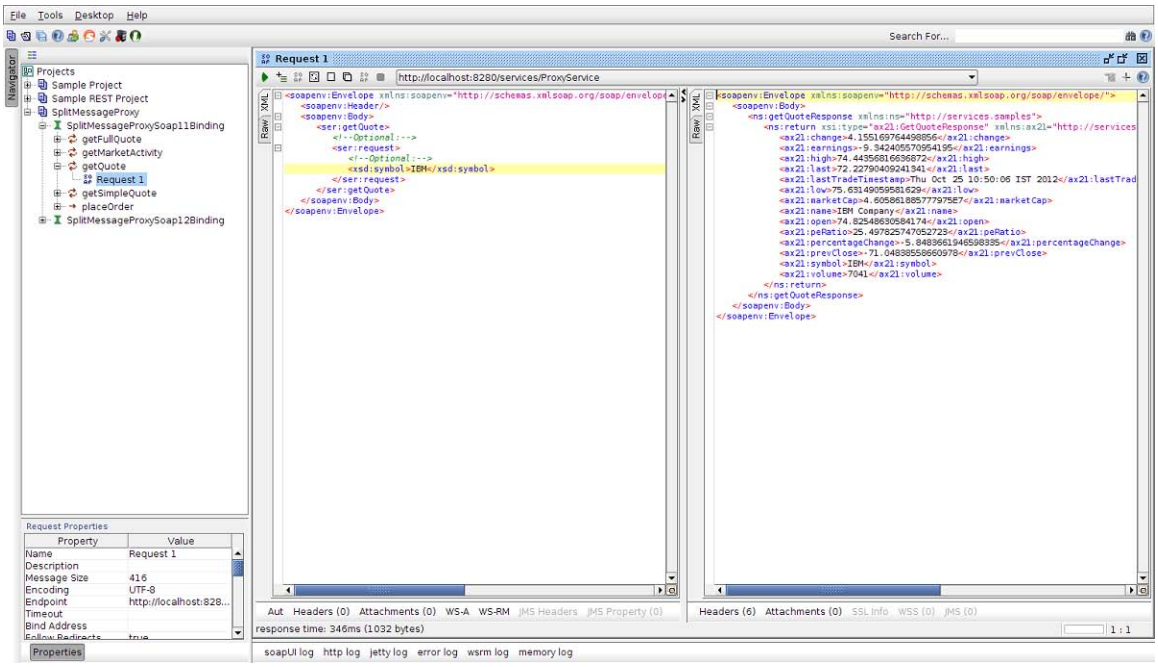
ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<proxy name="ProxyService" transports="http https" startOnLoad="true">
      <target>
         <inSequence>
            <send>
               <endpoint>
                  <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
               </endpoint>
            </send>
         </inSequence>
         <outSequence>
            <send/>
         </outSequence>
      </target>
   </proxy>
```

Simulating the sample scenario

Use a SOAP client like SoapUI to invoke the above proxy service. The following image illustrates how a response is generated to a request made by the client.



How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **inSequence** [line 3 in ESB config] - When a client sends a message, it is picked up by the `inSequence`.
- **send** [line 4 in ESB config] - The `send` mediator sends the message to the endpoint running on port 9000.
- **outSequence** [line 10 in ESB config] - The processed response from the endpoint will be sent back to the client through the `outSequence`. The `send` mediator inside the `outSequence` will direct the message back to the requesting client, which is on a channel separate from the requesting channel.

Return Address

This section explains, through an example scenario, how the Return Address Message EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Return Address

The Return Address EIP facilitates adding a return address to a request message, which indicates where to send the reply message. For more information, refer to http://www.eaipatterns.com/ReturnAddress.html.

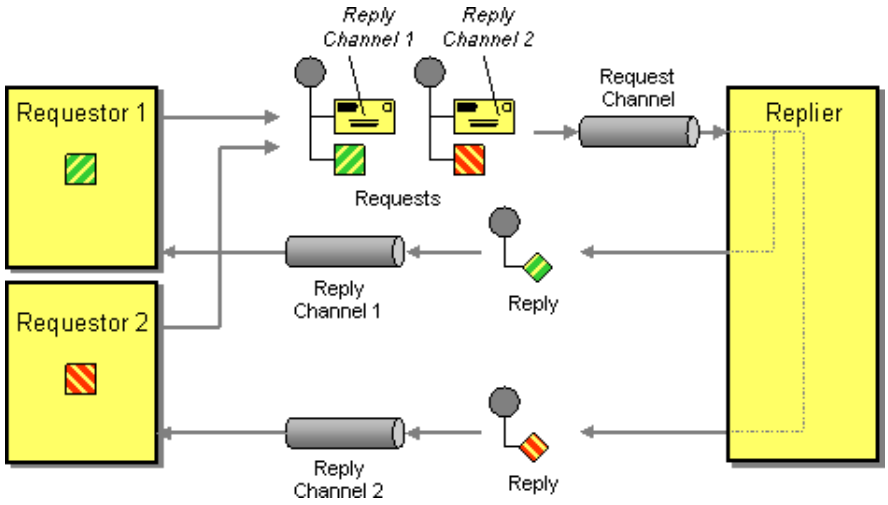


Figure 1: Return Address EIP

Example scenario

This example is a stock quote service where a client sends a stock quote request to WSO2 ESB with a return address embedded in the message header, which will indicate to the replier where the response message should be sent.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

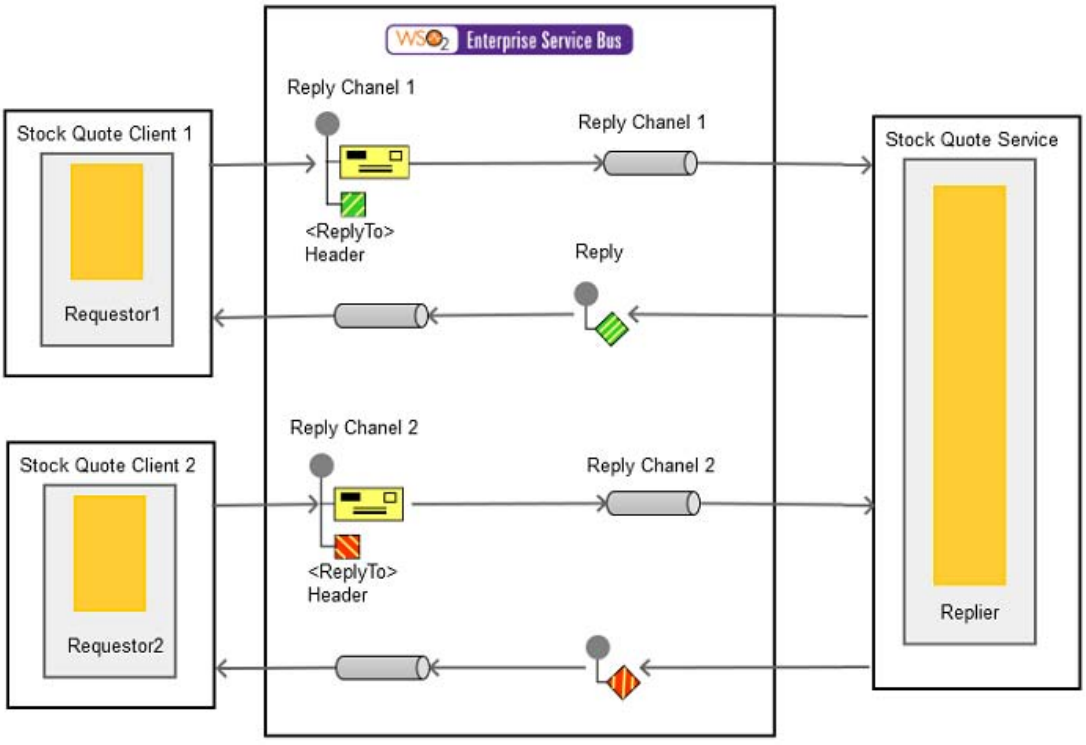


Figure 2: Example Scenario of the Return Address EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Return Address EIP by comparing their core components.

Return Address EIP (Figure 1)	Return Address Example Scenario (Figure 2)

Requestor 1	Stock Quote Client Instance
Request Channel 1	Send Mediator
Requestor 2	Stock Quote Client Instance
Request Channel 2	Send Mediator
Replier	Stock Quote Service Instance

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="main">
      <in>
         <log level="full"/>
         <send/>
      </in>
      <out>
         <log level="full"/>
         <send/>
      </out>
   </sequence>
</definitions>
```

Simulating the sample scenario

1. Send a request using the Stock Quote client to WSO2 ESB in the following manner. For information about the Stock Quote client, refer to the Sample Clients section in the WSO2 ESB documentation.

   ```
   ant stockquote -Daddurl=http://localhost:9000/soap/SimpleStockQuoteService
   -Dtrpurl=http://localhost:8280/ -Dmode=dualquote -Dsymbol=Foo
   ```

2. If you use TCPmon to analyze the message passing, you will notice that the client sends the following message. Note that in line 4, the WS-Addressing `ReplyTo` header is set to a service called anonService2. Since the reply is made to this service on a separate channel, the client will receive no response.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <soapenv:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">

<wsa:To>http://localhost:9000/service/SimpleStockQuoteService</wsa:To>
         <wsa:ReplyTo>

<wsa:Address>http://10.150.3.53:8200/axis2/services/anonService2/</wsa:Address
>
         </wsa:ReplyTo>

<wsa:MessageID>urn:uuid:9aa8e783-2eb7-4649-9d36-a7fb3ad17abd</wsa:MessageID>
         <wsa:Action>urn:getQuote</wsa:Action>
      </soapenv:Header>
      <soapenv:Body>
         <m0:getQuote xmlns:m0="http://services.samples">
            <m0:request>
               <m0:symbol>Foo</m0:symbol>
            </m0:request>
         </m0:getQuote>
      </soapenv:Body>
   </soapenv:Envelope>
```

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **send** [line 10 in ESB config] - The `send` mediator forwards messages to the address implied in the `ReplyTo` header field by default, unless it is made explicit that the reply should go to a specific address by using an endpoint mediator.

Correlation Identifier

This section explains how the Correlation Identifier EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Correlation Identifier
- How WSO2 ESB implements the EIP

Introduction to Correlation Identifier

The Correlation Identifier EIP facilitates a unique identifier that indicates which request message a given reply is for. It enables a requester that has received a reply to know which request the reply is for. For more information, refer to http://www.eaipatterns.com/CorrelationIdentifier.html.

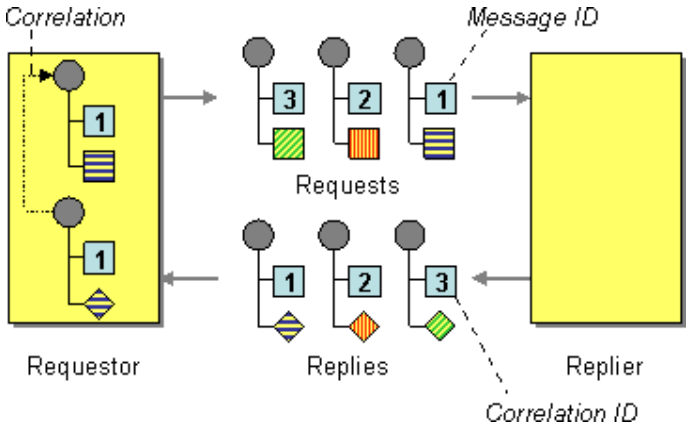


Figure 1: Correlation Identifier EIP

How WSO2 ESB implements the EIP

WSO2 message flow automatically manages requests sent by a client. It injects a unique identification number into the message context of each request. When the response arrives, the message will be uniquely identified using this number, allowing the requester to distinguish between requests.

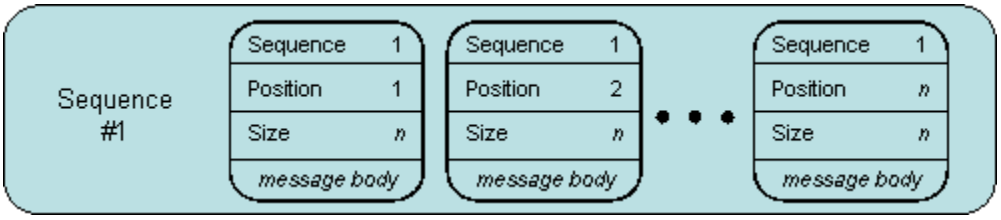
Message Sequence

This section explains how the Message Sequence EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Message Sequence
- How WSO2 ESB implements the EIP

Introduction to Message Sequence

When transmitting a large set of data, you might want to break it into smaller chunks in order to maintain system performance. The Message Sequence EIP facilitates this by sending data as a sequence of smaller messages and marking each message with sequence identification fields. For more information, refer to http://www.eaipatterns.com/MessageSequence.html.



How WSO2 ESB implements the EIP

When the sent request is large, WSO2 ESB by default breaks it into smaller chunks. In order to maintain consistency of the message, each chunk is then mapped with a sequence identity number so that its sequential order is not lost.

Message Expiration

This section explains, through an example scenario, how the Message Expiration EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Message Expiration

Introduction to Message Expiration

The Message Expiration EIP allows a sender to indicate when a message should be considered stale and shouldn't be processed. You can set the Message Expiration to specify a time limit in which a message is viable. For more information, refer to http://www.eaipatterns.com/MessageExpiration.html.

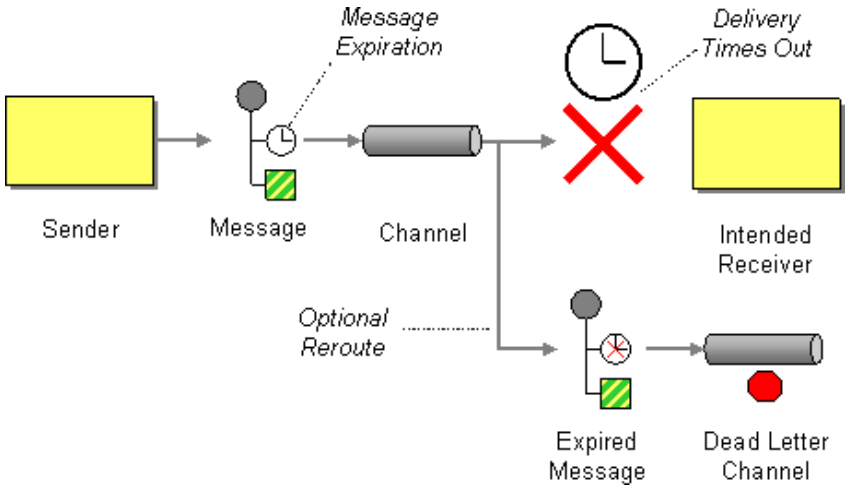


Figure 1: Message Expiration EIP

Example scenario

This example scenario simulates message expiration using the endpoint timeout property in WSO2 ESB. Message expiration sets a time limit for a sent message to be visible. If the limit is exceeded, the message will be discarded without being sent to the receiver.

While setting a delivery timeout for a message in the Message Expiration EIP can be an inherent component of a sender process, WSO2 ESB can implement this EIP through the use of an endpoint timeout. Alternatively, you can use a MessageStore with a **Time To Live** duration set by WSO2 ESB, which expires messages if this duration is reached before passing the message to the intended receivers.

The diagram below depicts how to simulate the example scenario in WSO2 ESB.

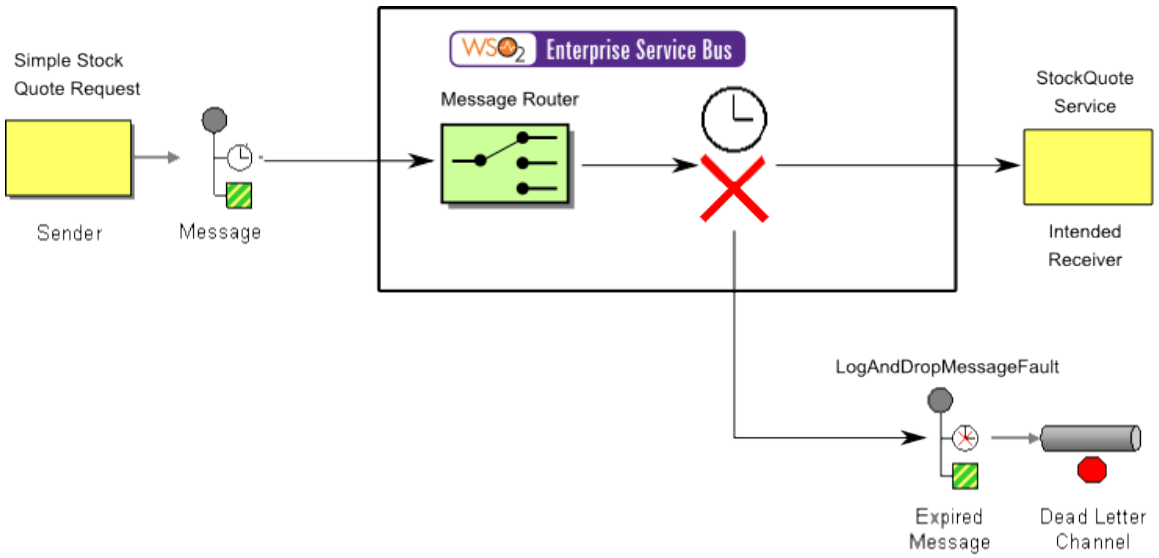


Figure 2: Example Scenario of the Message Expiration EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Message Expiration EIP by comparing their core components.

Message Expiration EIP (Figure 1)	Message Expiration Example Scenario (Figure 2)
Sender	Stock Quote Client
Channel	Proxy Service
Dead Letter Channel	Fault Sequence
Intended Receiver	Stock Quote Service Instance

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB Documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.


```
<definitions xmlns="http://ws.apache.org/ns/synapse">
<proxy xmlns="http://ws.apache.org/ns/synapse" name="MessageExpirationProxy"
transports="https,http" statistics="disable" trace="disable" startOnLoad="true">
   <target>
      <inSequence onError="LogAndDropMessageFault">
         <log level="full"/>
         <send/>
      </inSequence>
      <outSequence onError="fault">
         <log level="full"/>
         <send/>
      </outSequence>
      <endpoint name="TimeoutEndpoint">
         <address uri="http://localhost:9000/services/SimpleStockQuoteService">
            <timeout>
               <duration>30000</duration>
               <responseAction>fault</responseAction>
            </timeout>
         </address>
      </endpoint>
   </target>
   <description></description>
</proxy>
<sequence name="LogAndDropMessageFault">
   <log level="full">
      <property name="MESSAGE" value="Executing default &#34;fault&#34; sequence"/>
      <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
      <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
   </log>
   <drop/>
</sequence>
</definitions>
```

Simulating the sample scenario

1. Send a request using the Stock Quote client to WSO2 ESB in the following manner. For information on the Stock Quote client, refer to the Sample Clients section in the WSO2 ESB documentation.

   ```
   ant stockquote -Dtrpurl=http://localhost:8280/services/MessageExpirationProxy
   -Dsymbol=Foo
   ```

2. Notice the expected response for the request. Next, drop the Axis2 server instance and restart the ESB. The endpoint will time out and be suspended after the timeout period, causing a fault condition.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **Endpoint** [line 13 in ESB config] - Defines the back-end service the messages are passed to.
- **timeout** [line 15 in ESB config] - Defines the timeout duration in milliseconds, and also the action to take (fault or discard) in the event of a response timeout. In this example scenario, when a timeout occurs, the `LogAndDropMessageFault` sequence activates where the message is dropped using the Drop Mediator. An alternative to dropping the message is to pass the message to a Dead Letter Channel.

Format Indicator

The Format Indicator EIP allows communication channels to know the format of data transmitted to the ESB. Once the format is identified, the receiver will know how to process the message based on its format. For more information, refer to http://www.eaipatterns.com/FormatIndicator.html.

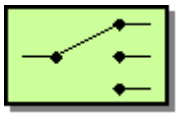
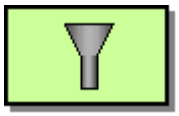
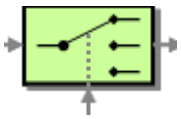
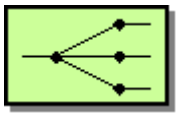
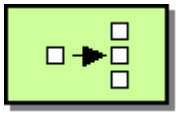
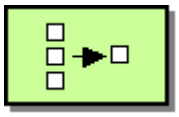
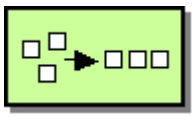
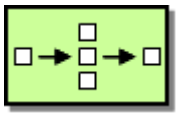
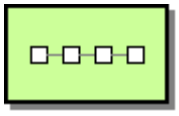
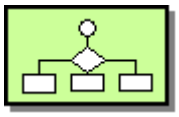
How WSO2 ESB implements the EIP

The Architecture section in the WSO2 ESB documentation describes how the ESB processes a message it receives. When a message arrives, before it is sent through mediation, it will be transformed to a SOAP envelope using message builders. This way, the message will be in a standard format for the message mediation engine to do the processing. Once the response is ready to be sent back to the receiver, it will be changed back to its original format through message formatters.

Message Routing

A message router is a basic architectural pattern of a messaging system used fundamentally for connecting different message channels. A router consumes a message from one message channel and republishes it to a different channel based on specified conditions.

This chapter introduces various types of routers and how each of them can be simulated using WSO2 ESB.

	Content-Based Router	How to handle a situation when the implementation of a single logical function (such as an inventory check) is spread across multiple physical systems.
	Message Filter	How a component avoids receiving uninteresting messages.
	Dynamic Router	How to avoid the dependency of a router in all possible destinations, while maintaining its efficiency.
	Recipient List	How to route a message to a list of dynamically specified recipients.
	Splitter	How to process a message if it contains multiple elements, each of which may have to be processed in a different way.
	Aggregator	How to combine the results of individual but related messages so that they can be processed as a whole.
	Resequencer	How to get a stream of related but out-of-sequence messages back into the correct order.
	Composed Msg. Processor	How to maintain the overall flow when processing a message consisting of multiple elements, each of which may require different processing.
	Scatter-Gather	How to maintain the overall flow when a message needs to be sent to multiple recipients, each of which may send a reply.
	Routing Slip	How to route a message consecutively through a series of steps when the sequence of the steps is not known at design time and may vary for each message.
	Process Manager	How to route a message through multiple processing steps, when the required steps may not be known at design time and may not be sequential.

	Message Broker	How to decouple the destination of a message from the sender and maintain central control over the flow of messages.

Content-Based Router

This section explains, through an example scenario, how the Content-Based Router EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Content-Based Router
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Content-Based Router

The Content-Based Router (CBR) reads the content of a message and routes it to a specific recipient based on its content. This approach is useful when an implementation of a specific logical function is distributed across multiple physical systems.

The following diagram depicts the Content-Based Router's behavior where the router performs a logical function (e.g., inventory check). It receives a request message (new order), reads it, and routes the request to one of the two recipients according to the message's content. For more information on the Content-Based Router, refer to http://www.eaipatterns.com/ContentBasedRouter.html.

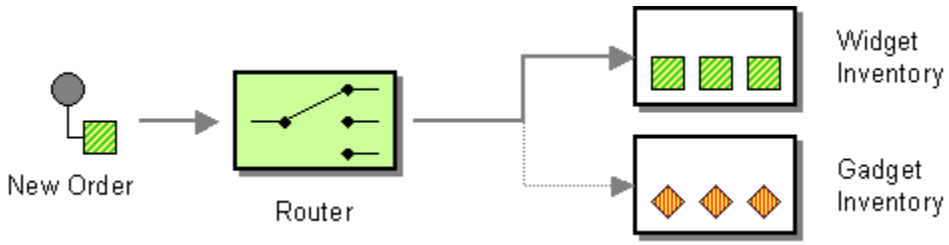


Figure 1: Content-Based Router EIP

Example scenario

The example scenario depicts an inventory for stocks and illustrates how the Content-Based Router routes a message based on the message content. When the router receives a stock request, it reads the content of the request. If the request is made for Foo, the request is routed to the Foo stock inventory service. If the request is for Bar, it is routed to the Bar stock inventory service.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

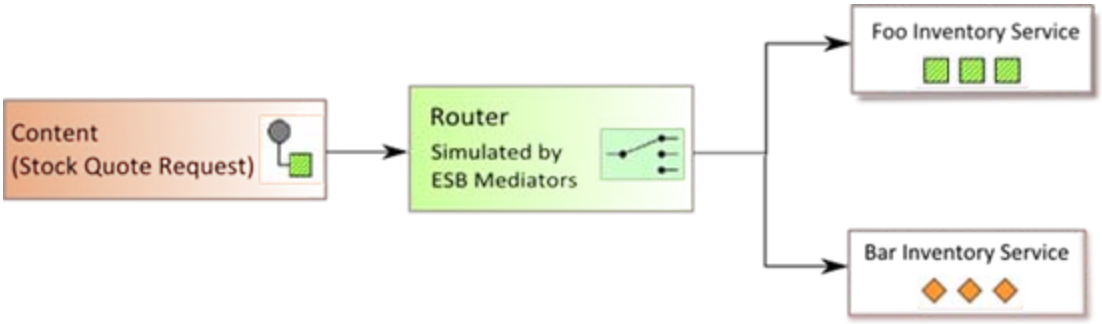


Figure 2: Content-Based Router Example Scenario

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Content-Based Router EIP by comparing their core components.

Content-Based Router EIP (Figure 1)	Content-Based Router Example Scenario (Figure 2)
New Order	Stock Quote Request
Router	Message routing is simulated by the Switch and Send mediators of WSO2 ESB. The Switch Mediator acts as the router and observes the content of the message, while the Send Mediator is used to send the message to a selected recipient. Each case defined should decide on routing the message to the appropriate service.
Widget and Gadget Inventory	Foo Inventory Service and Bar Inventory service act as two separate services in the example scenario.

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances on ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.


```
<!-- The example use of content based routing -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <!-- The service which the sender will be invoking -->
    <proxy name="ContentBasedRoutingProxy">
        <target>
           <!-- When a request arrives the following sequence will be followed -->

           <inSequence>
    <!-- The content of the incoming message will be isolated -->
             <switch source="//m0:getQuote/m0:request/m0:symbol"
xmlns:m0="http://services.samples">
                 <!-- The isolated content will be filtered -->
                    <case regex="Foo">
                  <!-- Will Route the content to the appropriate destination -->
      <send>
                        <endpoint>
                         <address
uri="http://localhost:9001/services/SimpleStockQuoteService?wsdl"/>
                        </endpoint>
                    </send>
              </case>
              <case regex="Bar">
                  <send>
                       <endpoint>
                       <address
uri="http://localhost:9002/services/SimpleStockQuoteService?wsdl"/>
                       </endpoint>
                    </send>
              </case>
              <default>
                  <!-- it is possible to assign the result of an XPath expression as
well -->
                 <property name="symbol" expression="fn:concat('Normal Stock - ',
//m0:getQuote/m0:request/m0:symbol)" xmlns:m0="http://services.samples"/>
              </default>
            </switch>
            </inSequence>
            <outSequence>
                <send/>
            </outSequence>
        </target>
    </proxy>
</definitions>
```

Simulating the sample scenario

1. Send a request using the `Stock Quote` client to WSO2 ESB in the following manner. For information about the `Stock Quote` client, refer to the section Sample Clients in the WSO2 ESB documentation.

   ```
   ant stockquote -Dtrpurl=http://localhost:8280/services/ContentBasedRoutingProxy
   -Dsymbol=Foo
   ```
2. After executing the above command through the client, observe that the request is transferred to the Foo inventory service. If the `-Dsymbol` parameter is changed to `Bar,` the request will be transferred to the Bar inventory service.

The structure of the request is as follows:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <ser:symbol>Foo</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **Proxy Service** [line 4 of ESB config] - The proxy service takes requests and forwards them to appropriate the back-end service, abstracting the routing logic from the client. Regardless of the request, the client sends it to the exposed service and not to the back-end services.
- **inSequence** [line 7 of ESB config] - When the service is invoked through the client, the message will be received by the inSequence and sent as per the routing logic.
- **switch** [line 9 of ESB config] - Observes the message and filters out the message content as per the XPath expression.
- **case** [line 11 and 19 of ESB config] - The filtered content will match the specified regular expression.
- **send** [line 13, 20, and 34 of ESB config] - When a matching case is found, the send mediator will route the message to the endpoint specified in the address URI.
- **default** [line 26 of ESB config] - If a matching condition is not found, the message will be diverted to the default case.
- **outSequence** [line 33 of ESB config] - The response from an endpoint is received through the outSequence. The message will be transferred back to the sender.

Message Filter

This section explains, through an example scenario, how the Message Filter EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Message Filter

The Message Filter EIP checks an incoming message against a certain criteria that the message should adhere to. If the criteria is not met, the filter will discard the message. Otherwise, it will route the message to the output channel. For more information, refer to http://www.eaipatterns.com/Filter.html.

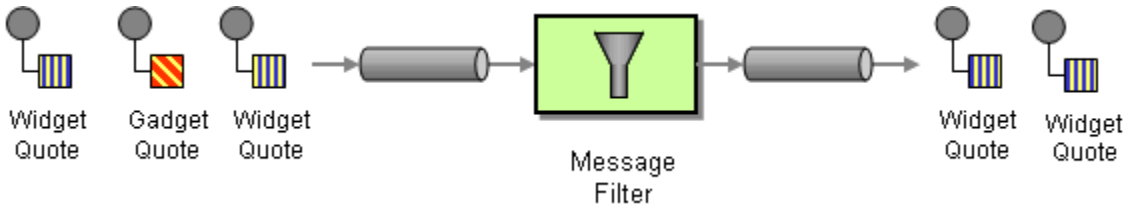


Figure 1: Message Filter EIP

Example scenario

The example scenario depicts an inventory for stocks where an incoming request will be filtered based on its content. If the content meets the criteria, the message filter will allow the request to proceed. Otherwise, the message will be dropped.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

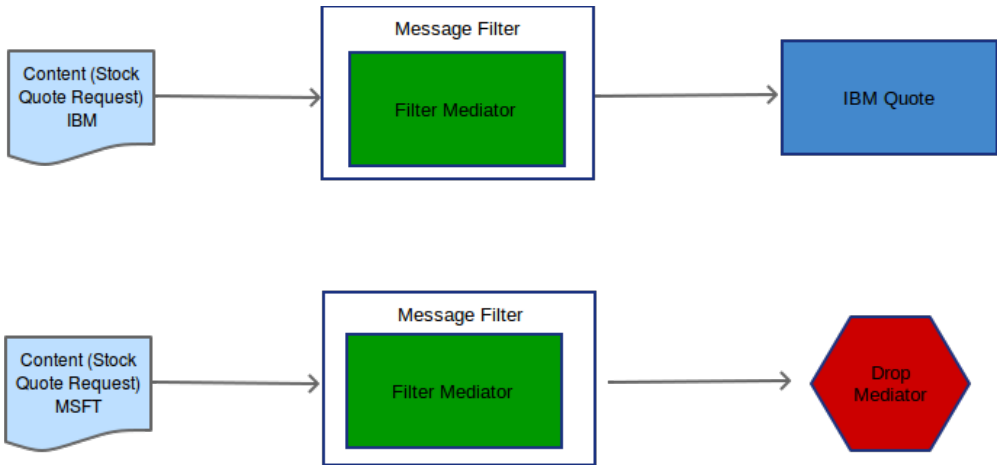


Figure 2: Message Filter Example Scenario

Before digging into implementation details, let's take a look at the co-relation of the example scenario and the Message Filter EIP by comparing their core components.

Message Filter EIP (Figure 1)	Message Filter Example Scenario (Figure 2)
Quote	Stock Quote Request
Message Filter	Filter Mediator is used to filter the content of the incoming message

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances in ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the

management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- The example use of Mesage Filtering -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <proxy name="MessageFilterProxy">
        <target>
            <inSequence>
        <filter xmlns:m0="http://services.samples"
source="//m0:getQuote/m0:request/m0:symbol" regex="Foo">
              <then>
              <send>
                <endpoint>
                    <address
uri="http://localhost:9000/services/SimpleStockQuoteService?wsdl"/>
                </endpoint>
              </send>
              </then>
              <else>
              <drop/>
              </else>
        </filter>
        </inSequence>
            <outSequence>
                <send/>
            </outSequence>
        </target>
    </proxy>
</definitions>
```

Simulating the sample scenario

1. Send a request using the Stock Quote client to WSO2 ESB in the following manner. For information about the Stock Quote client, refer to the section Sample Clients in the WSO2 ESB Documentation.

   ```
   ant stockquote -Dtrpurl=http://localhost:8280/services/ContentBasedRoutingProxy
   -Dsymbol=Foo
   ```
2. After executing the above command through the client, observe that the request is transferred to the Foo inventory service and a response is received. If the `-Dsymbol` parameter was changed to another value, there will be no response.

   The structure of the request is as follows:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <ser:symbol>FOO</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **Proxy Service** - The proxy service takes requests and forwards them to appropriate the back-end service, abstracting the routing logic from the client. Regardless of the request, the client sends it to the exposed service and not to the back-end services.
- **inSequence** - When the service is invoked through the client, the message will be processed by the inSequence and sent as per the routing logic.
- **filter** - Filters incoming messages, dropping any that do not meet the criteria.
- **send** - When a matching case is found, the send mediator will route the message to the endpoint specified in address URI.
- **outSequence** - The response from the endpoint is processed by the outSequence. The message will be transferred back to the sender.

Dynamic Router

This section explains, through an example scenario, how the Dynamic Router EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Dynamic Router

The Dynamic Router EIP avoids dependence on all possible destinations while maintaining efficiency. It is a router that can self-configure based on special configuration messages from participating destinations. Dynamic Router is available for configuration through a control channel by the receiving parties that can use this control channel.

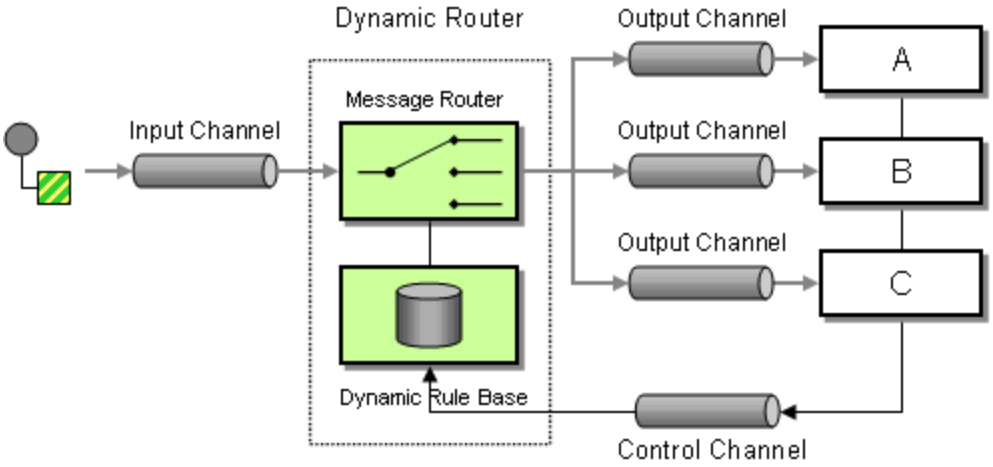


Figure 1: Dynamic Router EIP

Example scenario

This example scenario demonstrates a router that takes an incoming request and decides which back-end service to transmit the message to. To make that decision, it uses a property in the message itself, very much like the Content-Based Router. However, it can also cross-check a registry entry to see if a specific endpoint accepts messages with that property. This approach allows you to reconfigure the router when registry entries change.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

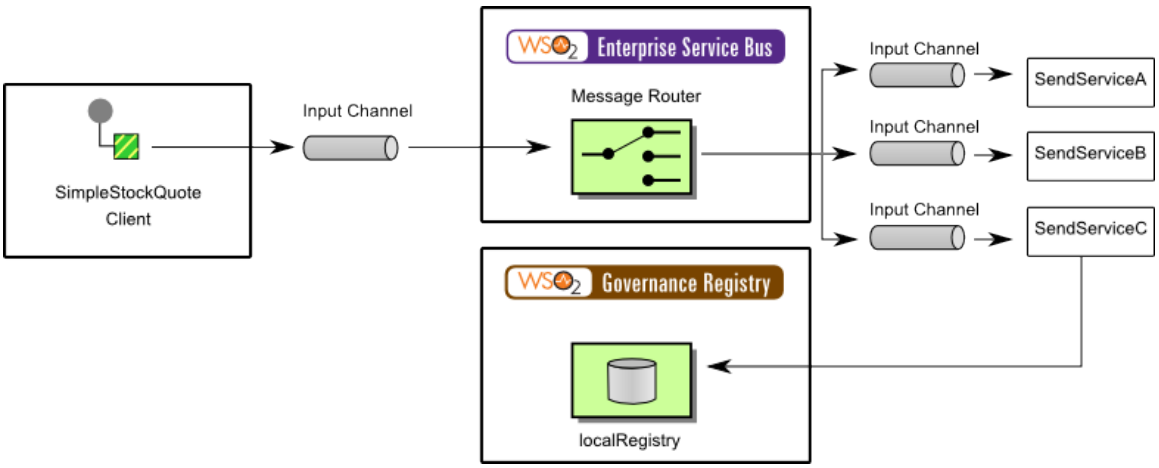


Figure 2: Dynamic Router Example Scenario

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Dynamic Router EIP by comparing their core components.

Dynamic Router EIP (Figure 1)	Dynamic Router Example Scenario (Figure 2)
Input Channel	Main sequence
Dynamic Rule Base	Local Registry
Dynamic Router	Switch Mediator
A, B, C	Simple Stock Quote Services

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start three sample Axis2 server instances on ports 9000, 9001, and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.
3. Create the following files inside the `<ESB_HOME>/repository` directory:
   - `configA.xml` - with content `<value>Foo</value>`
   - `configB.xml` - with content `<value>BAR</value>`
   - `configC.xml` - with content `<value>WSO2</value>`

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <!-- Defines the registry accessible by the configuration - a localRegistry -->
   <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
      <parameter name="localRegistry">/</parameter>
      <parameter name="cachableDuration">15000</parameter>
   </registry>
   <sequence name="SendServiceA">
      <property name="ServiceAProp"
expression="get-property('registry','file:./repository/configA.xml')"/>
      <switch xmlns:m0="http://services.samples"
source="//m0:getQuote/m0:request/m0:symbol">
         <case regex="get-property('ServiceAProp')">
            <send>
               <endpoint>
                  <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
               </endpoint>
            </send>
         </case>
      </switch>
   </sequence>

   <sequence name="SendServiceB">
      <property name="ServiceBProp"
expression="get-property('registry','file:./repository/configB.xml')"/>
      <switch xmlns:m0="http://services.samples"
source="//m0:getQuote/m0:request/m0:symbol">
         <case regex="get-property('ServiceBProp')">
            <send>
               <endpoint>
                  <address
uri="http://localhost:9001/services/SimpleStockQuoteService"/>
               </endpoint>
            </send>
         </case>
      </switch>
   </sequence>
   <sequence name="SendServiceC">
      <property name="ServiceAProp"
```

```
expression="get-property('registry','file:./repository/configC.xml')"/>
      <switch xmlns:m0="http://services.samples"
source="//m0:getQuote/m0:request/m0:symbol">
         <case regex="get-property('ServiceCProp')">
            <send>
               <endpoint>
                  <address
uri="http://localhost:9002/services/SimpleStockQuoteService"/>
               </endpoint>
            </send>
         </case>
      </switch>
   </sequence>
   <sequence name="main">
      <in>
         <switch source="get-property('To')">
            <case regex="http://localhost:9000.*">
               <sequence key="SendServiceA"/>
            </case>
            <case regex="http://localhost:9001.*">
               <sequence key="SendServiceB"/>
            </case>
            <case regex="http://localhost:9002.*">
               <sequence key="SendServiceC"/>
            </case>
         </switch>
      </in>
      <out>
         <send/>
```

```
        </out>
    </sequence>
</definitions>
```

Simulating the sample scenario

Send requests using the `Stock Quote` client to WSO2 ESB in the following manner. For information about the `Stock Quote` client, refer to the section Sample Clients in the WSO2 ESB documentation.

```
ant stockquote -Daddrurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=Foo
ant stockquote -Daddrurl=http://localhost:9001/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=BAR
ant stockquote -Daddrurl=http://localhost:9002/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=WSO2

ant stockquote -Daddrurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=BAR
ant stockquote -Daddrurl=http://localhost:9001/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=WSO2
ant stockquote -Daddrurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280/ -Dsymbol=Foo
```

Notice that only the first three commands are sent to the back-end services, as the symbols passed within them are the symbols associated with that particular endpoint service.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **sequence** [line 44 in ESB config] - This is the default main sequence invoked when a message is received by the ESB.
- **switch** [line 46 in ESB config] - The switch inside the main sequence's `in` mediator. It checks the To header inside the SOAP message to see which endpoint the message is intended for. Based on this, one of the three sequences is invoked: SendServiceA, SendServiceB or SendServiceC.
- **sequence** [line 7 in ESB config] - This is the sequence with key `SendServiceA`. The other sequences starting in lines 20 and 32 in the ESB config follow the same pattern as this sequence.
- **property** [line 8 in ESB config] - This is the `ServiceAProp` property mediator, set inside `SendServiceA`. It pulls the value out of the `configA.xml` file using the `get-property` XPath function, which looks inside the registry for the specified file.
- **switch** [line 9 in ESB config] - A switch mediator is used to compare the value of the symbol element inside the SOAP request body to see if it matches the value contained within the `configA.xml` file. If the two values are the same, the message is forwarded to the specified endpoint using the send mediator.

Recipient List

This section explains, through an example scenario, how the Recipient List EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Recipient List
- Example scenario

- Environment setup
- ESB configuration
- Simulating the sample scenario
- How the implementation works

Introduction to Recipient List

The Recipient List EIP routes a message to a list of dynamically specified recipients. It processes an incoming message and identifies its list of recipients. Once the list is identified, the message will be sent to all recipient channels. For more information, refer to http://www.eaipatterns.com/RecipientList.html.

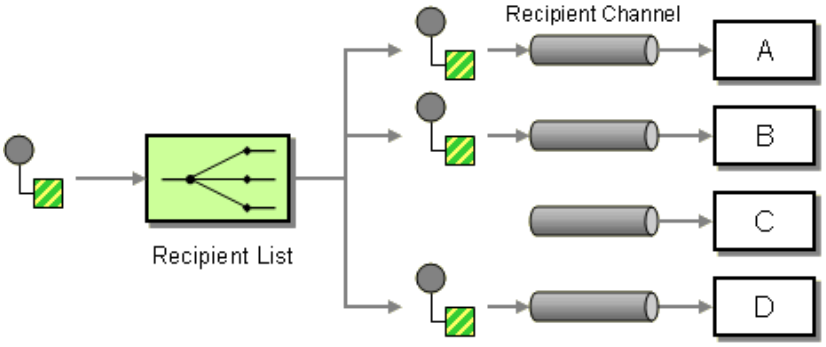


Figure 1: Recipient List EIP

Example scenario

This example scenario is a stock quote service sending a stock quote request to recipients that are instances of a sample Axis2 server. The Switch mediator identifies the content of the client request and distributes the content among the Recipient List endpoints.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

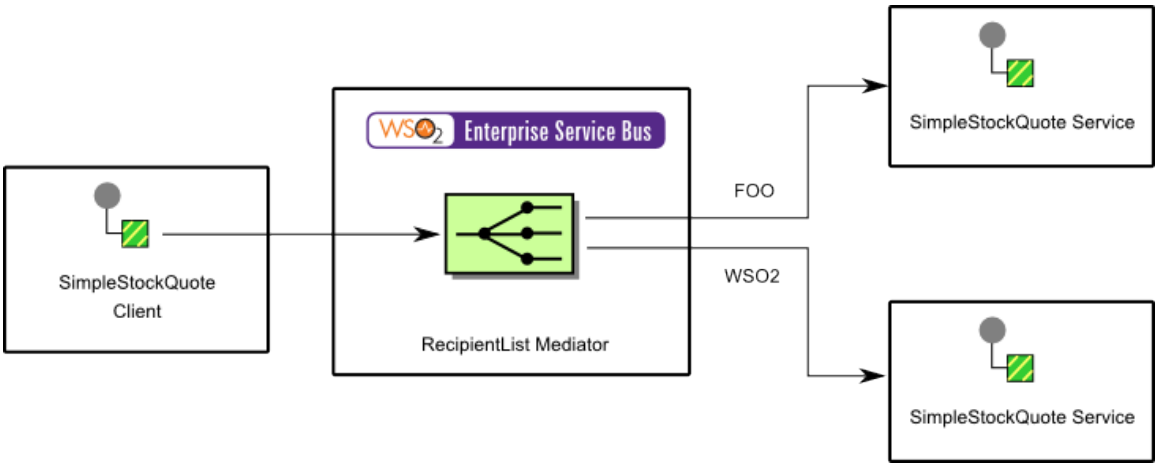


Figure 2: Recipient List Example Scenario

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Recipient List EIP by comparing their core components.

Recipient List EIP (Figure 1)	Recipient List Example Scenario (Figure 2)
Sender	StockQuoteClient

Recipient List	RecipientList mediator
Receivers (A, B, C, D)	SimpleStockQuote Service Instances (Foo, WSO2)

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Deploy the `SimpleStockQuoteService` and start three instances of Axis2 Server in ports 9000, 9001, 9002, and 9003. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- Would Route the Message Based on the List of Recipients-->
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
        <parameter name="root">file:repository/samples/resources/</parameter>
        <parameter name="cachableDuration">15000</parameter>
    </registry>
    <proxy name="RecipientListProxy">
        <target>
            <inSequence>
            <switch source="//m0:getQuote/m0:request/m0:symbol"
xmlns:m0="http://services.samples">
            <!-- First the recipient list will be identified -->
                <case regex="Foo">
               <send>
                  <!--Dynamic Recipient List-->
                        <endpoint>
                       <recipientlist>
                         <endpoint xmlns="http://ws.apache.org/ns/synapse">
                                  <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
                         </endpoint>
                         <endpoint xmlns="http://ws.apache.org/ns/synapse">
                                  <address
uri="http://localhost:9001/services/SimpleStockQuoteService"/>
                         </endpoint>
                        </recipientlist>
                    </endpoint>
                </send>
                  <drop/>
            </case>
            <case regex="WSO2">
               <send>
                  <!--Dynamic Recipient List-->
                        <endpoint>
                     <recipientlist>
                         <endpoint xmlns="http://ws.apache.org/ns/synapse">
                                  <address
uri="http://localhost:9002/services/SimpleStockQuoteService"/>
                         </endpoint>
                         <endpoint xmlns="http://ws.apache.org/ns/synapse">
                                  <address
uri="http://localhost:9003/services/SimpleStockQuoteService"/>
                         </endpoint>
                        </recipientlist>
                    </endpoint>
                </send>
                  <drop/>
            </case>
            <default>
              <!-- Message Should Be Discarded -->
            </default>
            </switch>
            </inSequence>
            <outSequence>
                <send/>
            </outSequence>
        </target>
     </proxy>
</definitions>
```

Simulating the sample scenario

Send a request using the Stock Quote client to WSO2 ESB in the following manner. For information about the Stock Quote client, refer to the Sample Clients section in the WSO2 ESB documentation.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/RecipientListProxy
-Dsymbol=WSO2
```

Note that the ESB sends the request to servers running on ports 9002 and 9003. If you change the symbol to Foo, it will send the requests to servers running on port 9000 and 9001.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **Switch** [line 10 in ESB config] - The Switch mediator performs a switch/case based on the symbol found inside the original request. In this example scenario, one of two send mediators are used, based on the value of the `symbol` element in the request.
- **recipientList** [line 16 in ESB config] - the `recipientList` mediator lists several endpoints inside the `send` t ags. ESB will forward the request to all endpoints in this list.

Splitter

This section explains, through an example scenario, how the Splitter EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Splitter

The Splitter EIP processes messages containing multiple elements that might have to be processed in different ways. The Splitter breaks out the composite message into a series of individual messages, each containing data related to one item. For more information, refer to http://www.eaipatterns.com/Sequencer.html.

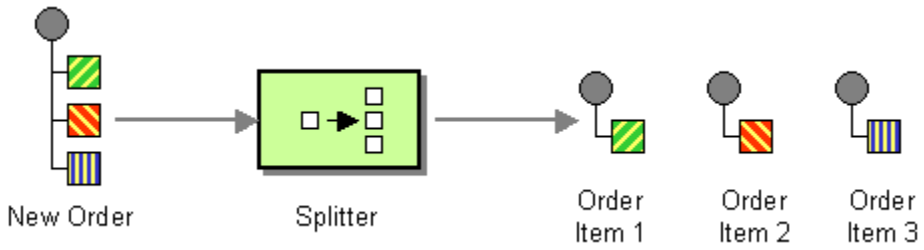


Figure 1: Splitter EIP

Example scenario

This example demonstrates WSO2 ESB implementing the Splitter EIP by processing a list of repeating elements,

each of which will be processed individually. The client sends multiple requests in a single message. Using the Iterate mediator of the ESB, each request will be processed individually and transmitted to a sample Axis2 server.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

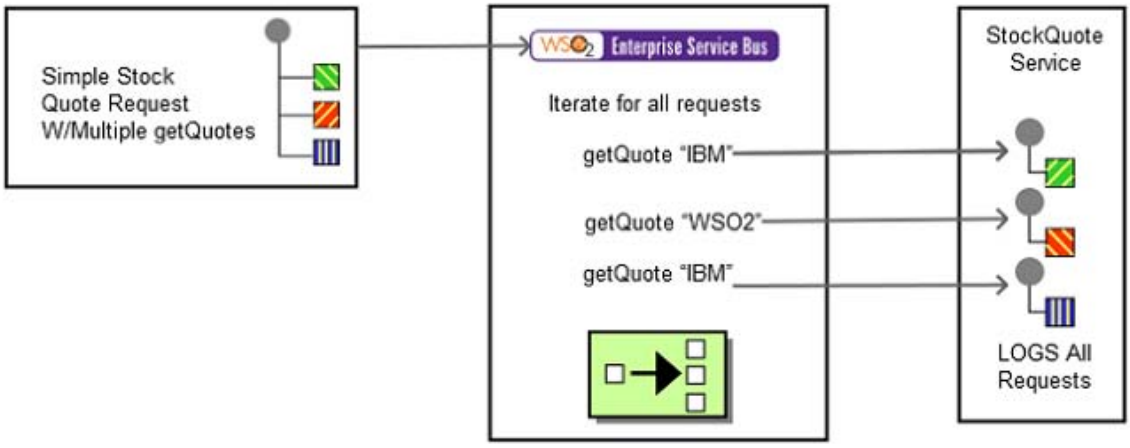


Figure 2: Splitter Example Scenario

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Splitter EIP by comparing their core components.

Splitter EIP (Figure 1)	Splitter Example Scenario (Figure 2)
New Order Request	Stock Quote Request
Splitter	Iterate Mediator
Order Item Request	Endpoint (stock quote service instances)

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start a Sample Axis2 server instance. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="SplitMessageProxy" transports="http https" startOnLoad="true">
      <target>
         <inSequence>
            <log level="full"/>
            <iterate xmlns:m0="http://services.samples"
                     preservePayload="true"
                     attachPath="//m0:getQuote"
                     expression="//m0:getQuote/m0:request">
               <target>
                  <sequence>
                     <send>
                        <endpoint>
                           <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
                        </endpoint>
                     </send>
                  </sequence>
               </target>
            </iterate>
         </inSequence>
         <outSequence>
            <drop/>
         </outSequence>
      </target>
      <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
   </proxy>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in/>
      <out/>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send the following request to the ESB server using a SOAP client like SoapUI.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <ser:request>
            <xsd:symbol>IBM</xsd:symbol>
         </ser:request>
          <ser:request>
            <xsd:symbol>WSO2</xsd:symbol>
         </ser:request>
         <ser:request>
            <xsd:symbol>IBM</xsd:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

Note that the three requests will appear as sent in the Axis2 server log.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **iterate** [line 6 in ESB config] - The Iterate mediator takes each child element of the element specified in its XPath expression and applies the sequence flow inside the iterator mediator. In this example, it takes each `getQuote` request specified in the incoming request to the ESB and forwards this request to the target endpoint.

Aggregator

This section explains, through an example scenario, how the Aggregator EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Aggregator
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Aggregator

The Aggregator EIP combines the results of individual, related messages so that they can be processed as a whole. It works as a stateful filter, collecting and storing individual messages until a complete set of related messages has been received. It then publishes a single message distilled from the individual messages. For more information, refer to http://www.eaipatterns.com/Aggregator.html.

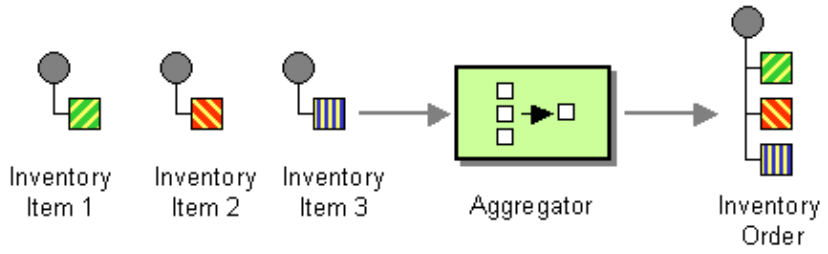


Figure 1: Aggregator EIP

Example scenario

This example scenario demonstrates how WSO2 ESB can be used to send multiple requests and merge the responses. Assume the sender wants to get responses from multiple servers, and it sends a message to the ESB. The ESB will duplicate the request to three different instances of a sample Axis2 server using the Clone Mediator. The ESB will merge the responses received from each of the server instances using the Aggregate Mediator and send it back to the client.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

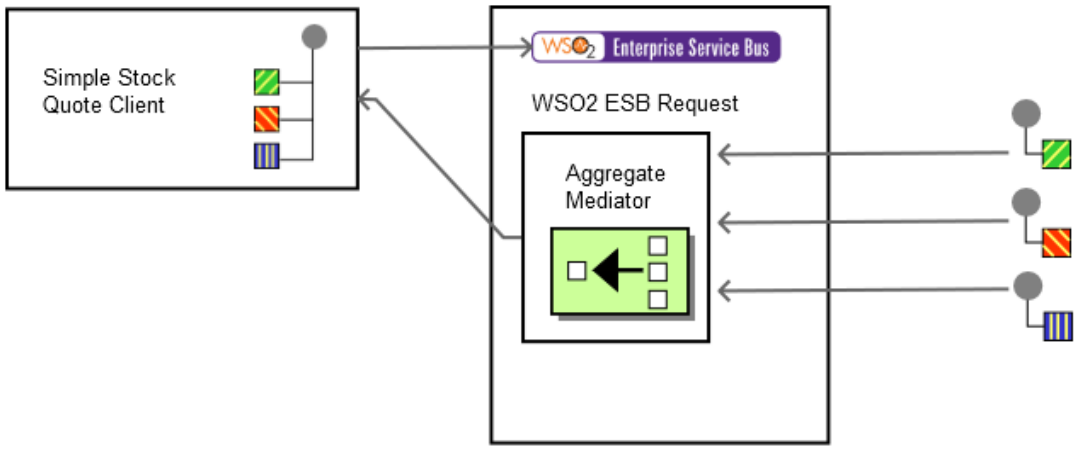


Figure 2: Aggregator Example Scenario

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Aggregator EIP by comparing their core components.

Aggregator EIP (Figure 1)	Aggregator Example Scenario (Figure 2)
Inventory Item	Simple Stock Quote Service Response
Aggregator	Aggregate Mediator
Inventory Order	WSO2 ESB Response

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start three Sample Axis2 server instances on ports 9000, 9001, and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="AggregateMessageProxy"
          transports="http https"
          startOnLoad="true">
      <target>
         <inSequence>
            <log level="full"/>
            <clone>
               <target>
                  <endpoint name="ReceiverA">
                     <address
uri="http://localhost:9000/services/SimpleStockQuoteService/"/>
                  </endpoint>
               </target>
               <target>
                  <endpoint name="ReceiverB">
                     <address
uri="http://localhost:9001/services/SimpleStockQuoteService/"/>
                  </endpoint>
               </target>
            </clone>
         </inSequence>
         <outSequence>
            <aggregate>
               <completeCondition>
                  <messageCount/>
               </completeCondition>
               <onComplete xmlns:m0="http://services.samples"
expression="//m0:getQuoteResponse">
                  <send/>
               </onComplete>
            </aggregate>
         </outSequence>
      </target>
   </proxy>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
      <drop/>
   </sequence>
</definitions>
```

Simulating the sample scenario

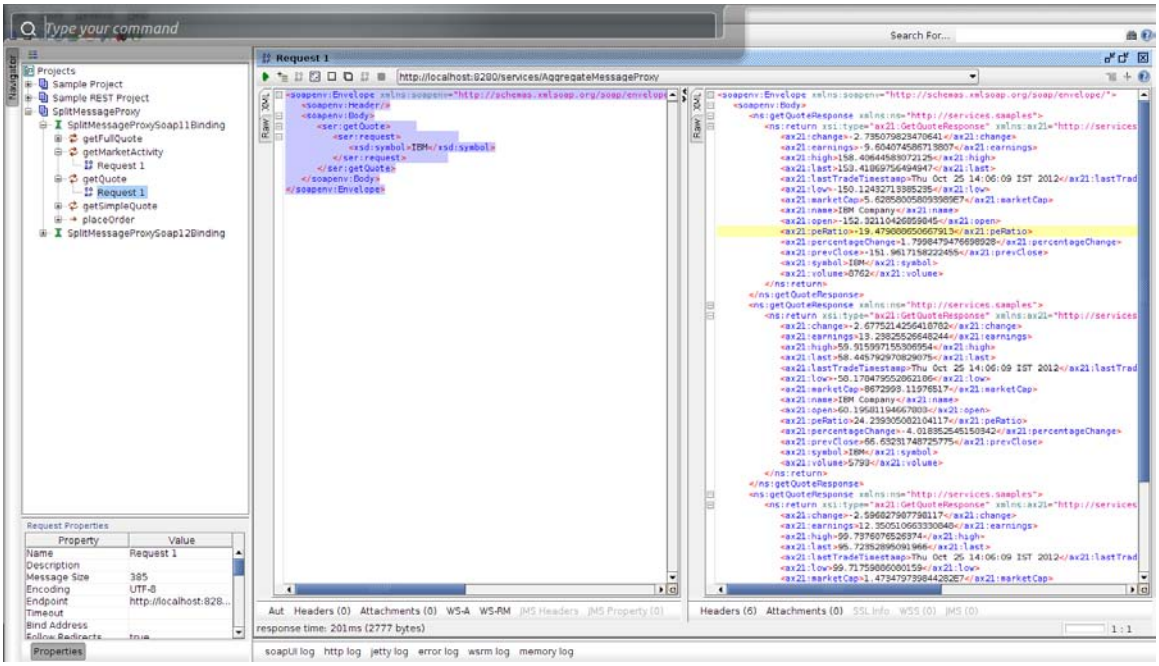
1. Send the following request to the ESB using a SOAP client like SoapUI.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">

   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <ser:request>
            <xsd:symbol>Foo</xsd:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>

</soapenv:Envelope>
```

2. The user will be able to see the merged response for all three quotes made in the request as shown in the following example screen.



How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **clone** [line 9 in ESB config] - The Clone mediator is similar to the Splitter EIP. It clones the incoming request and passes the requests in parallel to several endpoints.
- **aggregate** [line 23 in ESB config] - The aggregate mediator aggregates response messages for requests made by the Iterate or Clone mediators. The completion condition specifies a minimum or maximum number of messages to be collected. When all messages have been aggregated, the sequence inside `onComplete` will be run.

Resequencer

> The Resequncer EIP is available from WSO2 ESB version 4.6.1 onwards.

This section explains, through an example scenario, how the Resequencer EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Resequencer
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Resequencer

The Resequencer EIP puts a stream of related but out-of-sequence messages back into the correct order. It collects and re-orders messages so that they can be published to the output channel in a specified order. For more information, refer to http://www.eaipatterns.com/Resequencer.html.

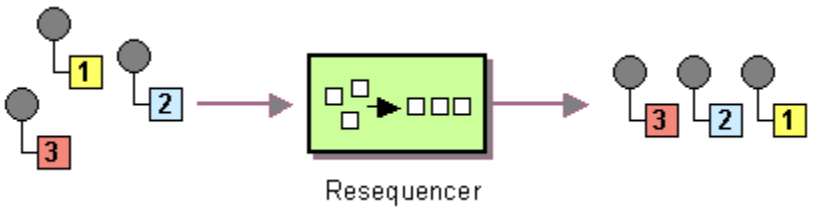


Figure 1: Resequencer EIP

Example scenario

Messages in the ESB usually take different mediation paths due to constructs like routers and filters. The time taken to process each message varies depending on the characteristics of each router. Some messages might arrive earlier than others, which can be disadvantageous in situations where the order of message delivery is important. To overcome this issue, WSO2 ESB has the resequencing EAI pattern introduced with a stateful processor called Resequencing Processor.

This example scenario demonstrates WSO2 ESB collecting incoming messages using a message store and resequencing or reordering them based on a definitive sequence number derived from some part of the message.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

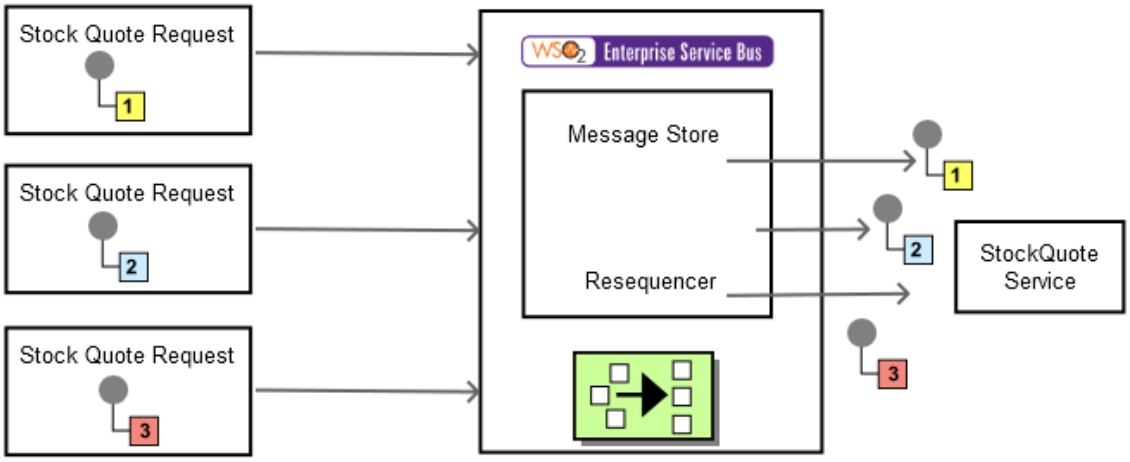


Figure 2: Resequencer Example Scenario

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Resequencer EIP by comparing their core components.

Resequencer EIP (Figure 1)	Resequencer Example Scenario (Figure 2)
Incoming Messages	Simple Stock Quote Requests
Resequence	Message Store with Resequencing Processor
Outgoing Messages	Simple Stock Quote Requests

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two sample Axis2 server instances on ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- A configuration to exhibit resequencing processor in ESB -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
 <proxy name="resequencingProxy" transports="http,https" startOnLoad="true">
 <target>
  <inSequence>
   <log level="full"/>
   <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
   <property name="OUT_ONLY" value="true"/>
   <store messageStore="MyStore"/>
  </inSequence>
  <outSequence>
   <send />
  </outSequence>
 </target>
 </proxy>
 <sequence name="next_seq">
 <send>
  <endpoint>
   <address uri="http://localhost:9000/services/SimpleStockQuoteService" />
  </endpoint>
 </send>
 </sequence>
 <messageStore name="MyStore"/>
 <messageProcessor
   class="org.apache.synapse.message.processors.resequence.ResequencingProcessor"
   name="ResequencingProcessor" messageStore="MyStore">
  <parameter name="interval">2000</parameter>
  <parameter name="seqNumXpath" xmlns:m0="http://services.samples"
expression="substring-after(//m0:placeOrder/m0:order/m0:symbol,'-')"/>
  <parameter name="nextEsbSequence">next_seq</parameter>
  <parameter name="deleteDuplicateMessages">true</parameter>
 </messageProcessor>
</definitions>
```

Simulating the sample scenario

1. Send the following requests using the Stock Quote client to WSO2 ESB. For information about the Stock Quote client, refer to the section Sample Clients in the WSO2 ESB documentation.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/resequencingProxy
-Dmode=placeorder -Dsymbol=WSO2-4
ant stockquote -Dtrpurl=http://localhost:8280/services/resequencingProxy
-Dmode=placeorder -Dsymbol=WSO2-1
ant stockquote -Dtrpurl=http://localhost:8280/services/resequencingProxy
-Dmode=placeorder -Dsymbol=WSO2-5
ant stockquote -Dtrpurl=http://localhost:8280/services/resequencingProxy
-Dmode=placeorder -Dsymbol=WSO2-2
ant stockquote -Dtrpurl=http://localhost:8280/services/resequencingProxy
-Dmode=placeorder -Dsymbol=WSO2-3
```

2. Note that the output from the axis2 server is in ascending order rather than the order in which you sent the requests.

```
Sat Sep 22 08:55:11 IST 2012 samples.services.SimpleStockQuoteService  ::
Accepted order #16 for : 15860 stocks of WSO2-1 at $ 183.71120268664384
Sat Sep 22 08:55:28 IST 2012 samples.services.SimpleStockQuoteService  ::
Accepted order #17 for : 5165 stocks of WSO2-2 at $ 173.03463846494202
Sat Sep 22 08:55:28 IST 2012 samples.services.SimpleStockQuoteService  ::
Accepted order #18 for : 19535 stocks of WSO2-3 at $ 145.35557152135468
Sat Sep 22 08:55:28 IST 2012 samples.services.SimpleStockQuoteService  ::
Accepted order #19 for : 17457 stocks of WSO2-4 at $ 145.05789309716536
Sat Sep 22 08:55:28 IST 2012 samples.services.SimpleStockQuoteService  ::
Accepted order #20 for : 6260 stocks of WSO2-5 at $ 56.49392913339839
```

The user can also change the method of selecting the starting sequence number and time between activations, preserve duplicate messages, and connect JMS stores.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **messageProcessor** [line 24 in ESB config] - The `messageProcessor` turns the message store into a resequencer. During the interval period, messages coming in are stored and reordered based on the sequence number, which is the value of the `seqNumXpath` parameter's XPath expression.

Composed Msg. Processor

This section explains, through an example scenario, how the Composed Msg. Processor EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Composed Msg. Processor

The Composed Msg. Processor EIP is used to process a composite message. It maintains the overall message flow when processing a message consisting of multiple elements, each of which might require different processing. The Composed Message Processor splits the message up, routes the sub-messages to the appropriate destinations, and then aggregates the responses back into a single message. For more information, refer to http://www.eaipatterns.com/DistributionAggregate.html.

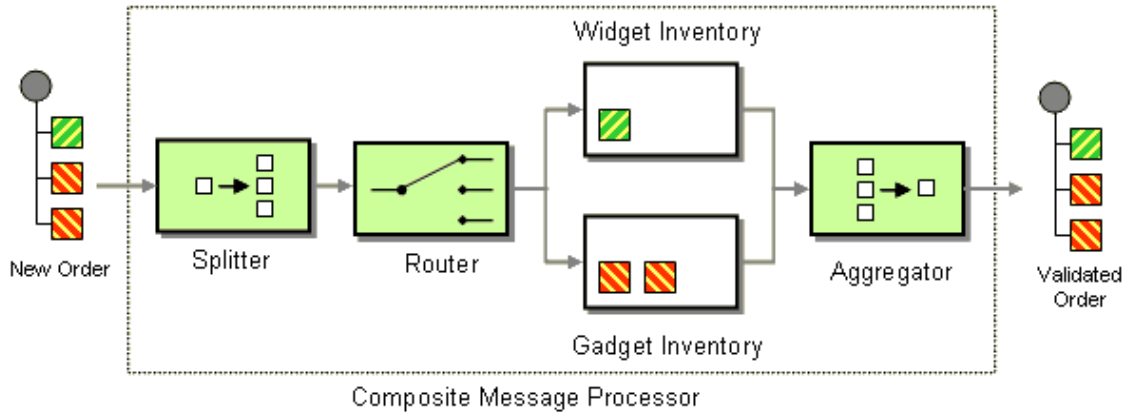


Figure 1: Composed Msg. Processor EIP

Example scenario

This example scenario demonstrates splitting a message into several requests, which are then routed to different servers, merged together, and sent back to the client. In this scenario, each server instance acts as an inventory controller. The user can have multiple requests in a single request message. The Iterate Mediator processes the request message and splits it. The ESB identifies the request content using the Switch mediator, and decides the routing destination. The Send mediator then sends the request message to the respective location (endpoint). The response, which will be sent from different endpoints, will be merged together using the Aggregate mediator of the ESB before sending back to the client.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

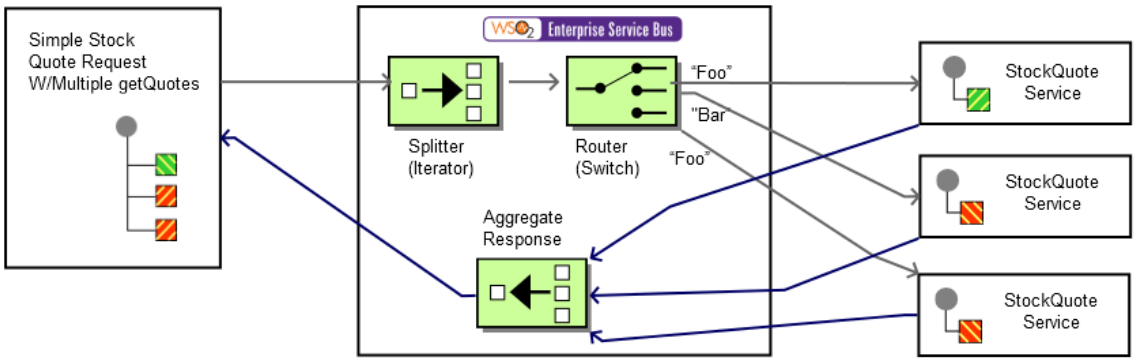


Figure 2: Composed Msg. Processor Example Scenario

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Composed Msg. Processor EIP by comparing their core components.

Composed Msg. Processor EIP (Figure 1)	Composed Msg. Processor Example Scenario (Figure 2)
New Order	Stock Quote Request
Splitter	Iterate Mediator
Router	Switch Mediator
Widget/Gadget Inventory	Stock Quote Service Instance

Aggregator	Aggregate Mediator
Validated Order	Aggregated Stock Quote Response

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances in ports 9000 and 9001. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="ComposedMessageProxy" startOnLoad="true">
      <target>
         <inSequence>
            <log level="full"/>
            <iterate xmlns:m0="http://services.samples"
                     preservePayload="true"
                     attachPath="//m0:getQuote"
                     expression="//m0:getQuote/m0:request">
               <target>
                  <sequence>
                     <switch xmlns:m1="http://services.samples/xsd"
source="//m1:symbol">
                        <case regex="IBM">
                           <send>
                              <endpoint>
                                 <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
                              </endpoint>
                           </send>
                        </case>
                        <case regex="WSO2">
                           <send>
                              <endpoint>
                                 <address
uri="http://localhost:9001/services/SimpleStockQuoteService"/>
                              </endpoint>
                           </send>
                        </case>
                        <default>
                           <drop/>
                        </default>
                     </switch>
                  </sequence>
               </target>
            </iterate>
         </inSequence>
```

```
         <outSequence>
            <aggregate>
               <completeCondition>
                  <messageCount/>
               </completeCondition>
               <onComplete xmlns:m0="http://services.samples"
expression="//m0:getQuoteResponse">
                  <send/>
               </onComplete>
            </aggregate>
         </outSequence>
      </target>
      <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
   </proxy>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
```

```
      <drop/>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send the following request using a SOAP client such as SoapUI.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <ser:request>
            <xsd:symbol>IBM</xsd:symbol>
         </ser:request>
          <ser:request>
            <xsd:symbol>WSO2</xsd:symbol>
         </ser:request>
         <ser:request>
            <xsd:symbol>IBM</xsd:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

Note that the three responses are merged together.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **iterate** [line 7 in ESB config] - The Iterate mediator takes each child element of the element specified in its XPath expression and applies the sequence flow inside the Iterate mediator. In this example, it takes each getQuote request specified in the incoming request and forwards this request to the target endpoint.
- **switch** [line 13 in ESB config] - Observes the message and filters out the message content to the given XPath expression.
- **case** [line 14 in ESB config] - The filtered content will be tallied with the given regular expression.
- **send** [line 15 in ESB config] - When a matching case is found, the Send mediator will route the message to the endpoint indicated in the address URI.
- **aggregate** [line 37 in ESB config] - The Aggregate mediator merges together the response messages for requests made by the Iterate or Clone mediators. The completion condition specifies the minimum or maximum number of messages to be collected. When all messages are aggregated, the sequence inside `onComplete` is run.

Scatter-Gather

This section explains, through an example scenario, how the Scatter-Gather EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Scatter-Gather
- Example scenario

- Environment setup
- ESB configuration
- Simulating the sample scenario
- How the implementation works

Introduction to Scatter-Gather

The Scatter-Gather EIP maintains the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply back. For more information, refer to http://www.eaipatterns.com/BroadcastAggregate.html.

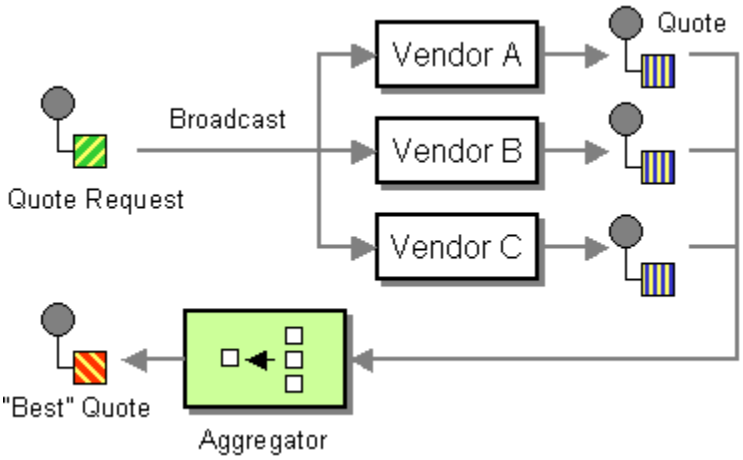


Figure 1: Scatter-Gather EIP

Example scenario

This example scenario demonstrates an implementation of Scatter-Gather EIP that broadcasts a message to multiple recipients using WSO2 ESB. The ESB uses the Aggregate mediator to collect the responses and merge them into a single response message.

We use a sample Stock Quote service as the service provided by the vendors. In this scenario, you send a quote request to three vendors, get quotes for certain items, and return the best quote to the client. We assume that all three vendors implement the same service contract. If the service contracts are different, the ESB must transform the messages before sending them to the vendor services and then transform the responses before returning them to the client. The XSLT mediator is designed to handle these transformations.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

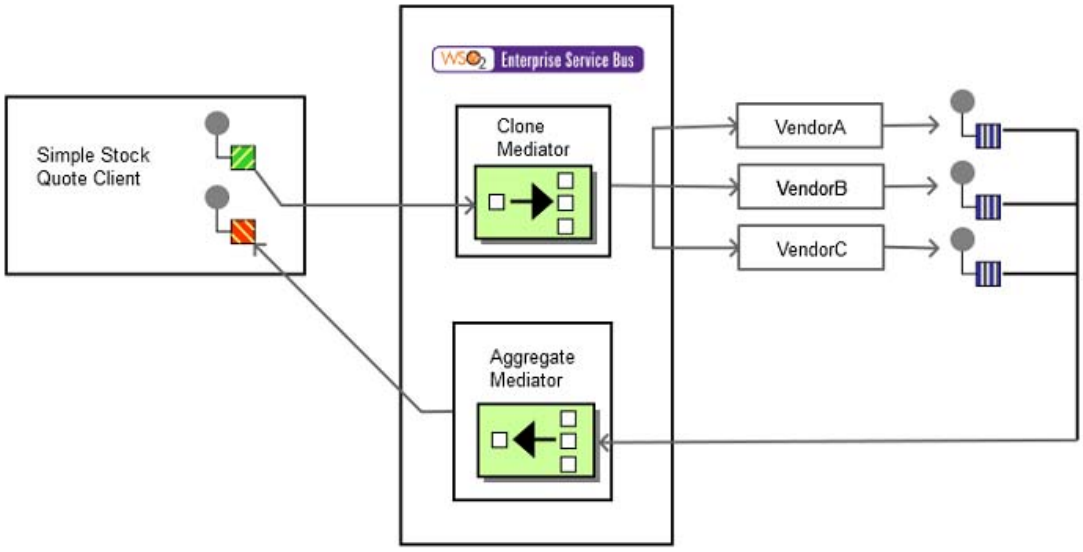


Figure 2: Scatter-Gather Example Scenario

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Scatter-Gather EIP by comparing their core components.

Scatter-Gather EIP (Figure 1)	**Scatter-Gather Example Scenario (Figure 2)**
Quote Request	Simple Stock Quote Request
Broadcast	Clone Mediator
Quote	Simple Stock Quote Service Response
Aggregator	Aggregate Mediator
Best Quote	Aggregated Response

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start three sample Axis2 server instances on ports 9000, 9001, and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Services**, then **Add** and then click **Proxy Service**. Next, copy and paste the following configuration, which helps you explore the example scenario, to a new Pass Through Proxy Service named **ScatterGatherProxy**.

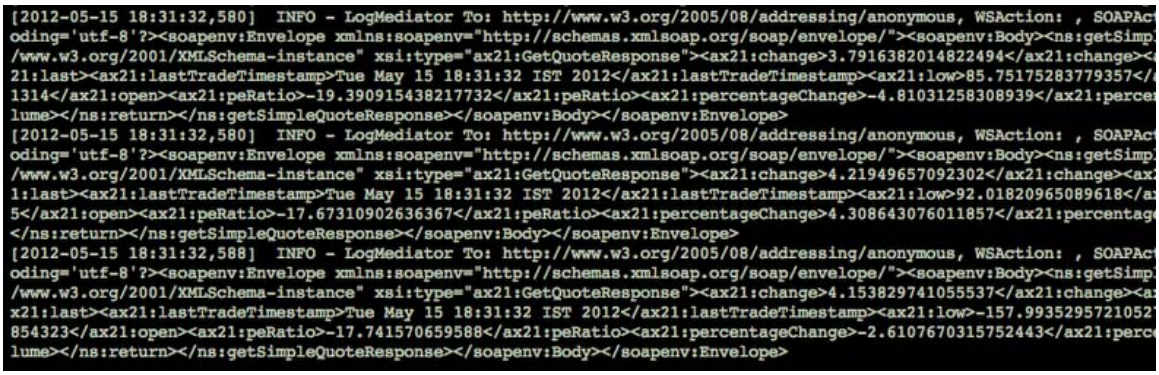
```
<proxy xmlns="http://ws.apache.org/ns/synapse" name="ScatterGatherProxy"
transports="https http" startOnLoad="true" trace="disable">
    <description/>
    <target>
        <inSequence>
            <clone>
                <target>
                    <endpoint name="vendorA">
                        <address
uri="http://localhost:9000/services/SimpleStockQuoteService/"/>
                    </endpoint>
                </target>
                <target>
                    <endpoint name="vendorB">
                        <address
uri="http://localhost:9001/services/SimpleStockQuoteService/"/>
                    </endpoint>
                </target>
                <target>
                    <endpoint name="vendorC">
                        <address
uri="http://localhost:9002/services/SimpleStockQuoteService/"/>
                    </endpoint>
                </target>
            </clone>
        </inSequence>
        <outSequence>
            <log level="full"/>
            <aggregate>
                <completeCondition>
                    <messageCount min="3"/>
                </completeCondition>
                <onComplete xmlns:m1="http://services.samples/xsd"
xmlns:m0="http://services.samples" expression="//m0:return">
                    <enrich>
                        <source xmlns:m1="http://services.samples/xsd" clone="true"
xpath="//m0:return[not(preceding-sibling::m0:return/m1:last &lt;= m1:last) and
not(following-sibling::m0:return/m1:last &lt; m1:last)]"/>
                        <target type="body"/>
                    </enrich>
                    <send/>
                </onComplete>
            </aggregate>
        </outSequence>
    </target>
</proxy>
```

Simulating the sample scenario

1. Use a SOAP client like SoapUI to send the following request to the `ScatterGatherProxy` service.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getSimpleQuote>
         <ser:symbol>Foo</ser:symbol>
      </ser:getSimpleQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

2. Because the log mediator is enabled inside the outSequence, there will be three responses from the three vendors. The logs will be similar to the following:



3. In SoapUI, you will get the response from the vendor providing the best quote as follows:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
      <ns:return xsi:type="ax21:GetQuoteResponse" xmlns:ns="http://services.samples" xmlns:ax21="h
         <ax21:change>3.7916382014822494</ax21:change>
         <ax21:earnings>13.466693987895878</ax21:earnings>
         <ax21:high>85.76671867978592</ax21:high>
         <ax21:last>83.27092828344247</ax21:last>
         <ax21:lastTradeTimestamp>Tue May 15 18:31:32 IST 2012</ax21:lastTradeTimestamp>
         <ax21:low>85.75175283779357</ax21:low>
         <ax21:marketCap>3.688303100863601E7</ax21:marketCap>
         <ax21:name>IBM Company</ax21:name>
         <ax21:open>-83.06280156151314</ax21:open>
         <ax21:peRatio>-19.390915438217732</ax21:peRatio>
         <ax21:percentageChange>-4.81031258308939</ax21:percentageChange>
         <ax21:prevClose>-78.82311463108903</ax21:prevClose>
         <ax21:symbol>IBM</ax21:symbol>
         <ax21:volume>15798</ax21:volume>
      </ns:return>
   </soapenv:Body>
</soapenv:Envelope>
```

4. Compare the logged response messages with the response received by the client to see that the `ScatterGatherProxy` service returns the best quote to the client.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **clone** [line 6 in ESB config] - In the `inSequence` of the `ScatterGatherProxy` service, we use the Clone mediator to make three copies of the request. Those requests are then forwarded to the three vendor

services (`SimpleStockeQuoteService`). The responses to those three requests are received at the `outSequence`. The Clone mediator is similar to the Splitter EIP. It clones the incoming request and passes the requests in parallel to several endpoints.

- **log** [line 25 in ESB config] - All received responses are logged before the Aggregate mediator merges them.
- **aggregate** [line 26 in ESB config] - The Aggregate mediator aggregates response messages for requests made by the Iterate or Clone mediator. The completion condition specifies the minimum or maximum number of messages to be collected.
- **onComplete** [line 30 in ESB config] - When all messages are aggregated, the `onComplete` sequence of the Aggregate mediator will run. This sequence is called once all responses are received or the specified completion condition is met. The responses are aggregated based on the value of the `return` element in the response.
- **enrich** [line 32 in ESB config] - The Enrich mediator is used to extract the response, which contains the best quote. The following XPath 1.0 expression is used for this purpose:

  ```
  //m0:return[not(preceding-sibling::m0:return/m1:last <= m1:last) and
  not(following-sibling::m0:return/m1:last < m1:last)]
  ```

  In essence, this expression instructs the ESB to pick the response that has the lowest `last` value. (The XPath 2.0 `min` function could reduce the complexity of the above expression, but XPath 1.0 is the current default supported by WSO2 ESB.) Once the proper response is found, we enrich the SOAP body with it and send that response back to the client.

Routing Slip

This section explains, through an example scenario, how the Routing Slip EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Routing Slip
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Routing Slip

The Routing Slip EIP routes a message consecutively through a series of processing steps when the sequence of steps is not known at design-time, and may vary for each message. For more information, refer to http://www.eaipatterns.com/RoutingTable.html.

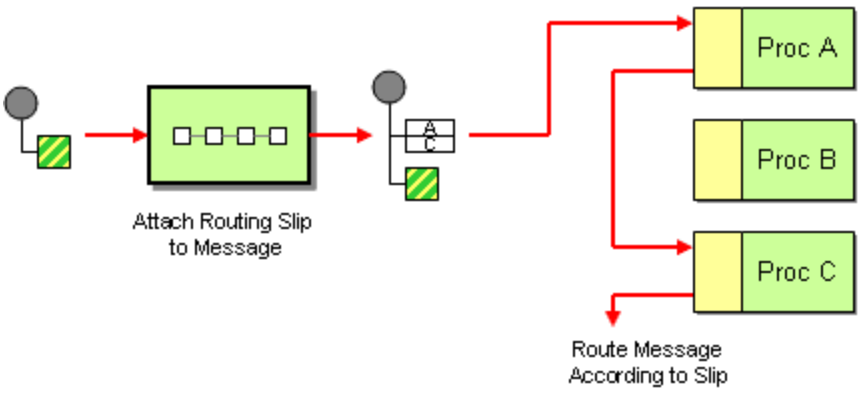


Figure 1: Routing Slip EIP

The sequence in which a message should process usually varies based on the request that arrives. This EIP observes the message at the initial state and attaches a specific list of steps that the message should follow.

Example scenario

In the example scenario, when the ESB receives a message, it will attach a property to the message using the Header mediator to indicate the set of processors that it should follow. It defines each process as a separate sequence. This example consists of three independent sequences. The attached properties are processed using the Iterate mediator, and the process is analyzed using the Switch mediator in each iteration cycle. Once the process is analyzed, the message will be sent to the respective process (sequence).

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

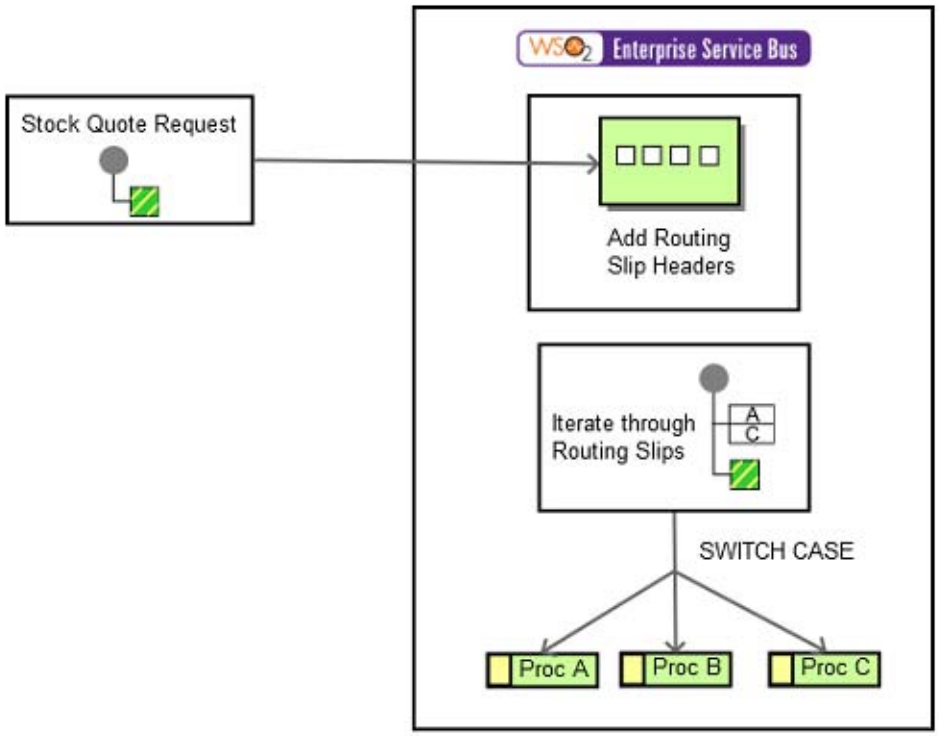


Figure 2: Routing Slip Example Scenario

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Routing Slip EIP by comparing their core components.

Routing Slip EIP (Figure 1)	Routing Slip Example Scenario (Figure 2)
Request Message	Simple Stock Quote Request
Routing Slip	Header Mediator appends node to SOAP header
Proc A/B/C	Sequence

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server instances. For instructions, refer to the section ESB Samples Setup - Starting

Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="RoutingSlipProxy" transports="http https" startOnLoad="true">
      <target>
         <inSequence>
   <!-- Will Attach The Routing Slip -->
            <header xmlns:m1="http://services.samples"
                    name="m1:RoutingSlip"
                    value="Process_A"/>
            <header xmlns:m1="http://services.samples"
                    name="m1:RoutingSlip"
                    value="Process_C"/>
            <log level="full"/>
   <!-- Will Perform All The Steps Based on The Slip -->
            <iterate xmlns:m0="http://services.samples"
                     preservePayload="true"
                     expression="//m0:RoutingSlip">
               <target>
                  <sequence>
                     <switch xmlns:m2="http://services.samples"
source="//m2:RoutingSlip">
                        <case regex="Process_A">
                           <sequence key="process_a"/>
                        </case>
                        <case regex="Process_B">
                           <sequence key="process_b"/>
                        </case>
                        <case regex="Process_C">
                           <sequence key="process_c"/>
                        </case>
                        <default>
                           <drop/>
                        </default>
                     </switch>
                  </sequence>
               </target>
            </iterate>
         </inSequence>
         <outSequence>
            <drop/>
         </outSequence>
      </target>
   </proxy>
   <sequence name="process_b">
      <log level="custom">
         <property name="Process" value="B"/>
      </log>
   </sequence>
   <sequence name="process_c">
```

```
      <log level="custom">
         <property name="Process" value="C"/>
      </log>
   </sequence>
   <sequence name="process_a">
      <log level="custom">
         <property name="Process" value="A"/>
      </log>
      <sequence key="send_seq"/>
   </sequence>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="send_seq">
      <send>
         <endpoint name="simple">
            <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
         </endpoint>
      </send>
   </sequence>
   <sequence name="main">
      <log/>
      <in/>
```

```
        <out/>
    </sequence>
</definitions>
```

Simulating the sample scenario

1. Send a request like the following to the client.

   ```
   ant stockquote -Dtrpurl=http://localhost:8280/services/RoutingSlipProxy
   -Dsymbol=Foo
   ```
2. Note that the steps are attached to the message header initially. Thereafter, processing will be decided based on the attached slip. You can observe process A and process C being logged in the ESB management console.

   You can also allow the message to flow through Process B by indicating a header in the following manner.

   ```
   <header xmlns:m1="http://services.samples"
                       name="m1:RoutingSlip"
                       value="Process_B"/>
   ```

   If you add the above header at the beginning, you will notice the message going through Process B as well.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **header** [line 7 in ESB config] - The Header mediator appends a key/value pair to the SOAP header. It can be used to remove such pairs. In this example, the configuration adds a header field called `RoutingSlip` with a value of Process A. It then adds another header field `RoutingSlip` with a value of Process C.
- **iterate** [line 15 in ESB config] - The Iterate mediator is used to iterate over all `RoutingSlip` nodes inside the header.
- **switch** [line 20 in ESB config] - The Switch mediator is used to filter out and match the value in `RoutingSlip` to run the relevant sequence.

Process Manager

This section explains how the Process Manager EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Process Manager EIP
- How WSO2 ESB implements the EIP

Introduction to Process Manager EIP

The Process Manager EIP routes a message through multiple processing steps, when the required steps might not be known at design-time and might not be sequential. It maintains the state of the sequence and determines the next processing step based on intermediate results. For more information, refer to http://www.eaipatterns.com/ProcessManager.html.

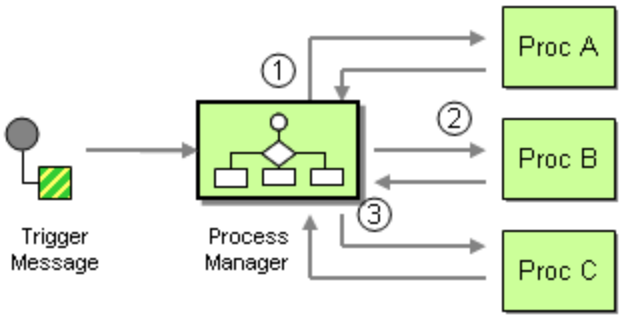


Figure 1: Process Manager EIP

How WSO2 ESB implements the EIP

Message routing can take a different series of processing steps. In the Routing Slip EIP, a message contains the routing path (a sequence of processing steps), which is decided at the design stage. But in many cases the routing decisions have to be made based on intermediate results, and processing steps might need to be executed in parallel. To achieve this dynamic behavior, a processing unit called Process Manager has been introduced, which determines the next processing step based on intermediate results.

Designing and configuring a Process Manager is a fairly big area of study. WSO2 Business Process Server enables you to easily develop business processes using WS-BPEL standards and provides a hosting environment.

Message Broker

This section explains, through an example scenario, how the Message Broker EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Message Broker
- Example scenario
    - Environment setup
    - ESB configuration
    - Simulating the sample scenario
    - How the implementation works

Introduction to Message Broker

The Message Broker EIP decouples the destination of a message from the sender and maintains central control over the flow of messages. It receives messages from multiple destinations, determines the correct destination, and routes the message to the correct channel. The Message Broker EIP decouples messages from senders and receivers. For more information, refer to http://www.eaipatterns.com/MessageBroker.html.

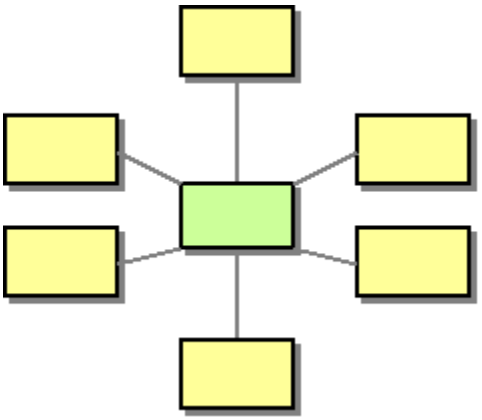


Figure 1: Message Broker EIP

Example scenario

This example scenario demonstrates how WSO2 ESB works with WSO2 Message Broker to implement the Message Broker EIP. In this scenario, sent messages are put into a Message Broker queue, which any interested receiver can consume. If you want to add more receivers, you can use topics in WSO2 Message Broker in a similar manner discussed here.

Environment setup

Setting up WSO2 Message Broker

1. Download a binary version of WSO2 Message Broker from http://wso2.com/products/message-broker, and extract the distribution, which will be referred to as <MB_HOME>.
2. Open <MB_HOME>/repository/conf/carbon.xml file, and change the offset of ports to 1. This is done to ensure that there will be no port conflicts when you run multiple WSO2 products simultaneously on the same server.

```
<Ports>
<Offset>1</Offset>
...
```

3. Change the default virtual host to **carbon** in <MB_HOME>/repository/conf/advanced/qpid-virtualhosts.xml file.

```
<virtualhosts>
    <default>carbon</default>
...
```

4. Start WSO2 MB server by executing wso2server.sh (or wso2server.bat in Windows) file in the <MB_HOME>/bin directory.

Setting up WSO2 ESB

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. Installation home will be referred to as <ESB_HOME>. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Copy the following files from <MB_HOME>/client-lib to <ESB_HOME>/repository/components/lib.
    - geronimo-jms_1.1_spec-1.1.0.wso2v1.jar
    - qpid-client-0.11.0.wso2v2.jar
3. Enable the JMS transport receivers and senders by uncommenting the relevant sections in the <ESB_HOME>/repository/conf/axis2/axis2.xml file. For example:

```
<transportReceiver name="jms"
class="org.apache.axis2.transport.jms.JMSListener">
        <!--Only Enabled settings related to Queue since the sample here only
uses a queue-->
        <parameter name="myQueueConnectionFactory" locked="false">
            <parameter name="java.naming.factory.initial"
locked="false">org.apache.qpid.jndi.PropertiesFileInitialContextFactory</param
eter>
            <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
            <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
            <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
        </parameter>
        <parameter name="default" locked="false">
            <parameter name="java.naming.factory.initial"
locked="false">org.apache.qpid.jndi.PropertiesFileInitialContextFactory</param
eter>
            <parameter name="java.naming.provider.url"
locked="false">repository/conf/jndi.properties</parameter>
            <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
            <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
        </parameter>
</transportReceiver>
.........
<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender"/>
```

4. Define the following properties in the <ESB_HOME>/repository/conf/jndi.properties file.

```
.....
#Need change QueueConnection factory as follows
connectionfactory.QueueConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'

# register some queues in JNDI using the form
# queue.[jndiName] = [physicalName]
destination.myqueue=destinationMyQueue; {create:always}

# register some topics in JNDI using the form - Commented since this sample is
not going to use Topics
# topic.[jndiName] = [physicalName]
#topic.MyTopic = example.MyTopic
```

5. Start the sample Axis2 server on ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (https://localhost:9443/carbon). In the

management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="StockQuoteProxy"
          transports="http"
          startOnLoad="true"
          trace="disable">
      <description/>
      <target>
   <!-- Send message to WSO2 MB -->
         <endpoint>
            <address
uri="jms:/myqueue?transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory&amp
;java.naming.factory.initial=org.apache.qpid.jndi.PropertiesFileInitialContextFactor
y&amp;java.naming.provider.url=repository/conf/jndi.properties&amp;transport.jms.Des
tinationType=queue"/>
         </endpoint>
         <inSequence>
            <property name="OUT_ONLY" value="true"/>
            <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
            <property name="transport.jms.ContentTypeProperty"
                      value="Content-Type"
                      scope="axis2"/>
         </inSequence>
         <outSequence>
            <property name="TRANSPORT_HEADERS" scope="axis2" action="remove"/>
            <send/>
         </outSequence>
      </target>
   </proxy>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
      <drop/>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send a request using Stock Quote client to the proxy service in the following manner. For information on the Stock Quote client, refer to the Sample Clients section in WSO2 ESB documentation.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/StockQuoteProxy -Dsymbol=WSO2
```

Note that the request is stored in WSO2 Message Broker. Any consumer can access the stored message by

accessing `destinationMyQueue` in WSO2 Message Broker.

How the implementation works

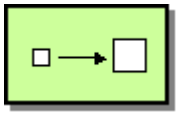
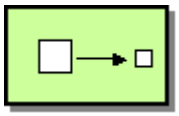
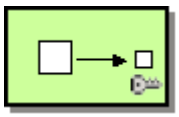
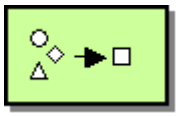
Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **Proxy service** [line 3 in ESB config] - Defines a proxy service named `StockQuoteProxy`.
- **endoint** [line 10 in ESB config] - Defines an endpoint inside the proxy service. The address of the endpoint is a JMS URL. The JMS URL is made up of the following elements:
    - **jms:/myqueue -** Looks for a JNDI entry `myqueue` (see JNDI properties above).
    - **?** - Separator indicating extra attributes.
    - **transport.jms.ConnectionFactoryJNDIName=QueueConnectionFactory -** Looks up `ConnectionFactory` in JNDI with name `QueueConnectionFactory`.
    - **&** - Separator (this will convert to '&')
    - **java.naming.factory.initial=org.apache.qpid.jndi.PropertiesFileInitialContextFactory -** Uses the Qpid properties-based JNDI.
    - **&** - Another separator (this will convert to '&')
    - **java.naming.provider.url=repository/conf/jndi.properties -** Looks in `repository/conf/jndi.properties` for the JNDI properties file.

Message Transformation

One challenge of data communication is that the formats of data and their storage mechanisms vary among different systems. A message translator is an architectural pattern that acts as a filter between other filters or applications to translate one data format to another, transforming your message as it passes through the ESB.

This chapter introduces varieties of message translators and how each can be simulated using the WSO2 ESB.

	Envelope Wrapper	How existing systems participate in a messaging exchange, which places specific requirements in the message format, such as message header fields or encryption.
	Content Enricher	How to communicate with another system if the message originator does not have all the required data items available.
	Content Filter	How to simplify dealing with a large message when you are interested only in a few data items.
	Claim Check	How to reduce the data volume of a message sent across the system without sacrificing information content.
	Normalizer	How to process messages that are semantically equivalent but arrive in a different format.
	Canonical Data Model	How to minimize dependencies when integrating applications that use different data formats.

Envelope Wrapper

This section explains, through an example scenario, how the Envelope Wrapper EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Envelope Wrapper
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Envelope Wrapper

The Envelope Wrapper EIP allows existing systems to participate in a messaging exchange that places specific requirements on the message format, such as message header fields or encryption. It wraps application data inside an envelope that is compliant with the messaging infrastructure. The message is unwrapped when it arrives at the destination. For more information, refer to http://www.eaipatterns.com/EnvelopeWrapper.html.

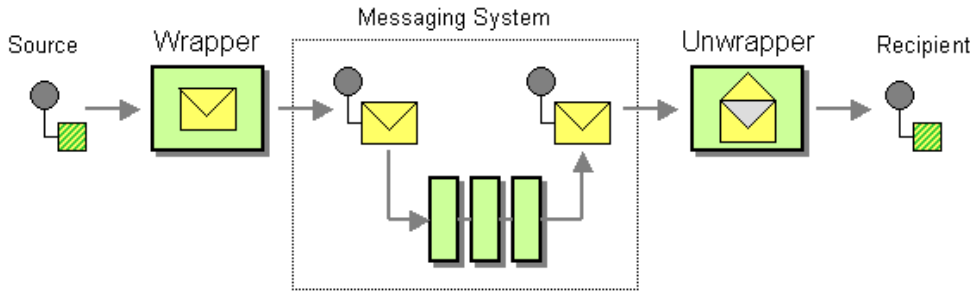


Figure 1: Envelope Wrapper EIP

Example scenario

This example scenario receives a message with application data wrapped inside an envelope, unwraps the message, and sends it to a specific endpoint. The sender sends the request inside a SOAP envelope. Once the ESB receives the envelope, it unwraps it and sends it as a Plain Old XML (POX) request to the sample back-end Axis2 server.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

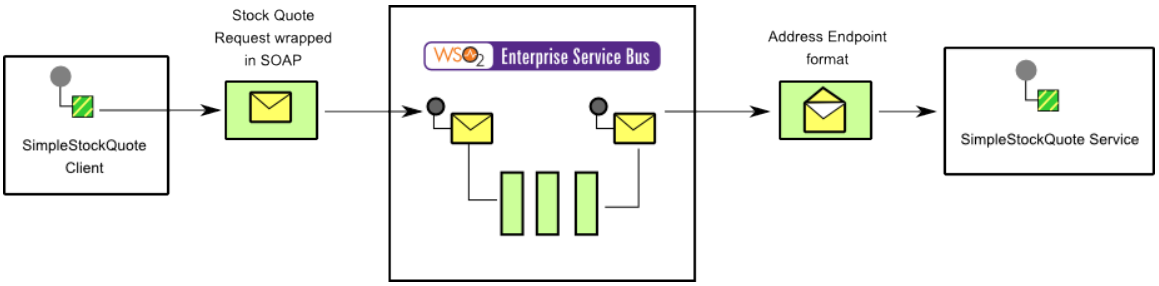


Figure 2: Example Scenario of the Envelope Wrapper EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Envelope Wrapper EIP by comparing their core components.

Envelope Wrapper EIP (Figure 1)	Envelope Wrapper Example Scenario (Figure 2)
Wrapper	Stock Quote Request wrapped in SOAP
Messaging System	WSO2 ESB
Unwrapper	Address Endpoint format
Recipient	Stock Quote Service Instance

> An alternative implementation of this EIP is to have the Address Endpoint wrap from one envelope format to another (for example, wrapping a SOAP 1.1 envelope in a SOAP 1.2 envelope).

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample

Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="EnvelopeUnwrapProxy"
          transports="https http"
          startOnLoad="true">
      <target>
         <endpoint>
            <address uri="http://localhost:9000/services/SimpleStockQuoteService"
                     format="pox"/>
         </endpoint>
         <outSequence>
            <send/>
         </outSequence>
      </target>
      <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
   </proxy>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <send/>
   </sequence>
</definitions>
```

Simulating the sample scenario

1. Send the following request using a SOAP client like SoapUI, and monitor the message using TCPMon.

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

2. Notice that the request data is inside a SOAP envelope. When the request was monitored through TCPMon before it was sent to the ESB, it was structured as follows:

```
POST /services/EnvelopeUnwrapProxy HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: "urn:getQuote"
Content-Length: 385
Host: 127.0.0.1:8281
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)

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

3. The request sent to the back-end Axis2 server has the following structure:

```
POST /services/SimpleStockQuoteService HTTP/1.1
Content-Type: application/xml; charset=UTF-8
Accept-Encoding: gzip,deflate
SOAPAction: urn:getQuote
Transfer-Encoding: chunked
Host: localhost:9000
Connection: Keep-Alive
User-Agent: Synapse-HttpComponents-NIO

e0
<ser:getQuote xmlns:ser="http://services.samples">
         <ser:request>
            <xsd:symbol
xmlns:xsd="http://services.samples/xsd">Foo</xsd:symbol>
         </ser:request>
      </ser:getQuote>
0
```

This means that the SOAP envelope was removed by the ESB.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **address** [line 8 in ESB config] - The endpoint address contains the attribute `format='pox'`, which makes the ESB convert incoming requests to Plain Old XML. Other supported formats for wrapping include soap11, soap12 and get. For more information, refer to the Address Endpoint mediator.

Content Enricher

This section explains, through an example scenario, how the Content Enricher EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Content Enricher

The Content Enricher EIP facilitates communication with another system if the message originator does not have all the required data items available. It accesses an external data source to augment a message with missing information. For more information, refer to http://www.eaipatterns.com/DataEnricher.html.

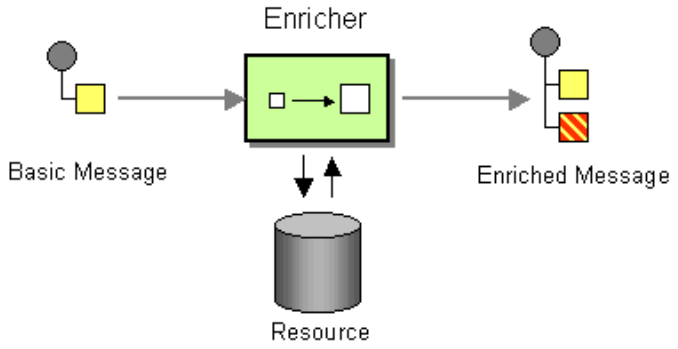


Figure 1: Content Enricher EIP

Example scenario

This example scenario depicts a stock quote service. The client sends a stock quote request to the ESB with only an identity number. But in order to provide a stock quote, the sample Axis2 server at the back-end needs to map the identity number with a corresponding name, which is in an external source. The values are stored in the registry as a local entry. When the request arrives, the identity will be analyzed using the Switch mediator. Sequentially, the identity number will be replaced with the local entry using the Enrich mediator.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

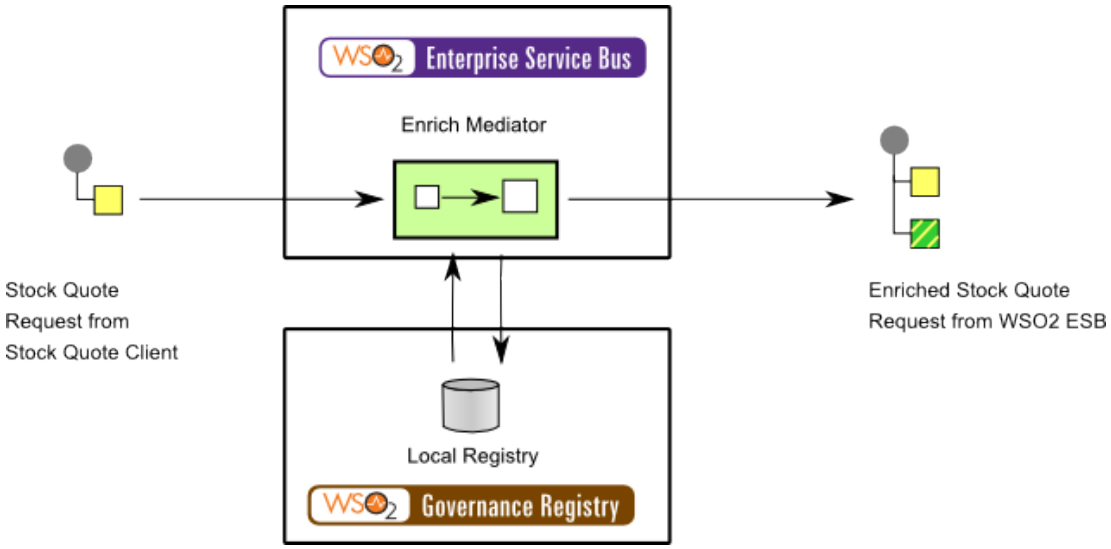


Figure 2: Example Scenario of the Content Enricher EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Content Enricher EIP by comparing their core components.

Content Enricher EIP (Figure 1)	Content Enricher Example Scenario (Figure 2)
Basic Message	Stock Quote Request from Stock Quote Client
Enricher	Enrich Mediator
Resource	Local Registry

Enriched Message	Enriched Stock Quote Request from WSO2 ESB

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- Content Enricher -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
     <proxy name="ContentEnrichProxy">
        <target>
        <inSequence>
            <!-- Would Enrich the Value Based On the Number -->
                  <switch source="//m1:symbol" xmlns:m0="http://services.samples"
xmlns:m1="http://services.samples/xsd">
                <case regex="1">
                     <log level="full" />
                      <enrich>
                        <source type="inline" key="Location1"/>
                        <target xmlns:m1="http://services.samples/xsd"
xpath="//m1:symbol/text()"/>
                                    </enrich>
                 </case>
                 <case regex="2">
                     <enrich>
                        <source type="inline" key="Location2"/>
                        <target xmlns:m1="http://services.samples/xsd"
xpath="//m1:symbol/text()"/>
                                    </enrich>
                 </case>
                </switch>
                  <!--Will Send the Enriched Message -->
                  <send>
                    <endpoint>
                       <address
uri="http://localhost:9000/services/SimpleStockQuoteService" />
                    </endpoint>
                  </send>
                <!--  <drop />    -->
        </inSequence>
        <outSequence>
                <send />
        </outSequence>
      </target>
    <publishWSDL uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
    </proxy>
    <localEntry key="Location1">IBM</localEntry>
    <localEntry key="Location2">WSO2</localEntry>
</definitions>
```

Simulating the sample scenario

Send the following request to the ESB using a SOAP client like SoapUI.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <xsd:symbol>2</xsd:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

Note that the Axis2 server log displays that the request is processed.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **enrich** [line 10 in ESB config] - Mediator used for message enrichment.
- **source** [line 11 in ESB config] - The location in which you can find the source configuration. In this example, it is a simple inline text string located in the local registry entry with key **Location1**.
- **target** [line 12 in ESB config] - The location where the source configuration should be applied. This is specified using an XPath expression.
- **localEntry** [lines 36 and 37 in ESB config] - Entries from the local registry.

Content Filter

This section explains, through an example scenario, how the Content Filter EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Content Filter
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Content Filter

The Content Filter EIP helps manage a large message when you are interested in a few data items only. It removes unimportant data items from a message and leaves only the important ones. In addition to removing data elements, Content Filter can be used to simplify a message structure. For more information, refer to http://www.eaipatterns.com/ContentFilter.html.

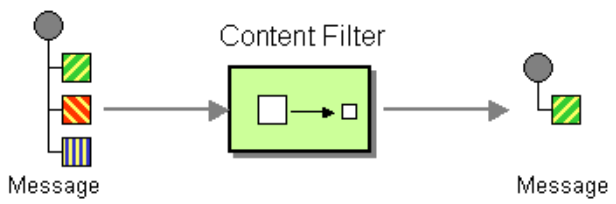


Figure 1: Content Filter EIP

Example scenario

This example scenario explains how a message's content can be optimized to gain a more productive response. A client sends a stock quote request containing additional data that is not necessary for message processing. WSO2 ESB uses an XSLT mediator to restructure the request message and optimize it with only the required data before sending it to the Stock Quote service. This approach improves performance, as removing unnecessary bits of data makes the message more lightweight.

The diagram below depicts how to simulate the example scenario in WSO2 ESB.

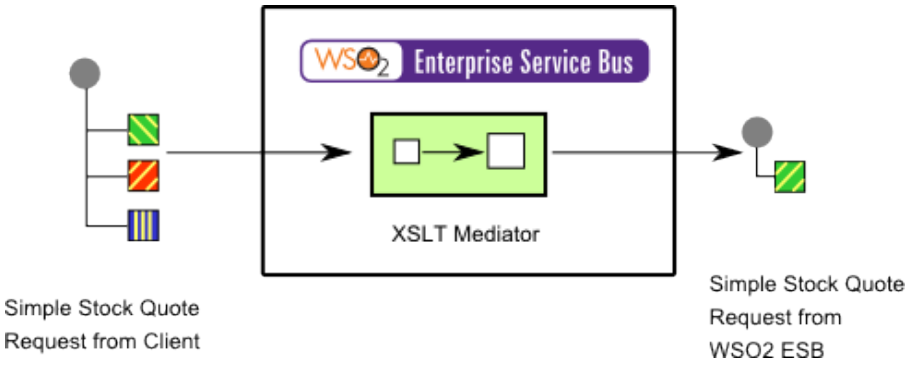


Figure 2: Example Scenario of the Content Filter EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Content Filter EIP by comparing their core components.

Content Filter EIP (Figure 1)	Content Filter Example Scenario (Figure 2)
Original Message	Simple Stock Quote Request from Client
Content Filter	XSLT Mediator
Filtered Message	Simple Stock Quote Request from WSO2 ESB

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.
3. Save the following XSLT document as `<ESB_HOME>/repository/samples/resources/transform/split_message.xslt`.


```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="2.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:fn="http://www.w3.org/2005/02/xpath-functions"
                xmlns:m0="http://services.samples"
                exclude-result-prefixes="m0 fn">
    <xsl:output method="xml" omit-xml-declaration="yes" indent="yes"/>

     <xsl:template match="/">

        <m:getQuote xmlns:m="http://services.samples">
            <m:request>
                <m:symbol>
                    <xsl:value-of select="//m0:getQuote/m0:request[1]"/>
                </m:symbol>
            </m:request>
        </m:getQuote>

    </xsl:template>
</xsl:stylesheet>
```

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- Content Filter -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <registry provider="org.wso2.carbon.mediation.registry.ESBRegistry">
        <parameter name="root">file:./repository/samples/resources/</parameter>
        <parameter name="cachableDuration">15000</parameter>
    </registry>
    <!-- define the request processing XSLT resource as a static URL source -->
    <localEntry key="filter_data"
src="file:repository/samples/resources/transform/split_message.xslt"/>
      <proxy name="ContentFilterProxy">
        <target>
        <inSequence>
             <xslt key="filter_data"/>
             <log level="full" />
             <send>
                <endpoint>
                    <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
                </endpoint>
             </send>
        </inSequence>
        <outSequence>
            <log level="full" />
                <send />
        </outSequence>
      </target>
    <publishWSDL uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
   </proxy>
</definitions>
```

Simulating the sample scenario

Send the following request to the ESB using a SOAP client like SoapUI.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <ser:symbol>1</ser:symbol>
         </ser:request>
          <ser:request>
            <!--Optional:-->
            <ser:symbol>2</ser:symbol>
         </ser:request>
         <ser:request>
            <!--Optional:-->
            <ser:symbol>3</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

When a request arrives to the ESB, it is first simplified to the following structure using the XSLT mediator. You can use TCPMon to observe this behavior.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <ser:request>
            <ser:symbol>1</ser:symbol>
         </ser:request>
       </ser:getQuote>
   </soapenv:Body>
```

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **localEntry** [line 8 in ESB config] - Entry from the local registry.
- **xslt** [line 12 in ESB config] - The XSLT mediator applies an XSLT transformation to the selected element in the message specified using the `source` attribute. In this case, no source attribute is given, so the transformation will be applied to the first child element of the message.

Claim Check

This section explains, through an example scenario, how the Claim Check EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Claim Check

Introduction to Claim Check

The Claim Check EIP reduces the data volume of messages sent across a system without sacrificing information content. It stores the entire message at the initial stage of a sequence of processing steps, and it extracts only the parts required by the preceding steps. Once processing is completed, it retrieves the stored message and performs any operations. This pattern ensures better performance, since large chunks of unwanted data are reduced to lightweight bits before being processed. For more information, refer to http://www.eaipatterns.com/StoreInLibrary.html.

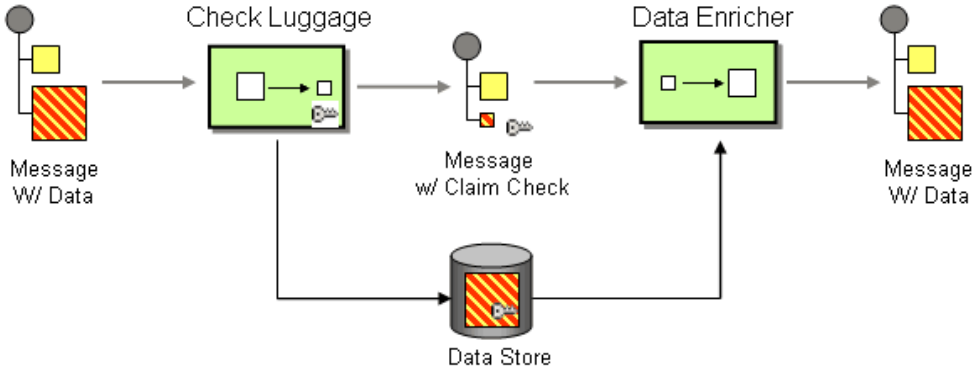


Figure 1: Claim Check EIP

Example scenario

The following scenario illustrates an instance where a stock quote requires an authentication in order for it to allow access to the back-end service on the Axis2 server. For authentication, it is not necessary for the whole message to flow through the mediation. Instead, initially the whole request will be stored in a property using the Enrich mediator, and the request will then be filtered to contain only the user name. The filtered message will be taken through the authentication step by the Filter mediator. If the authentication succeeds, the original content will be retrieved from the property by the Enrich mediator, and the whole message will be forwarded to the Axis2 server.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

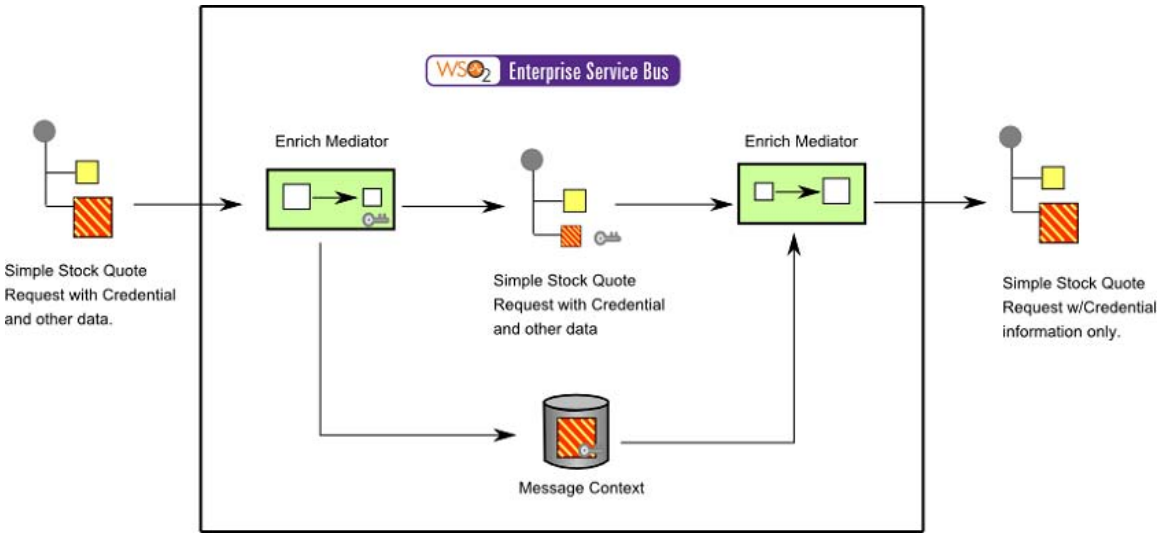


Figure 2: Example Scenario of the Claim Check EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Claim Check EIP by comparing their core components.

Claim Check EIP (Figure 1)	Claim Check Example Scenario (Figure 2)
Message w/Data from Client	Simple Stock Quote Request with Credential and other data.
Check Luggage	Enrich Mediator is used to store the original message as a new property inside a Message Context.
Data Store	Message Context
Message w/Claim Check	Simple Stock Quote Request w/Credential information only.
Data Enricher	Enrich Mediator is used to replace the SOAP Payload body with the original message stored as a property in the Message Context.
Message w/Data from WSO2 ESB	Simple Stock Quote Request with Credential and other data.

> An alternative implementation of this EIP is to use an actual data store instead of appending and replacing the SOAP payload.

Environment setup

1. Download and install the WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the

management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- Claim Check -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
      <proxy name="ClaimCheckProxy">
        <target>
        <inSequence>
                <!-- First The Whole Message Will Be Stored -->
                <enrich>
                    <source type="body"/>
                    <target type="property" property="CLAIM_STORE"/>
                </enrich>
                 <log level="custom" xmlns="http://ws.apache.org/ns/synapse">
                      <property name="text"
expression="get-property('CLAIM_STORE')"/>
                 </log>
                <!-- Will Filter The Content Through Restructuring The Required
Information -->
                <payloadFactory>
                <format>
                    <m:RequiredInformation
xmlns:m="http://services.samples">$1</m:RequiredInformation>
                </format>
                <args>
                    <arg xmlns:m0="http://services.samples"
expression="//m0:credentials/m0:name"/>
                </args>
                </payloadFactory>
                <!-- Will Filter The Content Using The Re Structured Message -->
                <!-- Processing Steps -->
                <filter xmlns:m1="http://services.samples"
source="//m1:RequiredInformation" regex="UserName">
                <then>
                    <property name="Validity" value="1" />
                </then>
                <else>
                    <property name="Validity" value="0" />
                    <drop />
                </else>
                </filter>
                <!-- Based On The Validity Will Route The Original Message To The
End Point -->
                <filter source="get-property('Validity')" regex="1">
                <then>
                    <!-- Retrive The Original Message Back -->
                    <enrich>
                        <source type="property" property="CLAIM_STORE" />
                        <target action="replace" type="body" />
                    </enrich>
                    <log level="full" />
                     <send>
                        <endpoint>
                            <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
                        </endpoint>
                     </send>
                </then>
```

```
            <else>
            </else>
            </filter>
      </inSequence>
          <outSequence>
          <send />
      </outSequence>
    </target>
```

```
     <publishWSDL uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
   </proxy>
</definitions>
```

Simulating the sample scenario

Send the following request using a SOAP client like SoapUI.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header>
     <ser:credentials>
          <ser:name>UserName</ser:name>
          <ser:id>001</ser:id>
      </ser:credentials>
     </soapenv:Header>
   <soapenv:Body>
        <ser:getQuote>
         <ser:request>
             <ser:symbol>Foo</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

The message is optimized as shown below to go through the authentication process inside the ESB. Once the authentication is done, the original message will be attached back to the payload, and sent to the back-end service.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header />
    <soapenv:Body>
 <m:RequiredInformation
xmlns:m="http://services.samples">UserName</m:RequiredInformation>
 </soapenv:Body>
</soapenv:Envelope>
```

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **enrich** [line 7 in ESB config] - Enrich mediator is used to append the original message body as a new property `CLAIM_STORE` inside the message context.
- **payloadFactory** [line 15 in ESB config] - The original message is simplified to contain credential information only.
- **filter** [line 25 in ESB config] - A filter is used to check if the credential information exists inside the new message body. The property `Validity` is set based on this.
- **enrich** [line 38 in ESB config] - Once the validity is set, another mediator uses the validity setting to retrieve the original message stored in the `CLAIM_STORE` context and replaces the body of the SOAP payload with

it.

Normalizer

This section explains, through an example scenario, how the Normalizer EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Normalizer
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Normalizer

The Normalizer EIP processes messages that are semantically equivalent but arrive in different formats. It routes each message type through a custom Message Translator so that the resulting messages match a common format. For more information, refer to http://www.eaipatterns.com/Normalizer.html.

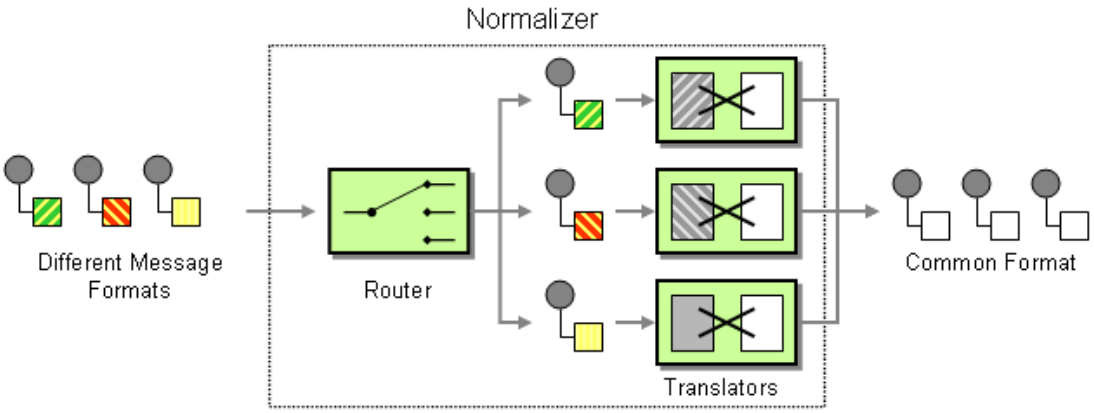


Figure 1: Normalizer EIP

Example scenario

This example scenario demonstrates how WSO2 ESB handles messages it receives in different formats using Message Builders and Message Formatters. The client application and back-end service do not have to be concerned about message formats, because the ESB processes them and sends the responses back in the same format.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

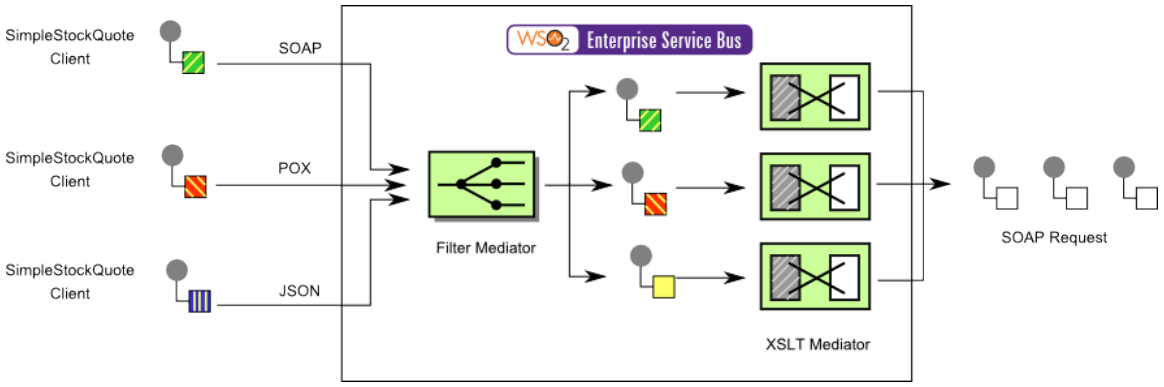


Figure 2: Example Scenario of the Normalizer EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Normalizer EIP by comparing their core components.

Normalizer EIP (Figure 1)	Normalizer Example Scenario (Figure 2)
Different Message Formats	SOAP, POX, or JSON Stock Quote Request
Router	Filter Mediator routes messages based on an existing XPath expression, which identifies what format the message is in.
Translators	XSLT Mediator
Common Format Message	SOAP Request from WSO2 ESB

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <!-- The proxy service to receive all kinds of messages -->
   <proxy name="ServiceProxy"   transports="https http"  startOnLoad="true"
trace="disable">
      <description/>
      <target>
         <inSequence>
            <log level="full"/>
   <!-- Filters incoming JSON messages -->
            <filter xmlns:m0="http://services.samples"
xpath="//m0:getQuote/m0:request/m0:symbol">
```



```
               <then>
                  <sequence key="sendSeq"/>
               </then>
               <else>
                  <sequence key="jsonInTransformSeq"/>
               </else>
            </filter>
         </inSequence>
         <outSequence>
   <!-- Filters outgoing JSON messages -->
            <filter source="get-property('TRANSFORMATION')" regex="JSONtoSOAP">
               <then>
                  <property name="messageType"  value="application/json"
scope="axis2" type="STRING"/>
               </then>
            </filter>
            <log level="full"/>
            <send/>
         </outSequence>
      </target>
   </proxy>
   <localEntry key="in_transform">
      <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                      xmlns:fn="http://www.w3.org/2005/02/xpath-functions"
                      xmlns:m0="http://services.samples"
                      version="2.0"
                      exclude-result-prefixes="m0 fn">
         <xsl:output method="xml" omit-xml-declaration="yes" indent="yes"/>
         <xsl:template match="*">
            <xsl:element name="{local-name()}" namespace="http://services.samples">
               <xsl:copy-of select="attribute::*"/>
               <xsl:apply-templates/>
            </xsl:element>
         </xsl:template>
      </xsl:stylesheet>
   </localEntry>
 <!-- Transform a JSON message -->
   <sequence name="jsonInTransformSeq">
      <xslt key="in_transform"/>
      <property name="TRANSFORMATION"   value="JSONtoSOAP"  scope="default"
type="STRING"/>
      <sequence key="sendSeq"/>
   </sequence>
   <!-- Normal flow of messages -->
   <sequence name="sendSeq">
      <send>
         <endpoint>
            <address uri="http://localhost:9000/services/SimpleStockQuoteService"
format="soap11"/>
         </endpoint>
      </send>
   </sequence>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default 'fault' sequence"/>
         <property xmlns:ns="http://org.apache.synapse/xsd"  name="ERROR_CODE"
expression="get-property('ERROR_CODE')"/>
         <property xmlns:ns="http://org.apache.synapse/xsd"  name="ERROR_MESSAGE"
expression="get-property('ERROR_MESSAGE')"/>
```

```
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
```

```
        <drop/>
    </sequence>
</definitions>
```

The configuration above first filters out the JSON messages to do the necessary transformation, since the back-end service understands only SOAP messages. JSON messages are also filtered in the `out` sequence. Most of these transformations are done at the code level.

Simulating the sample scenario

You can test this configuration for JSON, SOAP, and POX messages using the sample Axis2 client that comes with WSO2 ESB. You can find examples below.

- For SOAP: `ant stockquote -Dtrpurl=http://localhost:8280/services/ServiceProxy`
- For POX: `ant stockquote -Dtrpurl=http://localhost:8280/services/ServiceProxy`. Note that the XML message has the exact format the service wants. Otherwise, you should use the Payload Factory Mediator to create the required messages from the incoming message.
- For JSON: `ant jsonclient -Daddurl=http://localhost:8280/services/ServiceProxy`

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **filter** [line 9 in ESB config] - The Filter mediator looks for a particular XPath expression inside the request message. If the expression evaluates successfully, it is assumed to be a SOAP or POX message, and the mediation continues through the sequence `sendSeq`. If the expression does not evaluate, it is assumed to be a JSON message, and the mediation continues via the `jsonInTransformSeq` sequence.
- **localEntry** [line 30 in ESB config] - The local entry holds an XSL transformation that converts JSON requests to XML.
- **xslt** [line 47 in ESB config] - The XSLT mediator applies the defined XSLT to the payload.
- **address** [line 55 in ESB config] - The address element of the endpoint defines the back-end service and the message format that back-end service prefers. This format is used to normalize a message further, but only when there can be a 1-to-1 mapping between two different formats, for example, between SOAP 1.1 and SOAP 1.2.

Canonical Data Model

This section explains how the Canonical Data Model EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Canonical Data Model
- How WSO2 ESB implements the EIP

Introduction to Canonical Data Model

The Canonical Data Model EIP minimizes dependencies when integrating applications that use different data formats. It is independent from any specific application and requires each application to produce and consume messages in this common format. For more information, refer to http://www.eaipatterns.com/CanonicalDataModel.html.

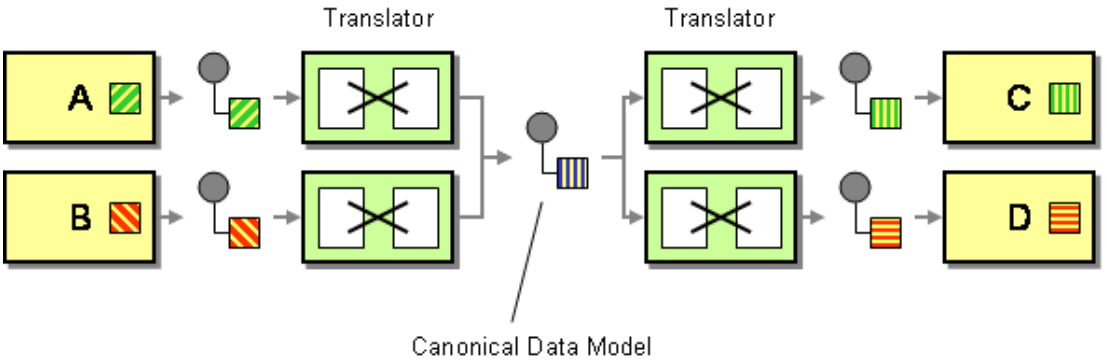


Figure 1: Canonical Data Model EIP

How WSO2 ESB implements the EIP

The Canonical Data Model EIP minimizes dependencies between applications that use different data formats in messaging systems. This model ensures loose-coupling between applications.

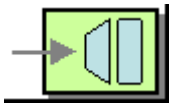
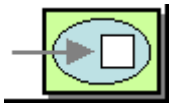
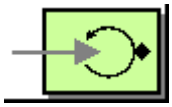
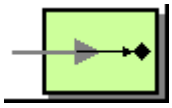
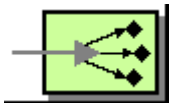
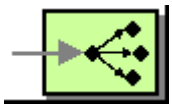
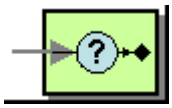
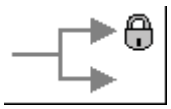
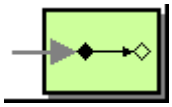
WSO2 ESB supports a number of different data formats including Plain Old XML (POX), JSON, and SOAP. Translating to a common data model (SOAP in WSO2 ESB) and back to the original format is done using the underlying implementations of message builders and message formatters. If the receiving messages are not in the format the back-end service requires, users can use the XSLT mediator or create a new message in SOAP format using the PayloadFactory mediator.

Also see Normalizer for sample scenarios and explanations.

Messaging Endpoints

An endpoint is used to connect an application to a messaging channel so that the application can send or receive messages.

This chapter introduces various endpoint patterns and how each can be simulated using WSO2 ESB.

	Messaging Gateway	How to encapsulate access to the messaging system from the rest of the application.
	Messaging Mapper	How to move data between domain objects and the messaging infrastructure, while keeping the two independent of each other.
	Transactional Client	How a client controls its transactions with the messaging system.
	Polling Consumer	How an application consumes a message when the application is ready.
	Event-Driven Consumer	How an application automatically consumes messages as they become available.
	Competing Consumers	How a messaging client processes multiple messages concurrently.
	Message Dispatcher	How multiple consumers on a single channel coordinate their message processing.
	Selective Consumer	How a message consumer selects which messages to receive.
	Durable Subscriber	How a subscriber avoids missing messages while it is not listening for them.
	Idempotent Receiver	How a message receiver deals with duplicate messages.
	Service Activator	How an application designs a service to be invoked via both messaging and non-messaging techniques.

Messaging Gateway

This section explains, through an example scenario, how the Messaging Gateway EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Messaging Gateway

Introduction to Messaging Gateway

The Messaging Gateway EIP encapsulates message-specific code from the rest of the application. It is a class that wraps messaging-specific method calls and exposes domain-specific methods to the application. Only the Messagin g Gateway knows about the actual implementation of the messaging system. The rest of the application calls the methods of the Messaging Gateway, which are exposed to external applications. For more information, refer to http://www.eaipatterns.com/MessagingGateway.html.

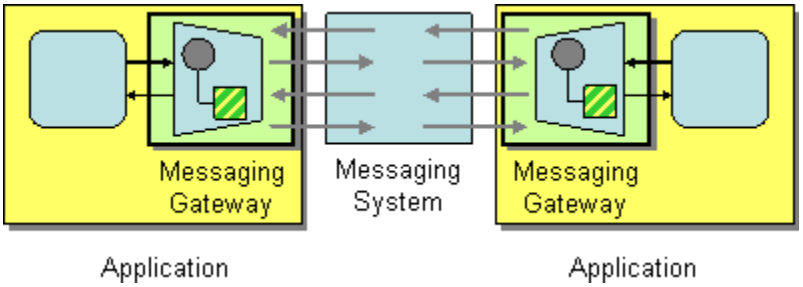


Figure 1: Messaging Gateway EIP

Example scenario

This example scenario demonstrates creating a proxy service with a `publishWSDL` element. The published WSDL's methods act as the Message Gateway, hiding details of the actual back-end service, and exposing only domain-specific methods to the client application.

Proxy services in WSO2 ESB act as Messaging Gateways, abstracting the details of the actual back-end services from implementing clients. For a more complex example of how WSO2 ESB can act as a Messaging Gateway, refer to Health Care Scenario, where a single Proxy Service acts as a Messaging Gateway between several back-end services.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

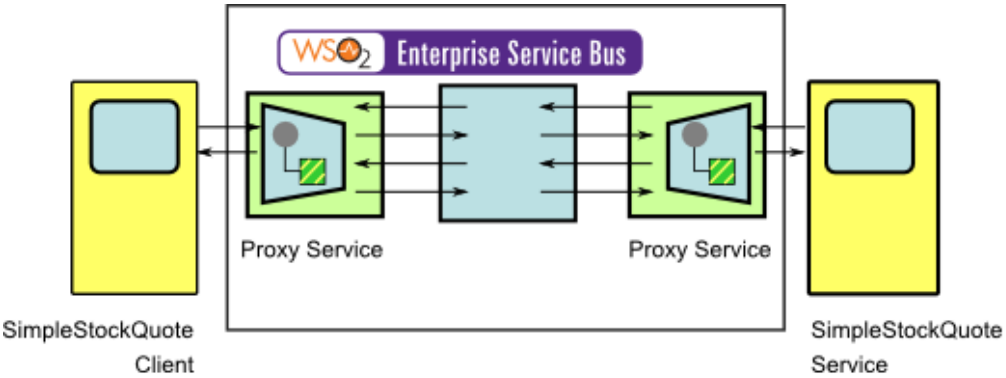


Figure 2: Example Scenario of the Messaging Gateway EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and

the Messaging Gateway EIP by comparing their core components.

Messaging Gateway EIP (Figure 1)	Messaging Gateway Example Scenario (Figure 2)
Application	Simple Stock Quote Client / Service
Messaging Gateway	Proxy Service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.
3. Copy the sample_proxy_3.wsdl file into your `<ESB_HOME>/repository/samples/resources/proxy` directory.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="StockQuoteProxy" startOnLoad="true">
      <target>
         <endpoint>
            <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
         </endpoint>
         <outSequence>
            <send/>
         </outSequence>
      </target>
      <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_3.wsdl"/>
   </proxy>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
      <drop/>
   </sequence>
</definitions>
```

Simulating the sample scenario

If you navigate to `http://localhost:9000/services/SimpleStockQuoteService`, you can see the WSDL

file of the back-end server. There are five methods exposed externally, but the Proxy Service `SimpleQuoteProxy` exposes only four externally, filtering out the `getFullQuote` method. See the `SimpleQuoteProxy` WSDL file in http://localhost:8280/services/StockQuoteProxy?wsdl.

Send the following request using a SOAP client like SoapUI to the `SimpleQuoteProxy` service.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:ser="http://services.samples"
xmlns:xsd="http://services.samples/xsd">
   <soap:Header/>
   <soap:Body>
      <ser:getFullQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <xsd:symbol>WSO2</xsd:symbol>
         </ser:request>
      </ser:getFullQuote>
   </soap:Body>
</soap:Envelope>
```

After sending the above message to the server, you'll get a server error as 'The endpoint reference (EPR) for the Operation not found is /services/StockQuoteProxy and the WSA Action = urn:getFullQuote. The reason for this error is that the `getFullQuote` method is not exposed to `SimpleQuoteProxy`, although the back-end server supports it.

Now, specify a different published WSDL file as follows and send the same SOAP message to the server again.

```
...
<publishWSDL uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
...
```

Note that you get the correct response from the server, since the new WSDL of the proxy service is the same as the back-end service.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **proxy** [line 2 in ESB config] - Defines a new proxy service called `StockQuoteProxy`.
- **endpoint** [line 4 in ESB config] - Defines the endpoint of the actual back-end service that this proxy service is connected to.
- **publishWSDL** [line 11 in ESB config] - Defines the WSDL file to expose for this proxy service. If no `publishWSDL` is given, the actual back-end service's WSDL is exposed.

Messaging Mapper

This section explains how the Messaging Mapper EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Messaging Mapper
- How WSO2 ESB implements the EIP

Introduction to Messaging Mapper

The Messaging Mapper EIP moves data between domain objects and the messaging infrastructure, while keeping the two independent of each other. It contains the mapping logic between the messaging infrastructure and the domain objects. Neither the objects nor the infrastructure have knowledge of the Messaging Mapper's existence. For more information, refer to http://www.eaipatterns.com/MessagingMapper.html.

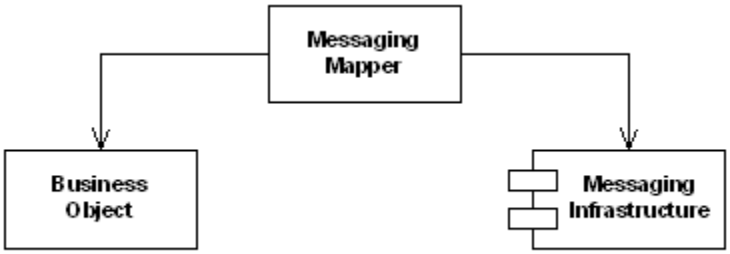


Figure 1: Messaging Mapper EIP

How WSO2 ESB implements the EIP

Messaging Mapper's objective is to serialize domain objects into a format more adaptable to the messaging infrastructure, such as SOAP or JSON.

In WSO2 ESB, the task of a Message Mapper is simulated by Message Builders and Message Formatters. They provide serialization logic to convert a byte stream to a standard data serialization format such as XML/SOAP or JSON. You can implement custom Message Builders/Formatters and plug them into the ESB to map just about any domain object into any sort of messaging infrastructure.

Transactional Client

This section explains, through an example scenario, how the Transactional Client EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Transactional Client
- Example scenario
    - Environment setup
    - ESB configuration
    - Simulating the sample scenario
    - How the implementation works

Introduction to Transactional Client

The Transactional Client EIP controls transactions with the messaging system. It makes the client's session with the messaging system transactional so that the client can specify transaction boundaries. For more information, refer to http://www.eaipatterns.com/TransactionalClient.html.

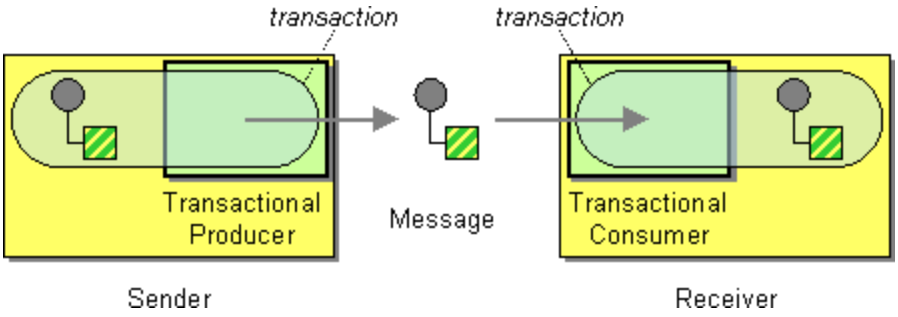


Figure 1: Transactional Client EIP

Example scenario

In many business scenarios, there can be data loss when transferring from one section to another because of crashes and other interruptions. This example scenario demonstrates how messages containing the `MESSAGE_COUNT` value (MESSAGE_COUNT=1) are assumed to be causing a failure in mediation, and how transactions guarantee that data will not be lost.

There are two ways to implement transaction in WSO2 ESB as follows.

- Transaction Mediator
- JMS Transactions

The ESB can control transactional behavior with a JMS queue and by simulating the Transactional Client EAI pattern. For information on Transactions in WSO2 ESB, refer to Transactions in the WSO2 ESB documentation.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

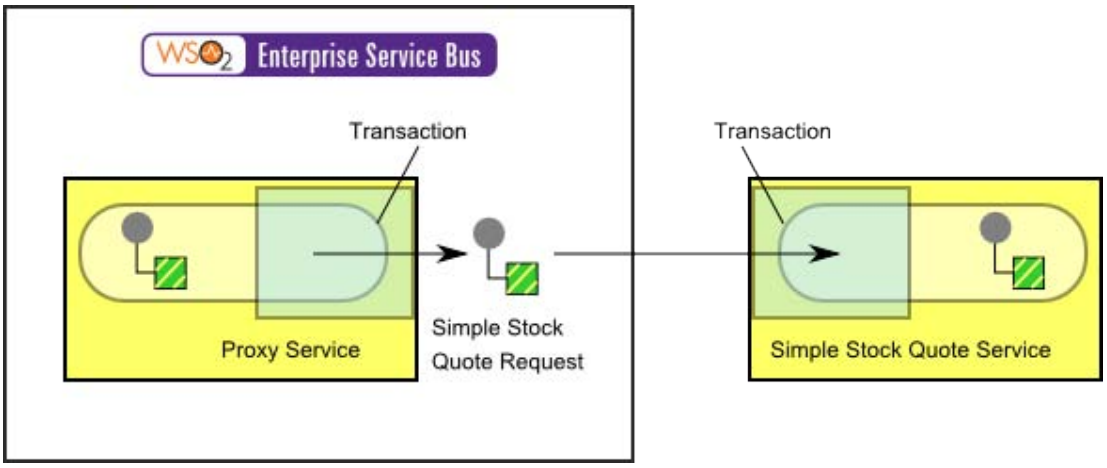


Figure 2: Example Scenario of the Transactional Client EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Transactional Client EIP by comparing their core components.

Transactional Client EIP (Figure 1)	Transactional Client Example Scenario (Figure 2)
Transactional Producer	Proxy service inSequence
Message	Simple Stock Quote Request
Transactional Consumer	Simple Stock Quote Service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Download and install a JMS server. We use ActiveMQ as the JMS provider in this example.
3. In order to enable the JMS transport, edit `<ESB_HOME>/repository/conf/axis2/axis2.xml` as follows.
   - Uncomment the Axis2 transport listener configuration for ActiveMQ as follows:
     `<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">...`
   - Set the `transport.jms.SessionTransacted` parameter to true. After making this update, the `transportReceiver` section in `axis2.xml` should look as follows:

```
<transportReceiver name="jms"
class="org.apache.axis2.transport.jms.JMSListener">
        <parameter name="myQueueConnectionFactory" locked="false">
            <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</pa
rameter>
            <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
            <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
             <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
             <parameter
name="transport.jms.SessionTransacted">true</parameter>
        </parameter>

        <parameter name="default" locked="false">
            <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</pa
rameter>
            <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
            <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
            <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
   <parameter name="transport.jms.SessionTransacted">true</parameter>
        </parameter>
</transportReceiver>
```

   - Uncomment the Axis2 transport sender configuration as follows:
     `<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender"/>`
4. Copy the following ActiveMQ client jar files to the `<ESB_HOME>/repository/components/lib` directory. It allows the ESB to connect to the JMS provider.
   - `activemq-core-5.2.0.jar`
   - `geronimo-j2ee-management_1.0_spec-1.0.jar`
5. You need to add a custom mediator called `MessageCounterMediator`. Download the `MessageCounterM`

ediator file and place it in the <ESB_HOME>/repository/components/lib folder. To learn how to write custom mediators, refer to Writing Custom Mediator Implementations in the WSO2 ESB documentation.

6. Start ActiveMQ (or equivalent JMS Server) and WSO2 ESB.
7. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (https://localhost:9443/carbon). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="StockQuoteProxy" startOnLoad="true" trace="disable">
      <target>
         <inSequence>
   <!-- Setting MESSAGE_COUNT value -->
            <class name="org.wso2.esb.client.MessageCounterMediator"/>
            <log level="full">
               <property name="MESSAGE*ID"
expression="get-property('MESSAGE_COUNT')"/>
            </log>
            <switch source="get-property('MESSAGE_COUNT')">
               <case regex="1">
     <!-- Undesired MESSAGE_COUNT value -->
                  <property name="SET_ROLLBACK_ONLY" value="true" scope="axis2"/>
                  <log level="custom">
                     <property name="Transaction Action" value="Rollbacked"/>
                  </log>
               </case>
               <default>
                  <log level="custom">
                     <property name="Transaction Action" value="Committed"/>
                  </log>
                  <send>
                     <endpoint name="proxy_endpoint">
                        <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
                     </endpoint>
                  </send>
               </default>
            </switch>
            <property name="OUT_ONLY" value="true"/>
         </inSequence>
      </target>
      <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
      <parameter name="transport.jms.ContentType">
         <rules>
            <jmsProperty>contentType</jmsProperty>
            <default>application/xml</default>
         </rules>
      </parameter>
   </proxy>
   <sequence name="main">
      <log/>
      <drop/>
   </sequence>
</definitions>
```

Simulating the sample scenario

Use the ESB's default `jmsclient` to send messages to the JMS queue in ActiveMQ as follows:

```
ant jmsclient -Djms_type=pox -Djms_dest=dynamicQueues/StockQuoteProxy
-Djms_payload=WSO2
```

Note in the ESB console that the first attempt will roll back and the second attempt will be committed.

```
[2012-10-26 22:55:32,520]  INFO - LogMediator To: , MessageID:
ID:buddhima-pc-59457-1351272332247-1:1:1:1:1, Direction: request, MESSAGE*ID = 1,
Envelope: <?xml version='1.0' encoding='utf-8'?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><m:placeOrde
r xmlns:m="http://services.samples">
    <m:order>
        <m:price>87.63805687450525</m:price>
        <m:quantity>18427</m:quantity>
        <m:symbol>WSO2</m:symbol>
    </m:order>
</m:placeOrder></soapenv:Body></soapenv:Envelope>
[2012-10-26 22:55:32,521]  INFO - LogMediator Transaction Action = Rollbacked

[2012-10-26 22:55:33,541]  INFO - LogMediator To: , MessageID:
ID:buddhima-pc-59457-1351272332247-1:1:1:1:1, Direction: request, MESSAGE*ID = 2,
Envelope: <?xml version='1.0' encoding='utf-8'?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"><soapenv:Body><m:placeOrde
r xmlns:m="http://services.samples">
    <m:order>
        <m:price>87.63805687450525</m:price>
        <m:quantity>18427</m:quantity>
        <m:symbol>WSO2</m:symbol>
    </m:order>
</m:placeOrder></soapenv:Body></soapenv:Envelope>
[2012-10-26 22:55:33,541]  INFO - LogMediator Transaction Action = Committed
```

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **class** [line 6 in ESB config] - A Custom mediator called `MessageCountMediator` is loaded. This mediator keeps track of the number of messages that pass through the sequence it calls by updating the value of a variable named `MESSAGE_COUNT`.
- **switch** [line 10 in ESB config] - The Switch mediator checks the value of `MESSAGE_COUNT`.
- **case** [line 11 in ESB config] - If the value of `MESSAGE_COUNT` is 1, the transaction is considered to have failed, and no message will be put on the send channel.
- **default** [line 18 in ESB config] - The default action of the switch case flow control. This occurs when the `MESSAGE_COUNT` is anything but 1. The message is put on the send channel, and the transaction is considered a success.

Polling Consumer

This section explains, through an example scenario, how the Polling Consumer EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Polling Consumer

The Polling Consumer EIP allows the ESB to explicitly make a call when the application wants to receive a message. For more information, refer to http://www.eaipatterns.com/PollingConsumer.html.

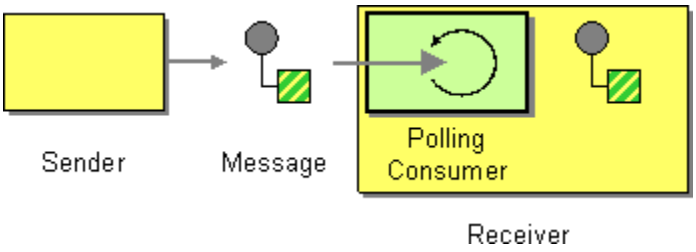


Figure 1: Polling Consumer EIP

Example scenario

This example scenario demonstrates the WSO2 ESB JMS proxy, which is a polling transport that is used to connect to various JMS providers. After configuring the JMS proxy, it will listen on a queue on the JMS server. Next, we send a message to the queue, and the JMS proxy in the ESB will pick up the message when it is ready for consumption.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

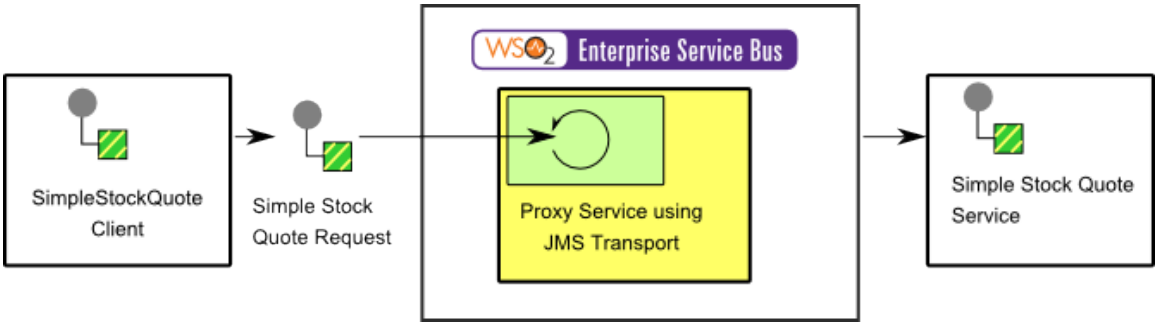


Figure 2: Example Scenario of the Polling Consumer EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Polling Consumer EIP by comparing their core components.

Polling Consumer EIP (Figure 1)	Polling Consumer Example Scenario (Figure 2)
Sender	Simple Stock Quote Client
Message	Simple Stock Quote Request
Polling Consumer	Proxy Service using JMS Transport
Receiver	Simple Stock Quote Service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Download and install a JMS server. We use ActiveMQ as the JMS provider in this example.
3. Make the following edits to the `<ESB_HOME>/repository/conf/axis2/axis2.xml` file.

- To enable the JMS transport, uncomment the Axis2 transport listener configuration for ActiveMQ as follows:
  `<transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">...`

- Set the `transport.jms.SessionTransacted` parameter to true. After making this update, the `transportReceiver` section of `axis2.xml` should appear as follows:

```
<transportReceiver name="jms"
class="org.apache.axis2.transport.jms.JMSListener">
        <parameter name="myQueueConnectionFactory" locked="false">
            <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</pa
rameter>
            <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
            <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
             <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
             <parameter
name="transport.jms.SessionTransacted">true</parameter>
        </parameter>

        <parameter name="default" locked="false">
            <parameter name="java.naming.factory.initial"
locked="false">org.apache.activemq.jndi.ActiveMQInitialContextFactory</pa
rameter>
            <parameter name="java.naming.provider.url"
locked="false">tcp://localhost:61616</parameter>
            <parameter name="transport.jms.ConnectionFactoryJNDIName"
locked="false">QueueConnectionFactory</parameter>
            <parameter name="transport.jms.ConnectionFactoryType"
locked="false">queue</parameter>
   <parameter name="transport.jms.SessionTransacted">true</parameter>
        </parameter>
</transportReceiver>
```

- Uncomment the Axis2 transport sender configuration as follows:
  `<transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender"/>`

4. Copy the following ActiveMQ client JAR files to the `<ESB_HOME>/repository/components/lib` directory to allow the ESB to connect to the JMS provider.
   - `activemq-core-5.2.0.jar`
   - `geronimo-j2ee-management_1.0_spec-1.0.jar`

5. Add a custom mediator called `MessageCounterMediator`. Download the `MessageCounterMediator` file , and save it in the `<ESB_HOME>/repository/components/lib` folder. To learn how to write custom mediators, refer to Writing Custom Mediator Implementations in the WSO2 ESB documentation.
6. Start ActiveMQ (or equivalent JMS Server) and WSO2 ESB.

7. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Log into the ESB's management console UI (`https://localhost:9443/carbon`), and navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <proxy name="StockQuoteProxy" transports="jms">
        <target>
            <inSequence>
                <property action="set" name="OUT_ONLY" value="true"/>
            </inSequence>
            <endpoint>
                <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
            </endpoint>
            <outSequence>
                <send/>
            </outSequence>
        </target>
        <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
        <parameter name="transport.jms.ContentType">
            <rules>
                <jmsProperty>contentType</jmsProperty>
                <default>application/xml</default>
            </rules>
        </parameter>
    </proxy>
</definitions>
```

Simulating the sample scenario

1. Send a stock quote request on JMS, as follows:
   ```
   ant jmsclient -Djms_type=pox -Djms_dest=dynamicQueues/StockQuoteProxy
   -Djms_payload=MSFT
   ```

2. Note a message on the console running the sample Axis2 server saying that the server has accepted an order. For example:
   ```
   Accepted order #1 for : 17718 stocks of MSFT at $ 79.83113379282025
   ```

   Also note that there is one message queued and one de-queued in the queue created in the ActiveMQ web console at `http://localhost:8161/admin/queues.jsp`.
3. Next, stop the ESB server, and send a stock quote request again. In the ActiveMQ web console, you can see that there are two messages queued and only one message de-queued.
4. Start the ESB server again, and view the console running the sample Axis2 server. You will see a message indicating that the server has accepted the second message.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB

configuration shown above.

- **proxy** [line 2 in ESB config] - A proxy service with a JMS transport.
- **parameter** [line 15 in ESB config] - Sets JMS transport parameter `contentType` to application/xml.

Event-Driven Consumer

This section explains, through an example scenario, how the Event-Driven Consumer EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Event-Driven Consumer

The Event-Driven Consumer EIP allows an application to automatically consume messages as they become available. For more information, refer to http://www.eaipatterns.com/EventDrivenConsumer.html.

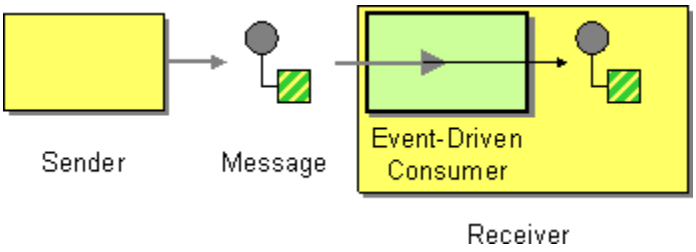


Figure 1: Event-Driven Consumer EIP

Example scenario

This EIP is also referred to as an asynchronous receiver. This example scenario demonstrates how an event will be triggered based on the availability of the receiver and a message will be consumed by the receiver.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

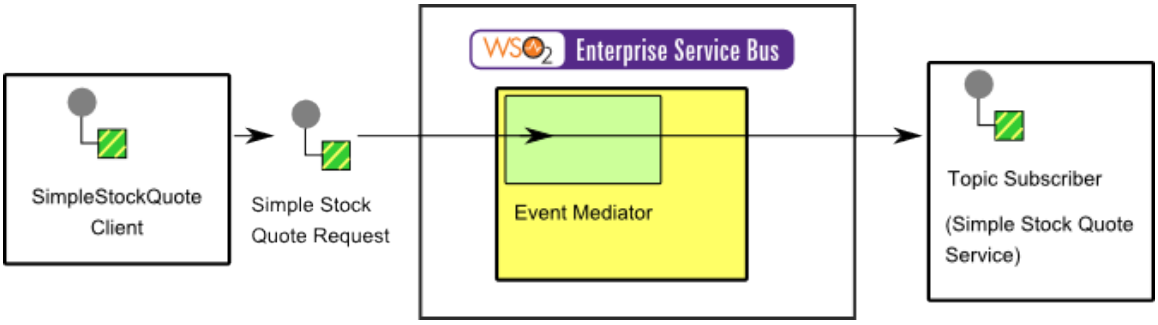


Figure 2: Example Scenario of the Event-Driven Consumer EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Event-Driven Consumer EIP by comparing their core components.

Event-Driven Consumer EIP (Figure 1)	Event-Driven Consumer Example Scenario (Figure 2)
Sender	Simple Stock Quote Client
Message	Simple Stock Quote Request
Event Driven Consumer	Event Mediator
Receiver	Topic Subscriber, Simple Stock Quote Service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.
3. Follow the steps below to create an event.
    - Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`) .
    - Select the **Topics** menu from the **Main** menu and then select the **Add** sub menu.
    - Enter `EventConsumerTopic` as the name for the topic, and then click **Add Topic**.
    - Click the newly created topic **EventConsumerTopic** in the topic browser tree and click **Subscribe** to create a static subscription.
    - Enter the value `http://localhost:9000/services/SimpleStockQuoteService` in the **Event Sink URL** field and click **Subscribe**.

ESB configuration

In the ESB's Management Console, navigate to **Main** menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <log/>
      <event topic="EventConsumerTopic"/>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send a request using the Stock Quote client to the ESB as follows. For information about the Stock Quote client, refer to the Sample Clients section in the WSO2 ESB documentation.

```
ant stockquote -Dtrpurl=http://localhost:8280 -Dsymbol=Foo
```

After sending the request, note that a message accepting the request will show on the Stock Quote service's console. This is triggered as an event when the message is published into the topic `EventConsumerTopic` created earlier. All subscribers will receive the topic.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below are mapped with the ESB configuration shown above.

- **event** [line 13 in ESB config] - Allows you to define a set of subscribers that will receive messages when the topic subscribed to receives a message. Also, see Eventing.

Competing Consumers

This section explains, through an example scenario, how the Competing Consumers EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Competing Consumers

The Competing Consumers EIP relates to multiple consumers that compete with each other to receive a request from a given Point-to-Point Channel. In this pattern, requests are handled and delegated similar to how messages are handled in the Point-to-Point channel using the round-robin algorithm. For more information, refer to http://www.eaipatterns.com/CompetingConsumers.html.

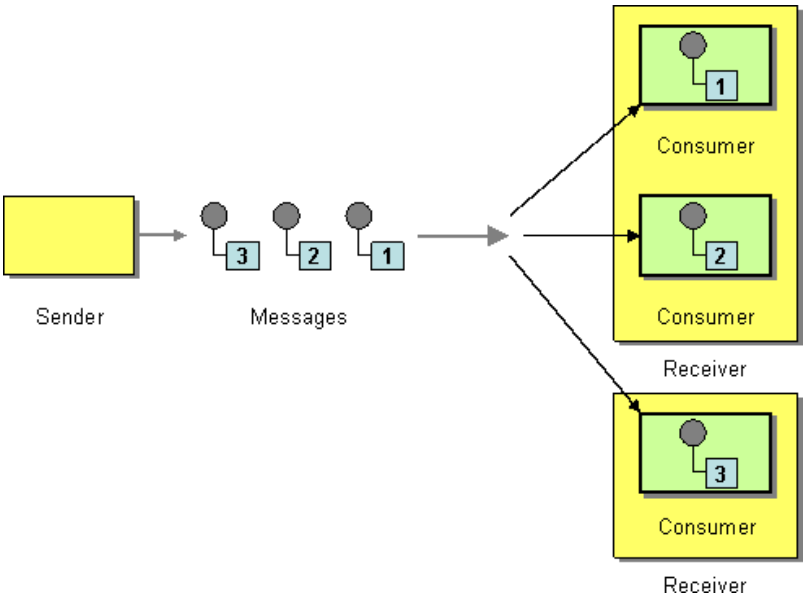


Figure 1: Competing Consumers EIP

Example scenario

In this example scenario, each Axis2 server instance acts as a consumer waiting for a specific message. When the client sends a message to WSO2 ESB, it diverts the request based on the round-robin algorithm among the consumers. This way, the consumers receive a request message adhering to the conditions of the algorithm.

An alternative implementation to the Competing Consumers EIP is to use a Content-Based Router and route messages to different receivers based on the content of a message.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

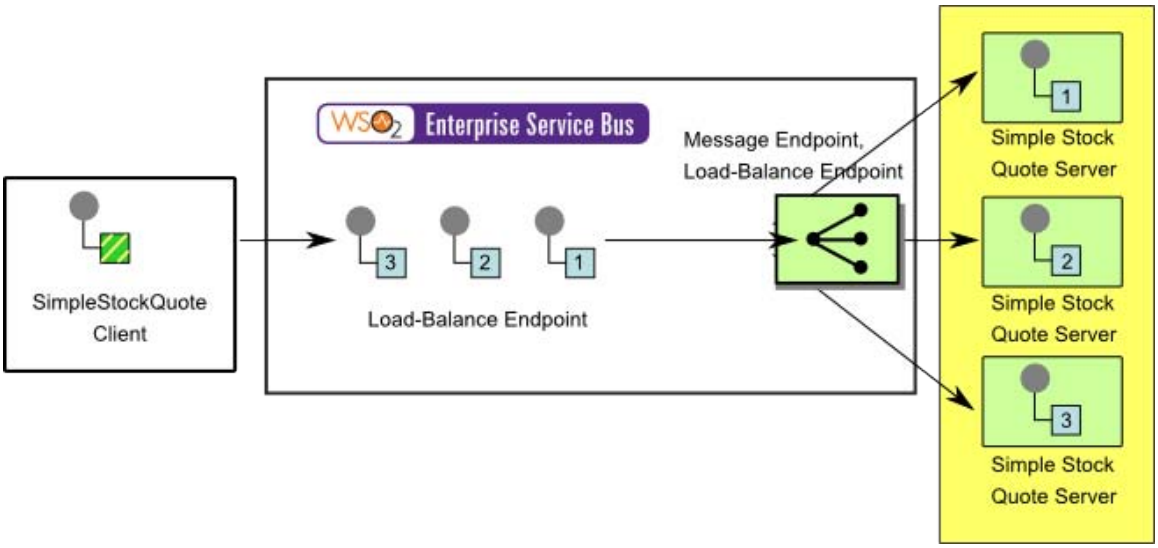


Figure 2: Example Scenario of the Competing Consumers EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Competing Consumers EIP by comparing their core components.

Competing Consumers EIP (Function 1)	Competing Consumers Example Scenario (Function 2)
Sender	Simple Stock Quote Client
Messages	Simple Stock Quote Requests with Load-Balance Endpoint
Consumer/Receiver	Simple Stock Quote Server Instances

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start three sample Axis2 server instances on ports 9000, 9001, and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in>
         <send>
            <endpoint>
               <loadbalance
algorithm="org.apache.synapse.endpoints.algorithms.RoundRobin">
                  <endpoint>
                     <address
uri="http://localhost:9000/services/SimpleStockQuoteService/"/>
                  </endpoint>
                  <endpoint>
                     <address
uri="http://localhost:9001/services/SimpleStockQuoteService/"/>
                  </endpoint>
                  <endpoint>
                     <address
uri="http://localhost:9002/services/SimpleStockQuoteService/"/>
                  </endpoint>
               </loadbalance>
            </endpoint>
         </send>
      </in>
      <out>
         <send/>
      </out>
   </sequence>
</definitions>
```

Simulating the sample scenario

Repeatedly send several requests to the ESB using the `stockquote` client as follows:

```
ant stockquote -Dtrpurl=http://localhost:8280/ -Dsymbol=Foo
```

Following is the request sent by the `stockquote` client in this example.

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header />
   <soapenv:Body>
        <ser:getQuote>
         <ser:request>
             <ser:symbol>Foo</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

Note in each Axis2 server console that the requests are distributed among several servers. Each server is acting as a competing consumer.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **endpoint** [line 13 in ESB config] - Defines the endpoint where the request should be sent.
- **loadbalance** [line 14 in ESB config] - Defines a set of endpoints where incoming requests are distributed using a particular algorithm. In this example, the algorithm distributes messages in a round-robin manner.

Message Dispatcher

This section explains, through an example scenario, how the Message Dispatcher EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Message Dispatcher

The Message Dispatcher EIP consumes messages from a single channel and distributes them among performers. It allows multiple consumers on a single channel to coordinate their message processing. For more information, refer to http://www.eaipatterns.com/MessageDispatcher.html.

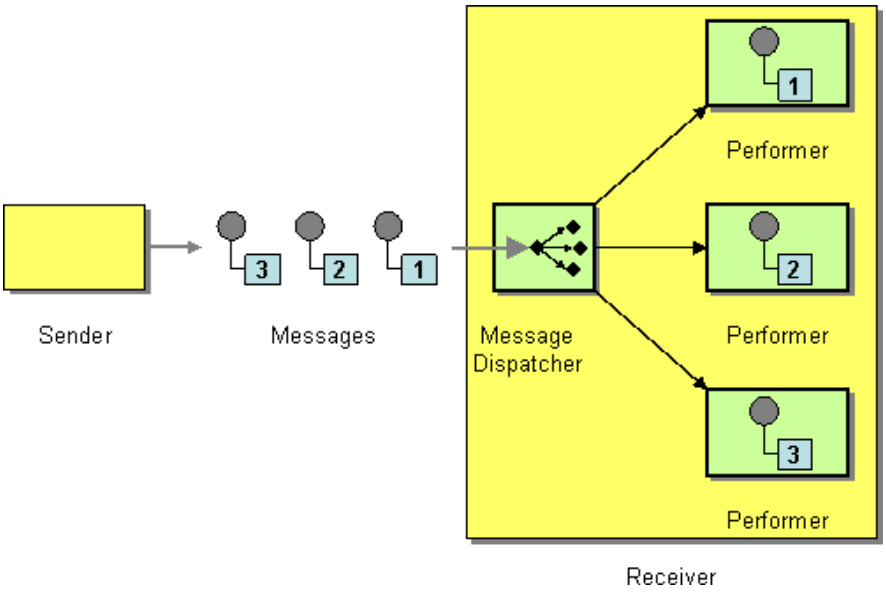


Figure 1: Message Dispatcher EIP

Example scenario

This example scenario demonstrates how to distribute messages among performers using the weighted load balance mediator. We have several Axis2 server instances, each considered to be a performer.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

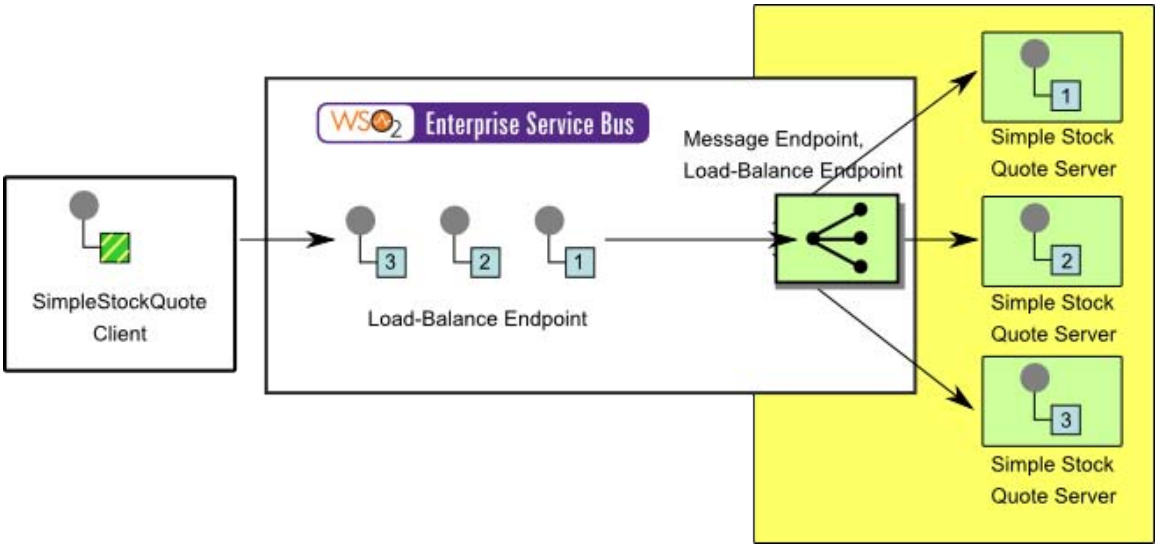


Figure 2: Example Scenario of the Message Dispatcher EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Message Dispatcher EIP by comparing their core components.

Message Dispatcher EIP (Figure 1)	Message Dispatcher Example Scenario (Figure 2)
Sender	Simple Stock Quote Client
Messages	Simple Stock Quote Requests

Message Dispatcher	Message Endpoint, Load-Balance Endpoint
Performers	Simple Stock Quote Server Instances

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start three sample Axis2 server instances on ports 9000, 9001, and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in>
         <send>
            <endpoint>
               <loadbalance
algorithm="org.apache.synapse.endpoints.algorithms.WeightedRoundRobin">
                  <endpoint>
                     <address
uri="http://localhost:9000/services/SimpleStockQuoteService/"/>
                     <property name="loadbalance.weight" value="1"/>
                  </endpoint>
                  <endpoint>
                     <address
uri="http://localhost:9001/services/SimpleStockQuoteService/"/>
                     <property name="loadbalance.weight" value="2"/>
                  </endpoint>
                  <endpoint>
                     <address
uri="http://localhost:9002/services/SimpleStockQuoteService/"/>
                     <property name="loadbalance.weight" value="3"/>
                  </endpoint>
               </loadbalance>
            </endpoint>
         </send>
      </in>
      <out>
         <send/>
      </out>
   </sequence>
</definitions>
```

Simulating the sample scenario

Repeatedly send several requests to the ESB using the Stock Quote client as follows:

```
ant stockquote -Dtrpurl=http://localhost:8280/ -Dsymbol=Foo
```

Following is the request sent by the Stock Quote client in this example:

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header />
   <soapenv:Body>
        <ser:getQuote>
         <ser:request>
             <ser:symbol>Foo</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

Note in each Axis2 server console that the requests are distributed among several servers in a weighted manner. Se rvers running on port 9000, 9001, and 9002 receive the request in that order until the process starts over again in a round-robin manner.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **endpoint** [line 13 in ESB config] - Defines the endpoint where the request should be sent.
- **loadbalance** [line 14 in ESB config] - Defines a set of endpoints where incoming requests are distributed using a particular algorithm. In this example, the algorithm distributes messages in a weighted round-robin manner.

Selective Consumer

This section explains, through an example scenario, how the Selective Consumer EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Selective Consumer
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Selective Consumer

The Selective Consumer EIP allows a message consumer to select which messages it will receive. It filters the messages delivered by its channel so that it only receives the ones that match its criteria. For more information, refer to http://www.eaipatterns.com/MessageSelector.html.

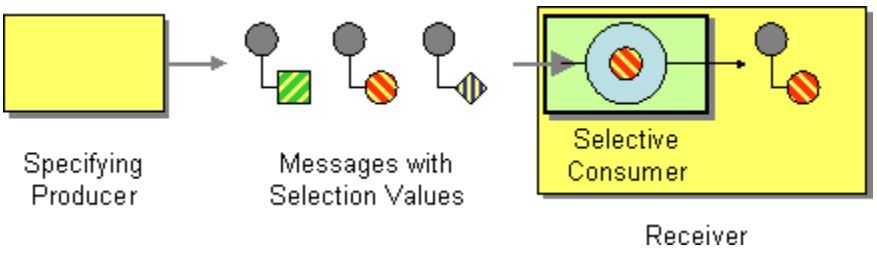


Figure 1: Selective Consumer EIP

Example scenario

This example scenario demonstrates how a specific receiver only processes messages that are pre-filtered based on certain criteria. We have an Axis2 server as the consumer. The consumer criteria is specified through an XML schema validation, which is stored as a local entry in the registry. We use the Validate mediator to check whether the messages that are sent to the ESB match the criteria of the schema. The Axis2 server can consume the message only if the message meets the validation criteria.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

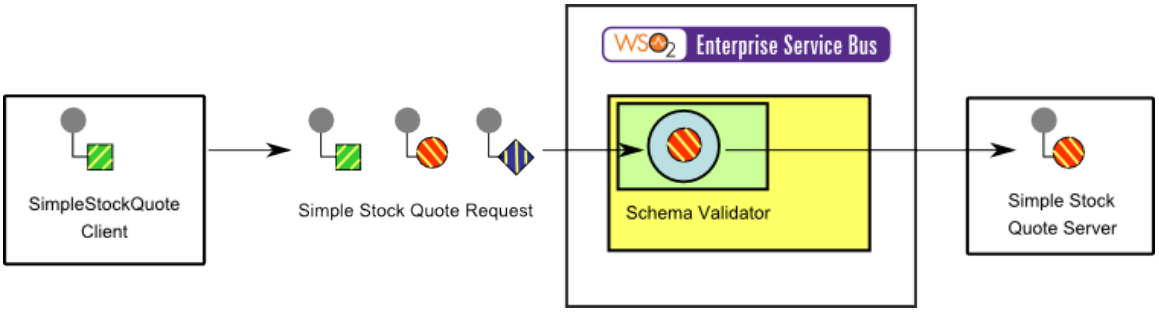


Figure 2: Example Scenario of the Selective Consumer EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Selective Consumer EIP by comparing their core components.

Selective Consumer EIP (Figure 1)	Selective Consumer Example Scenario (Figure 2)
Specifying Producer	Simple Stock Quote Client
Messages with Selection Values	Simple Stock Quote Request
Selective Consumer	Schema Validator (Validate Mediator)
Receiver	Simple Stock Quote Server

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.


```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <localEntry key="selective_criteria">
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 xmlns="http://www.apache-synapse.org/test"
                 elementFormDefault="qualified"
                 attributeFormDefault="unqualified"
                 targetNamespace="http://services.samples">
         <xs:element name="getQuote">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="request">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="stockvalue" type="xs:string"/>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:schema>
   </localEntry>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in>
         <validate>
            <schema key="selective_criteria"/>
            <on-fail>
               <makefault>
                  <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
value="tns:Receiver"/>
                  <reason value="Invalid custom quote request"/>
               </makefault>
               <property name="RESPONSE" value="true"/>
               <header name="To" expression="get-property('ReplyTo')"/>
               <drop/>
            </on-fail>
         </validate>
         <send>
            <endpoint>
               <address
uri="http://localhost:9000/services/SimpleStockQuoteService?wsdl"/>
            </endpoint>
         </send>
      </in>
      <out>
         <send/>
      </out>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send the following request using a SOAP client like SoapUI.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <ser:symbol>IBM</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

Note that the stock quote request is not processed. Send the following message to the ESB server.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <ser:stockvalue></ser:stockvalue>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

The consumer has specified the criteria of using a schema validation. Only the messages that meet this criteria will be consumed.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **localEntry** [line 2 in ESB config] - A local registry entry with key `selective_criteria` is used to define the XML schema used for validation inside the main sequence.
- **validate** [line 33] in ESB config] - This mediator is used to define the portion of a message used for validation. In this example, no source attribute is specified using an XPath expression, so the ESB performs the validation on the first child element of the SOAP body.
- **scehma** [line 34 in ESB config] - Defines which schema to use for validation. In this example, the local registry entry definition in line 2 is used.
- **on-fail** [line 35 in ESB config] - Defines the action to take on failure of a validation. In this example, a fault is created and the message is dropped.

Durable Subscriber

This section explains, through an example scenario, how the Durable Subscriber EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Durable Subscriber

The Durable Subscriber EIP avoids missing messages while it's not listening for them. It makes the messaging system save messages published while the subscriber is disconnected. This pattern is similar to the Publish-Subscribe EIP, which temporarily stores a message if a subscriber is offline at the time a message is published, and sends the message when it gets online again. For more information, refer to http://www.eaipatterns.com/DurableSubscription.html.

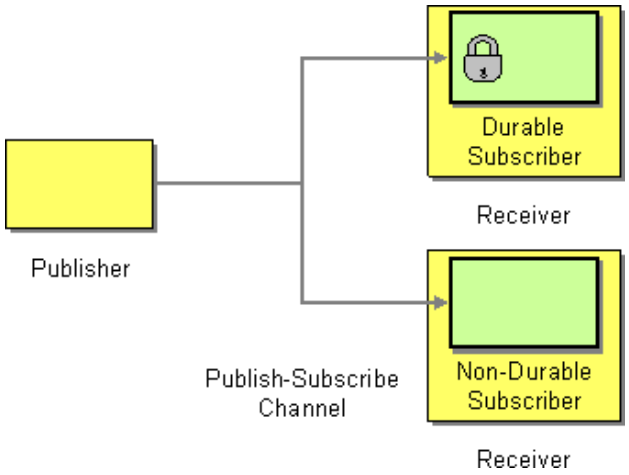


Figure 1: Durable Subscriber EIP

Example scenario

This example scenario demonstrates how a message is duplicated and routed to the subscribers using the Clone mediator when the publisher sends a message. We have two Axis2 servers as the subscribers. If only one subscriber is online at the time a message is sent, instead of discarding the message, it will be stored in a message store. The message forwarding processor will attempt to send the message in the store until the subscriber comes online. When the subscriber comes online, the message will be successfully delivered.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

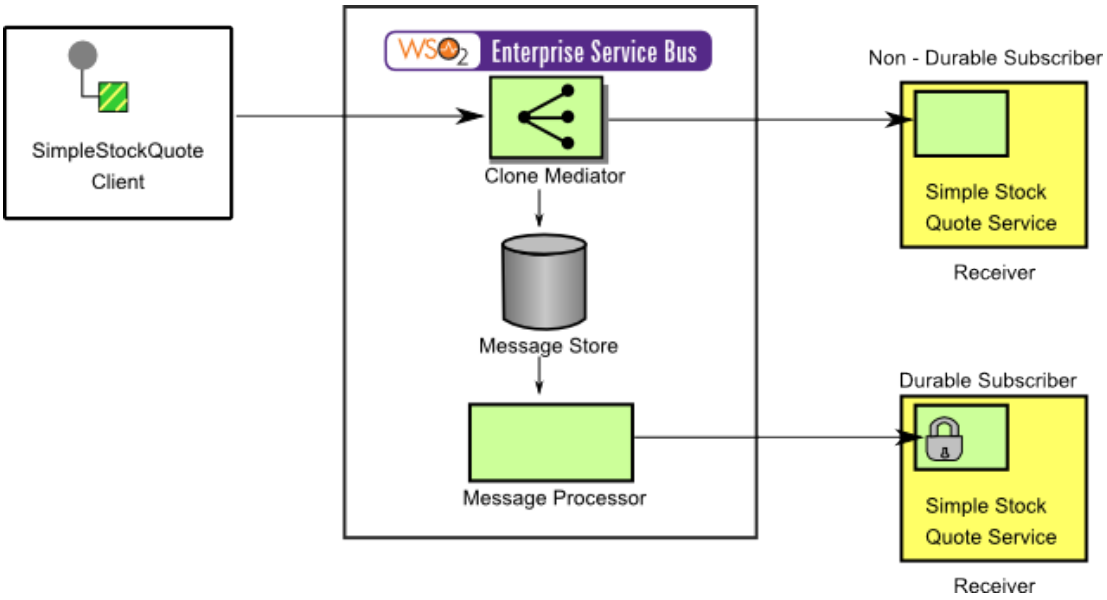


Figure 2: Example Scenario of the Durable Subscriber EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Durable Subscriber EIP by comparing their core components.

Durable Subscriber EIP (Figure 1)	Durable Subscriber Example Scenario (Figure 2)
Publisher	Simple Stock Quote Client
Publish Subscribe Channel	Clone Mediator, Message Store, Message Processor
Durable Consumer	Simple Stock Quote Service
Non Durable Consumer	Simple Stock Quote Service

Environment setup

1. Download and install the WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start three sample Axis2 server instances on ports 9000 and 9001. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- Durable Subscriber Proxy-->
<definitions xmlns="http://ws.apache.org/ns/synapse">
     <registry provider="org.wso2.carbon.mediation.registry.WSO2Registry">
        <parameter name="cachableDuration">15000</parameter>
    </registry>
   <proxy name="PublishProxy" startOnLoad="true" transports="http">
        <target>
        <inSequence>
            <clone>
```

```
                <target sequence="DurableSubscriber" />
              <target sequence="NonDurableSubscriber" />
            </clone>
        </inSequence>
        <outSequence>
            <drop />
        </outSequence>
    </target>
    </proxy>

    <!-- Error Sequences -->
    <sequence name = "sub1_fails" >
      <property name="target.endpoint" value="DurableSubscriberEndpoint" />
       <store messageStore="pending_subscriptions" />
    </sequence>
    <sequence name = "sub2_fails" >
         <drop/>
    </sequence>
    <!-- Subscription List-->
    <sequence name="DurableSubscriber" onError="sub1_fails">
        <in>
        <send>
            <endpoint name="Subscriber 1">
                    <address
uri="http://localhost:9000/services/SimpleStockQuoteService/"/>
                </endpoint>
        </send>
        </in>
   </sequence>
   <sequence name="NonDurableSubscriber" onError="sub2_fails">
        <in>
        <send>
                <endpoint name="Subscriber 2">
                      <address
uri="http://localhost:9001/services/SimpleStockQuoteService/"/>
                </endpoint>
        </send>
         </in>
    </sequence>
    <!-- Re Direction End Points -->
    <endpoint name="DurableSubscriberEndpoint">
        <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
    </endpoint>

    <!-- Message Store And Process -->
    <messageStore name="pending_subscriptions"/>

    <messageProcessor
class="org.apache.synapse.message.processors.forward.ScheduledMessageForwardingProce
ssor" name="send_pending_message"
    messageStore="pending_subscriptions">
        <parameter name="interval">1000</parameter>
```

```
        <parameter name="max.deliver.attempts">50</parameter>
    </messageProcessor>
</definitions>
```

Simulating the sample scenario

Use a SOAP client like SoapUI to forward the following request to the PublishProxy service.

```
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <ser:symbol>Foo</ser:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

Note that both Axis2 servers receive the request. Next, stop the Axis2 server running on port 9000 (the Durable Subscriber) and resend the request again. Note that the Durable Subscriber will not receive the request. Start the Axis2 server on port 9000. Note that the previously undelivered message will be delivered. We can do the same for the Axis2 server running on port 9001 (the Non Durable Subscriber). In that case, when the server is back on, any previously undelivered messages will not be received.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **Proxy Service** [line 6 in ESB config] - A proxy service takes an incoming Stock Quote client request and clones the request by forwarding one copy each to two target sequences, `DurableSubscriber` and `NonDurableSubscriber`.
- **Sequence** [line 29 in ESB config] - The `DurableSubscriber` sequence forwards the message to the Durable Endpoint. This endpoint has the `onError` attribute set to the `sub1_fails` sequence (line 21 in ESB config), which will store the message in case of a failure.
- **Sequence** [line 38 in ESB config] - The `NonDurableSubscriber` sequence works the same as the sequence described above. The only difference is that on failure, the `sub2_fails` sequence (line 25 in ESB config) is called, which simply drops the message.
- **Sequence** [line 21 in ESB config] - This sequence sets the `target.endpoint` property for the `DurableEndpointSubscriber` endpoint (defined on line 48 in ESB config), and uses a Store Mediator to define the message store used to save the message. In this example, it is the store with key `pending_subscription`.

- **messageStore** [line 53 in ESB config] - Defines a new message store with the name `pending_subscriptions`.
- **messageProcessor** [line 55 in ESB config] - The `messageProcessor` is used to define the type of processing done to a particular `messageStore`, which in this example is `pending_subscription`. This example defines a `messageProcessor` that uses a `ScheduledMessageForwardingProcessor`, which

retries sending the messages every second with a maximum number of delivery attempts set to 50.

Idempotent Receiver

This section explains, through an example scenario, how the Idempotent Receiver EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Idempotent Receiver

The Idempotent Receiver EIP defines how a message receiver getting multiple copies of the same message can safely ignore duplicates. For more information, refer to http://www.eaipatterns.com/IdempotentReceiver.html.

Example scenario

Idempotency is a Message Consumer Pattern where a receiver can identify duplicate messages sent by a sender due to a loss of acknowledgement or some other transport error. In this example scenario, the Resequencing Message Processor selects the message using the sequence number and hands it over to do the rest of the processing. Only one message with a particular sequence number will be further processed by the message store.

> The Resequncing Message Processor is available from WSO2 ESB version 4.6.1 onwards.

Environment setup

1. Download and install the WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server instance. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<!-- The example shows WSO2 ESB acting as an Idempotent Receiver -->
<definitions xmlns="http://ws.apache.org/ns/synapse">
    <proxy name="IdempotencyReceivingProxy">
        <target>
            <inSequence>
          <log level="full"/>
    <!-- Store all messages in an in-memory message store -->
    <store messageStore="MyStore"/>
         </inSequence>
            <outSequence>
                <send/>
            </outSequence>
        </target>
    </proxy>
 <!-- Further mediation of messages are done in this sequence -->
 <sequence name="next_seq">
  <send>
   <endpoint>
    <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
   </endpoint>
  </send>
 </sequence>
 <messageStore name="MyStore"/>
 <!-- Resequencing Processor takes the next sequence number and hand over to
"next_seq" and preserve Idempotency -->
 <messageProcessor
   class="org.apache.synapse.message.processors.resequence.ResequencingProcessor"
   name="ResequencingProcessor" messageStore="MyStore">
  <parameter name="interval">2000</parameter>
  <!-- Takes the sequence number using Xpath -->
  <parameter name="seqNumXpath" xmlns:m0="http://services.samples"
expression="substring-after(//m0:placeOrder/m0:order/m0:symbol,'-')"/>
  <parameter name="nextEsbSequence">next_seq</parameter>
        <parameter name="deleteDuplicateMessages">true</parameter>
 </messageProcessor>
</definitions>
```

Simulating the sample scenario

Send requests to WSO2 ESB as follows.

```
ant stockquote -Dtrpurl=http://localhost:8280/ -Dmode=placeorder -Dsymbol=FOO-1
ant stockquote -Dtrpurl=http://localhost:8280/ -Dmode=placeorder -Dsymbol=FOO-2
ant stockquote -Dtrpurl=http://localhost:8280/ -Dmode=placeorder -Dsymbol=FOO-2
ant stockquote -Dtrpurl=http://localhost:8280/ -Dmode=placeorder -Dsymbol=FOO-2
ant stockquote -Dtrpurl=http://localhost:8280/ -Dmode=placeorder -Dsymbol=FOO-3
```

Note that the Axis2 server processes each request (FOO-1, FOO-2, and FOO-3) only once.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB

configuration shown above.

- **Proxy Service** [line 3 in ESB config] - The proxy service with key `IdempotencyReceivingProxy` has an `inSequence` that places incoming messages into the message store with key `MyStore` (line 8 in ESB config).
- **Sequence** [line 16 in ESB config] - The sequence with key `next_seq` defines the endpoint where messages are sent when the sequence is called by its key elsewhere in the configuration.
- **messageStore** [line 23 in ESB config] - Defines a new message store with the name `MyStore.`
- **messageProcessor** [line 25 in ESB config] - The `messageProcessor` turns the message store `MyStore` in to a re-sequencer. During the interval period, messages coming in are stored and reordered based on the sequence number, which is the value of the `seqNumXpath` parameter's XPath expression. They are then delivered in order. The property `deleteDuplicateMessages` is set to true, which deletes any messages with the same sequence number.

Service Activator

This section explains, through an example scenario, how the Service Activator EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Service Activator
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Service Activator

The Service Activator EIP allows an application to design a service to be invoked both via various messaging technologies and non-messaging techniques. Service Activator interfaces methods and services in the back-end service layer so that the back-end services can be filtered and displayed to the client. For more information, refer to http://www.eaipatterns.com/MessagingAdapter.html.

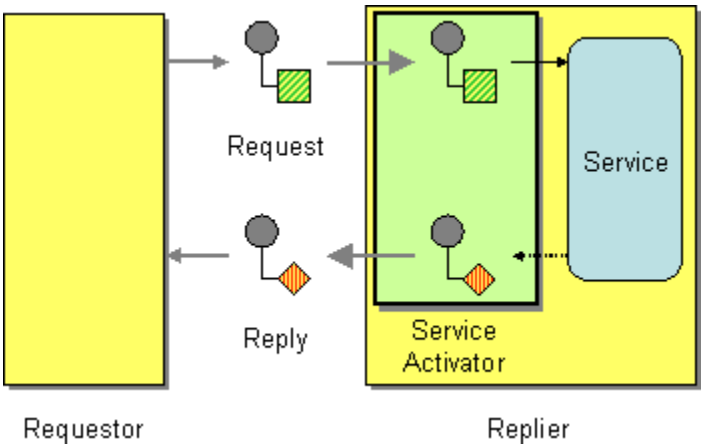


Figure 1: Service Activator EIP

Example scenario

This example scenario demonstrates how WSO2 ESB can be used to activate only a specific number of services on a back-end Axis2 server. Using the publishWSDL property, the service WSDL file is modified to filter out only a

specific number of services. The ability of the ESB to create proxy services allows the client to invoke the ESB proxy instead of invoking the service directly on the Axis2 server.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

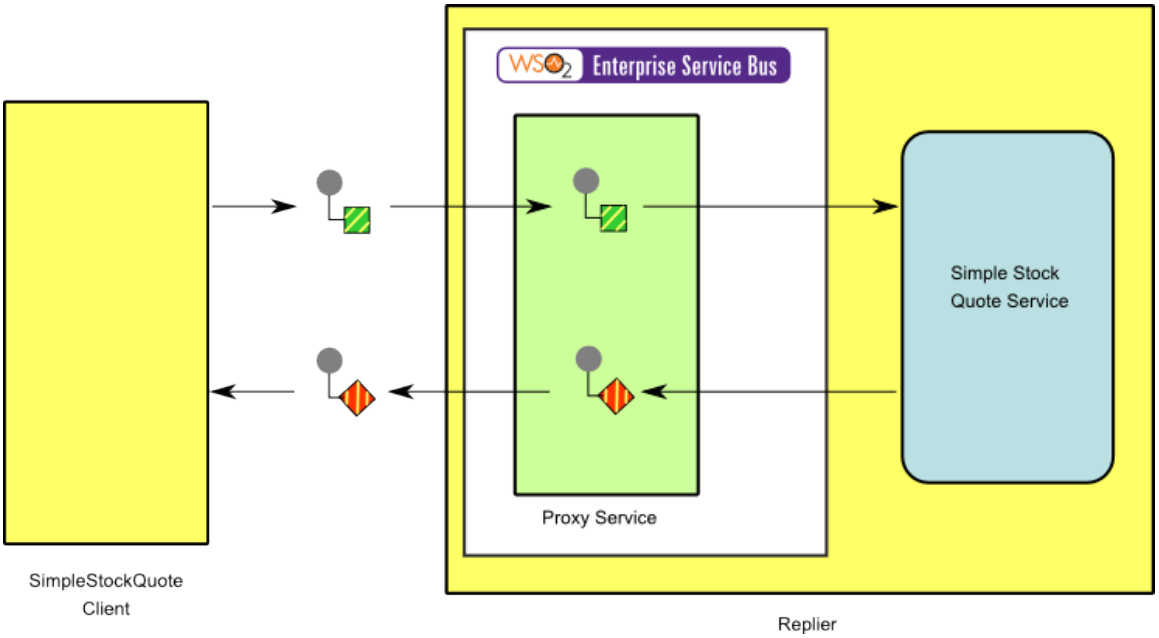


Figure 2: Example Scenario of the Service Activator EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Service Activator EIP by comparing their core components.

Service Activator EIP (Figure 1)	Service Activator Example Scenario (Figure 2)
Requestor	Simple Stock Quote Client
Service Activator	Proxy Service
Replier	Simple Stock Quote Service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances in ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.


```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="ServiceActivatorProxy" startOnLoad="true">
      <target>
         <endpoint>
            <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
         </endpoint>
         <outSequence>
            <send/>
         </outSequence>
      </target>
      <publishWSDL
uri="file:repository/samples/resources/proxy/sample_proxy_1.wsdl"/>
   </proxy>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in/>
      <out>
         <send/>
      </out>
   </sequence>
</definitions>
```

Simulating the sample scenario

The back-end service `StockQuoteService` offers the following services:

- getFullQuote
- getMarketActivity
- getQuote
- getSimpleQuote
- placeOrder

Only some of the back-end features will be published through the Service Activator Proxy. Browse `http://localhost:8280/services` to view the services offered through the `ServiceActivatorProxy`, and note that services `getMarketActivity` and `getSimpleQuote` are not available. Others are available and active.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

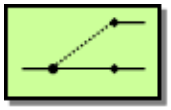
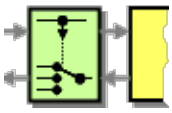
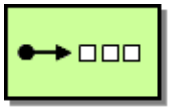
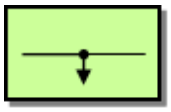
- **Proxy** [line 2 in ESB config] - The proxy service creates a virtual service between the real back-end service and a requestor.
- **publishWSDL** [line 11 in ESB config] - By default, a proxy service defines a one-to-one mapping of the back-end service interface in the form of a WSDL file that requestors can use to connect to the proxy service. By using the publishWSDL mediator, the proxy service can publish a custom interface. In this example, the

publishWSDL mediator is used to provide access only to a subset of all the service methods available to the back-end service.

System Management

System Management patterns aim to monitor the amount of sent messages and their processing times without analyzing the message data, except for some fields in the message header in selected patterns. This is in contrast to Business Activity Monitoring, which directly analyzes the payload data contained in a message.

This chapter introduces various system management patterns and how each can be simulated using WSO2 ESB.

	Channel Purger	Removes unwanted messages, which can disturb tests or running systems, from a channel.
	Control Bus	Administers a messaging system that is distributed across multiple platforms and a wide geographic area.
	Detour	Routes a message through intermediate steps to perform validation, testing, or debugging functions.
	Message History	Lists all applications that the message passed through since its origination.
	Message Store	Reports against message information without disturbing the loosely coupled and transient nature of a messaging system.
	Smart Proxy	Tracks messages on a service that publishes reply messages to the Return Address specified by the requestor.
	Test Message	Ensures the health of message processing components by preventing situations such as garbling outgoing messages due to an internal fault.
	Wire Tap	Inspects messages that travel on a Point-to-Point Channel EIP.

Channel Purger

This section explains, through an example scenario, how the Channel Purger EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Channel Purger
- Example scenario
    - Environment setup

- ESB configuration
- Simulating the sample scenario
- How the implementation works

Introduction to Channel Purger

The Channel Purger EIP removes unwanted messages, which can disturb tests or running systems, from a channel. Channel Purger is also useful when you need to reset a messaging system to a consistent state. For more information, refer to http://www.eaipatterns.com/ChannelPurger.html.

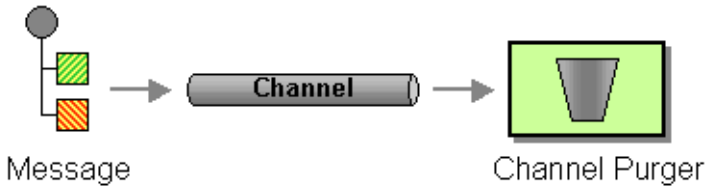


Figure 1: Channel Purger EIP

Example scenario

This example scenario is a stock quote service, and we send several stock quote requests to a message store to fill it. We then demonstrate how a user can delete left-over messages in a particular store through the management console UI of WSO2 ESB.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

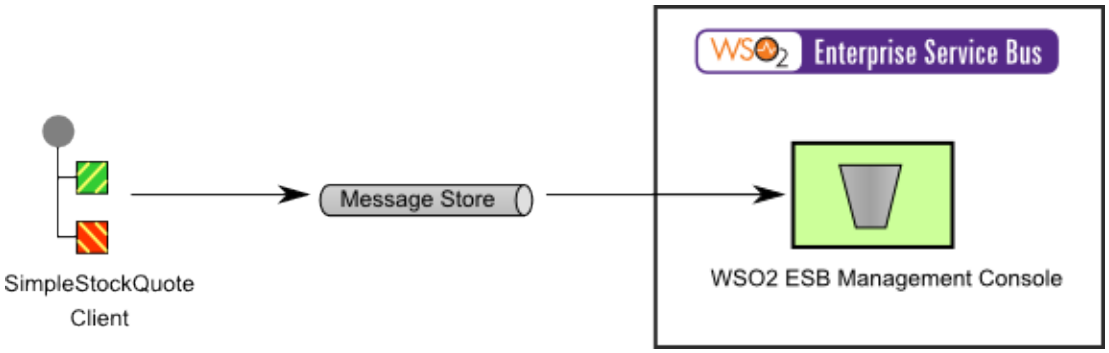


Figure 2: Example Scenario of the Channel Purger EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Channel Purger EIP by comparing their core components.

Channel Purger EIP (Figure 1)	Channel Purger Example Scenario (Figure 2)
Messages	Simple Stock Quote Requests
Channel	Message Store
Channel Purger	WSO2 ESB Management Console

Environment setup

1. Download and install the WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.

2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

1. Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

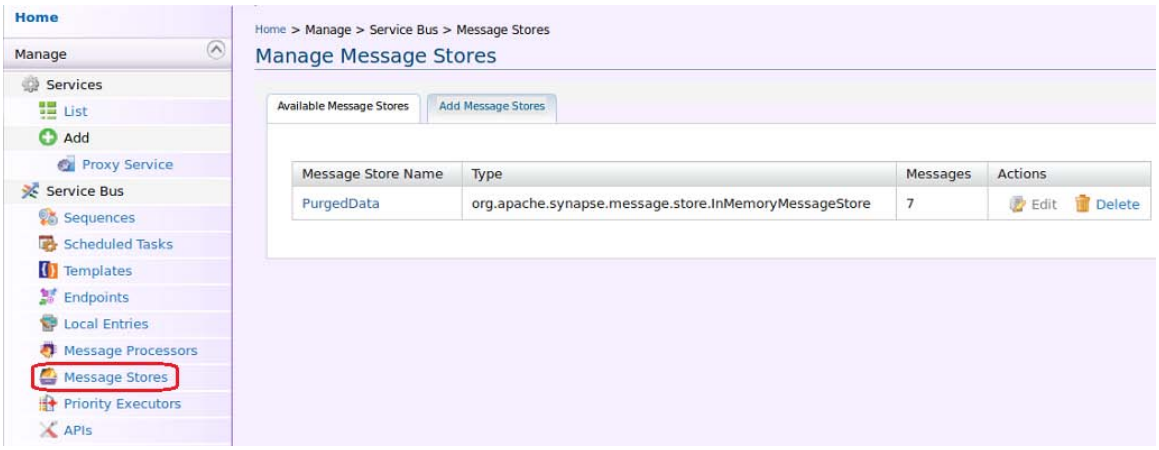
```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE"
expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <store messageStore="PurgedData"/>
   </sequence>
   <messageStore name="PurgedData"/>
</definitions>
```

2. Send several stock quote requests to the message store using the following command.
   `ant stockquote -Dtrpurl=http://localhost:8280/ -Dsymbol=Foo`

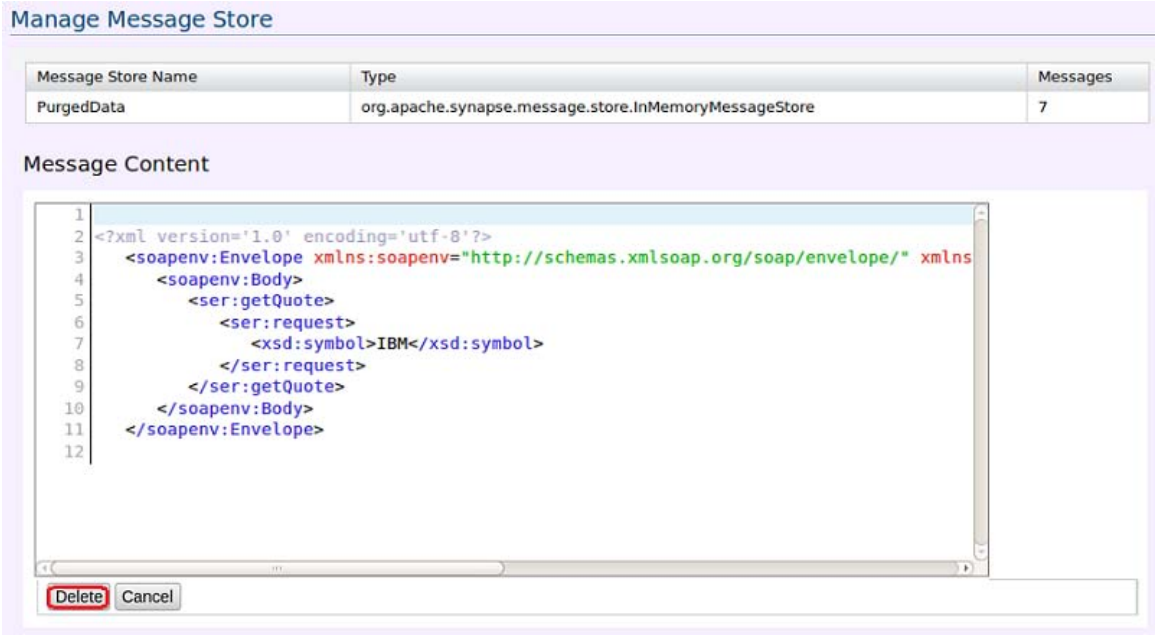
Simulating the sample scenario

Now that the message store is filled with several requests, let's delete some of them using the ESB management console.

1. Open the management console UI, click the **Main** menu and then **Service Bus**. Next, click the **Message Stores** sub menu. This will open the **Manage Message Stores** window.



2. Click the **PurgedData** option in the store. You will see the list of messages sent earlier. Click **Show Envelope** to view the message.



3. Click **Delete** to delete individual messages or **Delete All** to delete all messages at once.

How the implementation works

- **Sequence** [line 10 in ESB config] - The main sequence is the default sequence for WSO2 ESB.
- **Store** [line 11 in ESB config] - The Store mediator stores messages in the given message store.
- **messageStore** [line 13 in ESB config] - Defines a new message store with the name `PurgedData`. Message purging steps are given above in the section Simulating the Sample Scenario.

Control Bus

This section explains, through an example scenario, how the Control Bus EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Control Bus
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Control Bus

Effective network management is important in distributed networking environments. The Control Bus EIP helps effectively administer a messaging system that is distributed across multiple platforms and a wide geographic area. It uses the same messaging mechanism used by the application data, but separate channels are used to transmit data that is relevant to the management of components involved in the message flow. For more information, refer to http://www.eaipatterns.com/ControlBus.html.

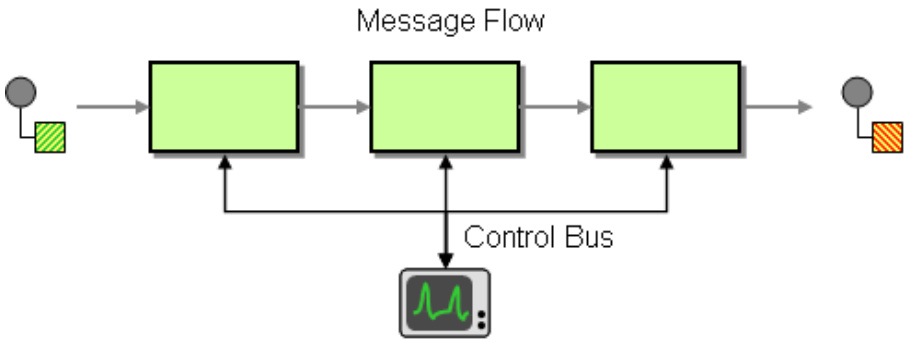


Figure 1: Control Bus EIP

Example scenario

In this example scenario, we use the Throttle mediator of WSO2 ESB to control access through throttling policies. Based on the policy, if the control flow is in order, it will be directed to `onAccept`. If not, it will be directed to `onReject`.

The diagram below depicts how to simulate the example scenario using WSO2 ESB.

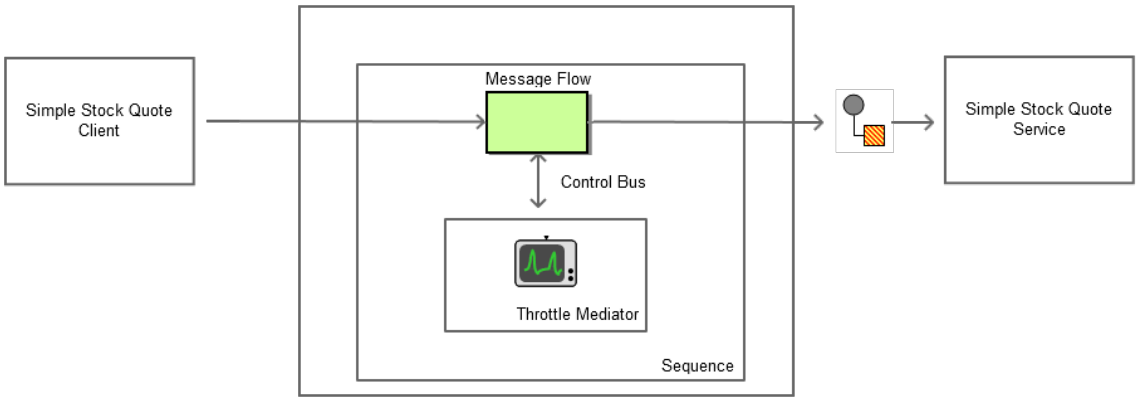


Figure 2: Example Scenario of the Control Bus EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Control Bus EIP by comparing their core components.

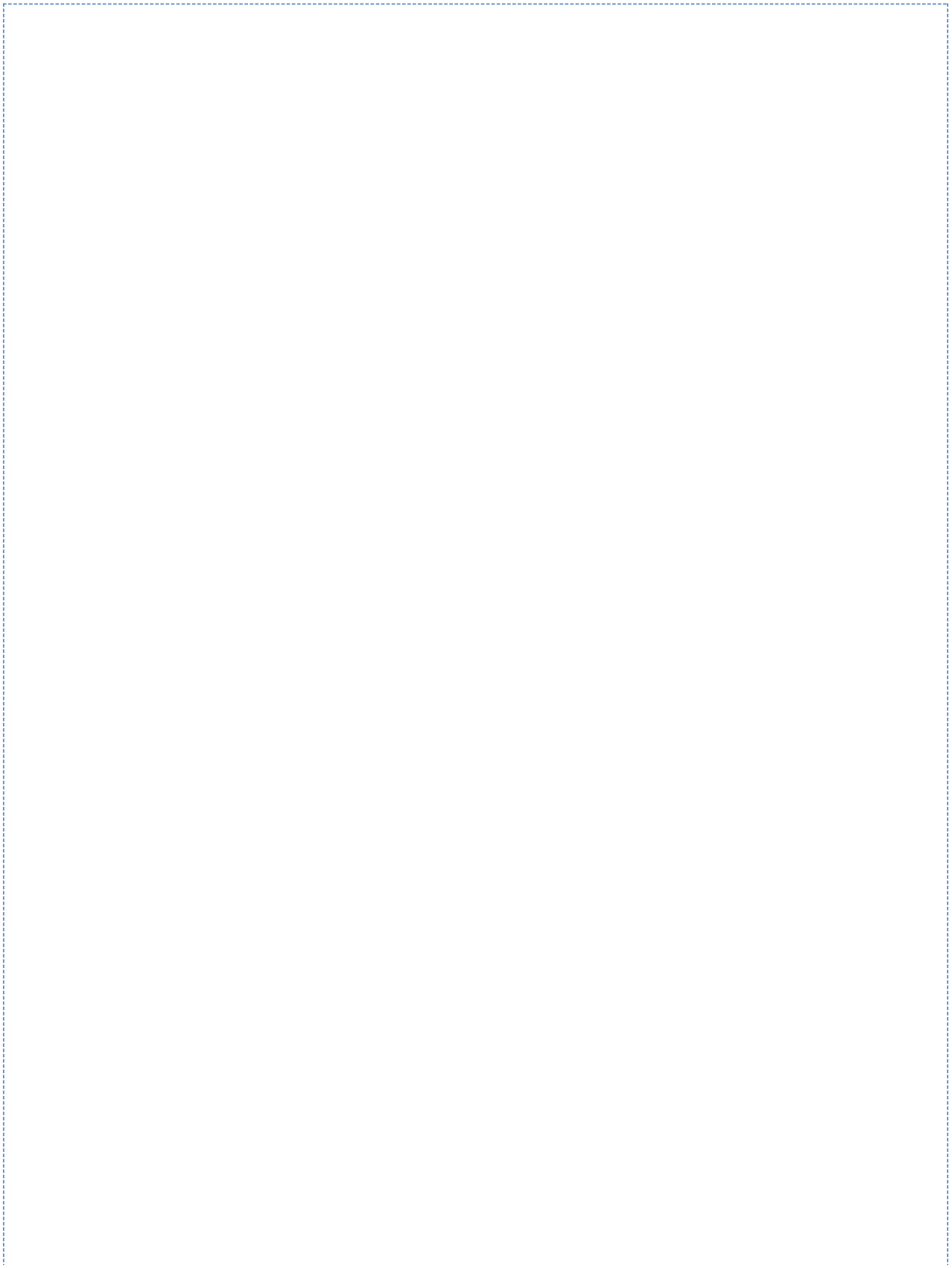
Control Bus EIP (Figure 1)	Control Bus Example Scenario (Figure 2)
Message Flow	Message Flow
Control Bus	Throttle Mediator

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.



```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in>
         <throttle id="A">
            <policy>
               <wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"

xmlns:throttle="http://www.wso2.org/products/wso2commons/throttle">
                  <throttle:ThrottleAssertion>

<throttle:MaximumConcurrentAccess>4</throttle:MaximumConcurrentAccess>
                  </throttle:ThrottleAssertion>
               </wsp:Policy>
            </policy>
            <onReject>
               <log level="custom">
                  <property name="text" value="Access Denied By Message System"/>
               </log>
               <makefault>
                  <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
value="tns:Receiver"/>
                  <reason value="Access Denied By Messaging System"/>
               </makefault>
               <property name="RESPONSE" value="true"/>
               <header name="To" action="remove"/>
               <send/>
               <drop/>
            </onReject>
            <onAccept>
               <log level="custom">
                  <property name="text" value="Access Granted By Message System"/>
               </log>
               <send>
                  <endpoint>
                     <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
                  </endpoint>
               </send>
            </onAccept>
         </throttle>
      </in>
      <out>
         <throttle id="A"/>
         <send/>
      </out>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send more than four requests to WSO2 ESB using the Stock Quote client as follows. For information about the Stock Quote client, refer to the Sample Clients section in the WSO2 ESB documentation.

```
ant stockquote -Dtrpurl=http://localhost:8280/ -Dsymbol=Foo
```

If more than four requests are sent simultaneously to the server, all other request will not be accepted. The Control bus regulates the message request flow through throttling.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **sequence** [line 11 in ESB config] - The main sequence, which is the default in the ESB.
- **throttle** [line 13 in ESB config] - Defines a throttling policy in WS-Policy.
- **WS-Policy Assertion** [line 18 in ESB config] - The WS-Policy assertion `throttle:MaximumConcurrentAccess` defines the number of concurrent connections that can be made.
- **onReject** [line 22 in ESB config] - The `onReject` element of the Throttle mediator defines what to do if the policy is rejected. The maximum number of concurrent connections at present exceeds 4. In this example, if the policy is rejected, the message is logged and a fault is returned to the client using `makeFault` with message 'access is denied'.
- **onAccept** [line 35 in ESB config] - The `onAccept` element of the Throttle mediator defines what to do when the policy is accepted. The maximum number of concurrent connections at present does not exceed 4. In this example, if the policy is accepted, the message is passed on to the service endpoint.

Detour

This section explains, through an example scenario, how the Detour EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Detour

The Detour EIP routes a message through intermediate steps to perform validation, testing, or debugging functions. Detour has two states. In one state, the router routes incoming messages through additional steps, while in the other, it routes messages directly to the destination channel. For more information, refer to http://www.eaipatterns.com/Detour.html.

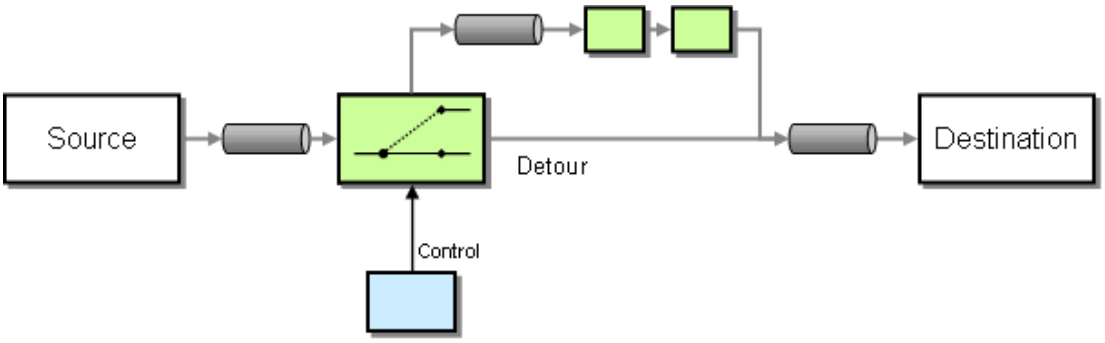


Figure 1: Detour EIP

Example scenario

This example scenario demonstrates how to use the Filter and Validate mediators of WSO2 ESB to implement the Detour pattern. First, the message is filtered using the `source` and `regex` parameters. Next the filtered message is validated. Messages are then forwarded to the appropriate destination.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

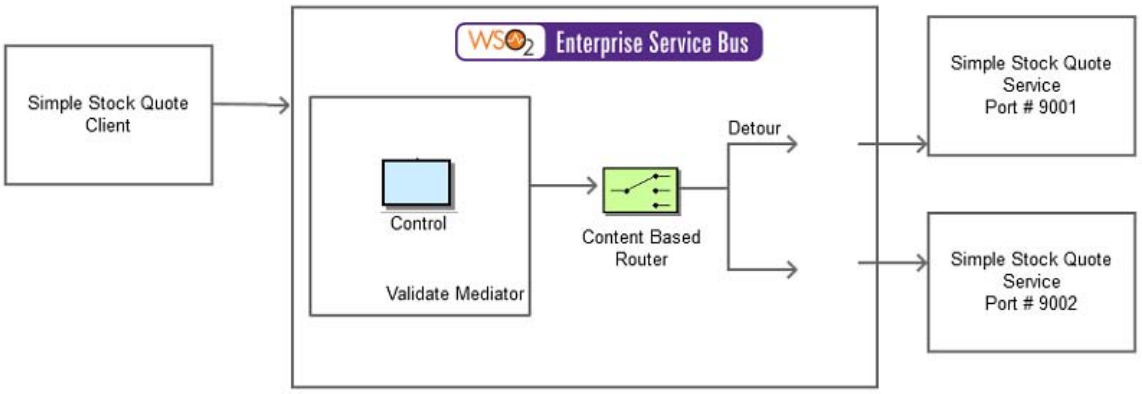


Figure 2: Example Scenario of the Detour EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Detour EIP by comparing their core components.

Detour EIP (Figure 1)	Detour Example Scenario (Figure 2)
Source	Simple Stock Quote Client
Control	XML Schema Validation using Validate Mediator
Detour	Content Based Routing
Destination	Simple Stock Quote Service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances in ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <localEntry key="validate_schema">
      <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 xmlns="http://services.samples"
                 elementFormDefault="qualified"
                 attributeFormDefault="unqualified"
                 targetNamespace="http://services.samples">
         <xs:element name="getQuote">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="request">
                     <xs:complexType>
                        <xs:sequence>
                           <xs:element name="stocksymbol" type="xs:string"/>
                        </xs:sequence>
                     </xs:complexType>
                  </xs:element>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
      </xs:schema>
   </localEntry>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in>
         <filter source="get-property('To')" regex=".*StockQuote.*">
            <validate>
               <schema key="validate_schema"/>
               <on-fail>
                  <makefault>
                     <code xmlns:tns="http://www.w3.org/2003/05/soap-envelope"
value="tns:Receiver"/>
                     <reason value="Invalid custom quote request"/>
                  </makefault>
               </on-fail>
            </validate>
         </filter>
         <send/>
      </in>
      <send/>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send a request using the Stock Quote client to WSO2 ESB in the following manner. For information about the Stock Quote client, refer to the section Sample Clients in the WSO2 ESB documentation.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280
```

Based on the **To** endpoint reference of `http://localhost:9000/services/SimpleStockQuoteService`, the ESB performs a comparison to the path `StockQuote`, and if the request matches the XPath expression of the filter mediator, the filter mediator's child mediators will be executed. Next, the child mediators start validating the request. Because we are sending the `stockquote` request using `ant stockquote ...` commands, we will get a fault as 'Invalid custom quote request'. This indicates that the schema validation has failed, which happens because the schema used in the example expects a `stocksymbol` element instead of a `symbol` to specify the stock symbol.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **localEntry** [line 2 in ESB config] - Specifies a local entry containing the XML schema to use for validation.
- **sequence** [line 31 in ESB config] - The main sequence for the ESB, which is the default sequence.
- **filter** [line 33 in ESB config] - The filter mediator that is used to provide content based routing. In this case, the SOAP Header field `To` is matched against a regular expression that checks to see whether the `To` field contains `StockQuote`.
- **validate** [line 34 in ESB config] - Performs validation on the SOAP body using the XML Schema defined in the `localEntry` (line 2 in ESB config).
- **on-fail** [line 36 in ESB config] - If the validation fails, the on-fail element provides a detour, which can be used to channel the message through a different route and ultimately passed on to the intended service. In the example scenario, a fault is produced and the fault is sent back to the requesting client.
- **send** [line 44 in ESB config] - the send mediator is used to send the message to the intended receiver - however if the `makeFault` inside the on-fail element (line 37 in ESB config) is executed before this mediator is reached, then the fault is returned to the requesting client.

Message History

This section explains, through an example scenario, how the Message History EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Message History

The Message History EIP provides a list of all applications that the message passed through since its origination. It helps analyze and debug the flow of messages in a loosely coupled system. For more information, refer to http://www.eaipatterns.com/MessageHistory.html.

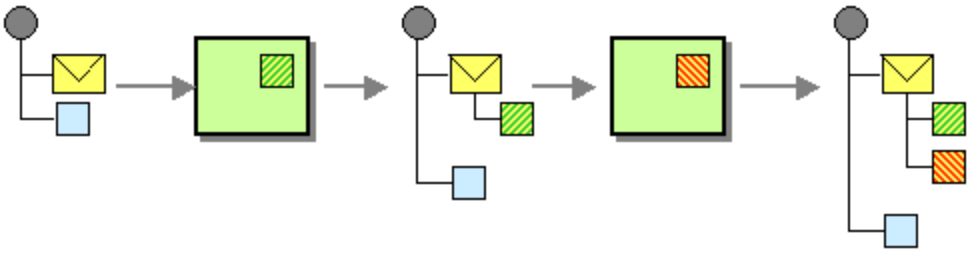


Figure 1: Message History EIP

Example scenario

Inside WSO2 ESB, a message travels through different paths and different types of mediations. This example scenario demonstrates how to track and remember, in a fine-grained manner, which mediation processes a message passes through inside the ESB. You can store a message in WSO2 ESB using a Property Mediator. These properties are assigned to the underlying Message Context of a particular sequence a message passes through. In this example, we show how each mediation updates the message history and finally shows it using a Log Mediator.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

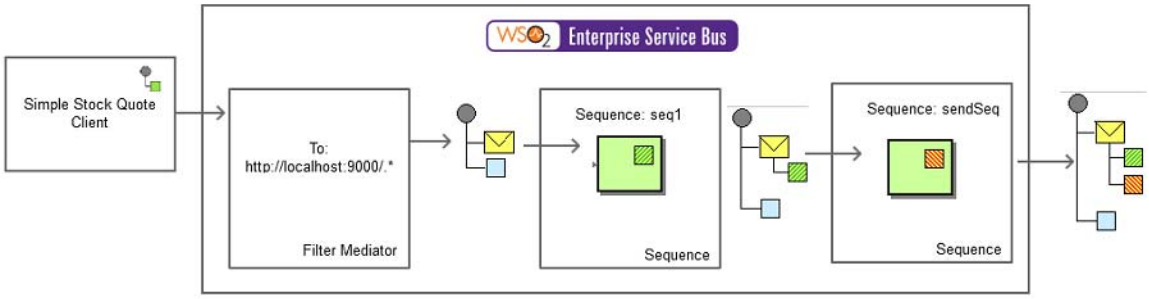


Figure 2: Example Scenario of the Message History EIP

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server instances in ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.


```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="sendSeq">
      <property name="sendSeq"
                value="*** At Sending Sequence ***"
                scope="default"
                type="STRING"/>
      <log level="custom">
         <property name="mainSeq" expression="get-property('mainSeq')"/>
         <property name="seq1" expression="get-property('seq1')"/>
         <property name="sendSeq" expression="get-property('seq1')"/>
      </log>
      <send>
         <endpoint>
            <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
         </endpoint>
      </send>
   </sequence>

   <sequence name="seq1">
      <property name="seq1"
                value="*** At Sequence 1 ***"
                scope="default"
                type="STRING"/>
      <sequence key="sendSeq"/>
   </sequence>

   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default 'fault' sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>

   <sequence name="main">
      <in>
         <filter xmlns:ns="http://org.apache.synapse/xsd"
                 source="get-property('To')"
                 regex="http://localhost:9000.*">
            <then>
               <property name="mainSeq" value="** At Main Sequence**"/>
               <sequence key="seq1"/>
            </then>
            <else/>
         </filter>
      </in>
      <out>
         <send/>
      </out>
      <description>The main sequence for the message mediation</description>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send a request using the Stock Quote client to WSO2 ESB in the following manner. For information about the Stock Quote client, refer to the section Sample Clients in the WSO2 ESB documentation.

```
ant stockquote -Daddurl=http://localhost:9000/services/SimpleStockQuoteService
-Dtrpurl=http://localhost:8280
```

Note that the message mediates though main, seq1, and sendSeq before reaching the endpoint. Each sequence adds a property to the message, and the final Log mediator at sendSeq puts out those properties as follows:

```
INFO - LogMediator mainSeq = ** At Main Sequence**, seq1 = *** At Sequence 1 ***,
sendSeq = *** At Sequence 1 ***
```

Similarly, you can add required entries to a message to track and display its mediation history.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown abovestep3.

- **Sequence** [line 19 in ESB config] - The sequence with key `seq1` defines a Message Context property called `seq1` and calls the sequence `sendSeq`.
- **Sequence** [line 36 in ESB config] - This is the main sequence that is invoked when the ESB receives a request. The main sequence filters the messages (line 38 in ESB config) to determine whether the request is going to an endpoint that begins with `http://localhost:9000` . If so, a Message Context property called `mainSeq` is defined, and the sequence `seq1` is invoked.
- **Sequence** [line 2 in ESB config] - The sequence with key `send_seq` defines a Message Context property called `sendSeq`. It then logs the properties by calling the `get-property` XPath function for the properties set in all the sequences, namely `mainSeq`, `seq1`, and `sendSeq`. After logging this information, the Send mediator is called to forward the message to the endpoint.

Message Store

This section explains, through an example scenario, how the Message Store EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Message Store

The Message Store EIP reports against message information without disturbing the loosely coupled and transient nature of a messaging system. It stores a duplicate of a message between each flow and captures the message information in a central location. For more information, refer to http://www.eaipatterns.com/MessageStore.html.

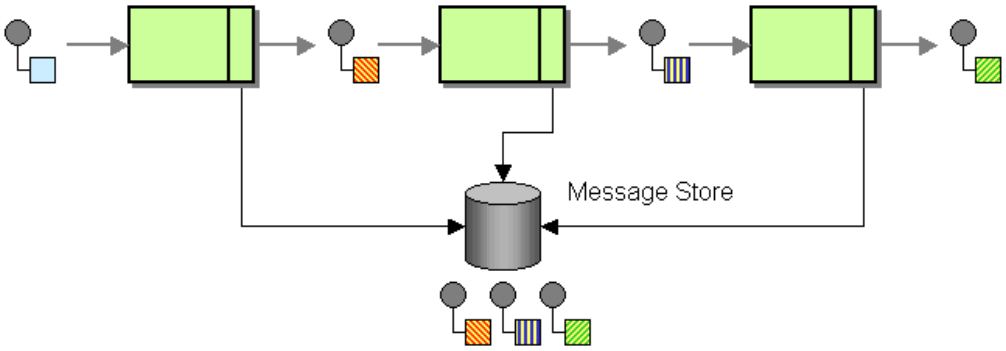


Figure 1: Message Store EIP

Example scenario

This example scenario is a stock quote service and demonstrates how Message Stores can be used to store the state of a message between each mediation cycle. We send several stock quote requests to the ESB using a sample client, and then use the ESB management console UI to check how those messages were stored when the request was sent and a reply has arrived.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

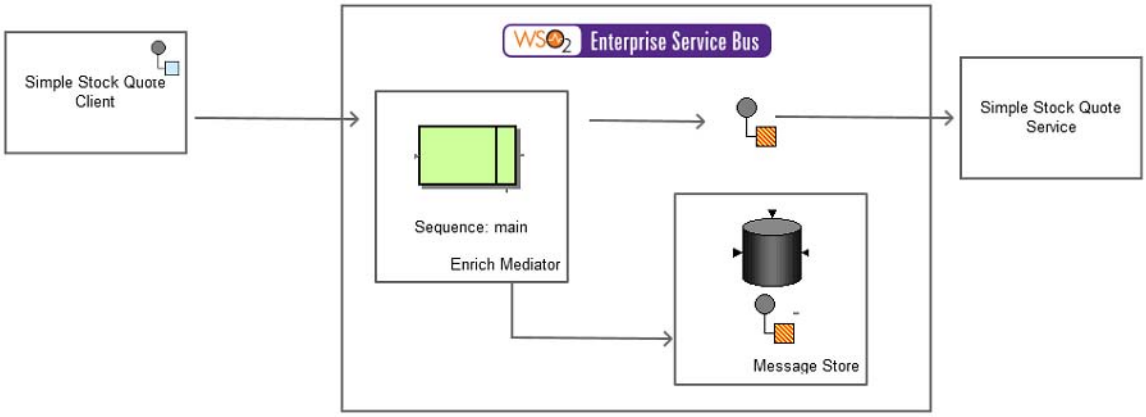


Figure 2: Example Scenario of the Message Store EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Message Store EIP by comparing their core components.

Message Store EIP (Figure 1)	Message Store Example Scenario (Figure 2)
Message Store	WSO2 ESB Message Store

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start the sample Axis2 server. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <localEntry key="Location">Foo</localEntry>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in>
         <enrich>
            <source type="inline" clone="true" key="Location"/>
            <target xmlns:m1="http://services.samples/xsd"
xpath="//m1:symbol/text()"/>
         </enrich>
         <store messageStore="CentralStorage"/>
         <send>
            <endpoint>
               <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
            </endpoint>
         </send>
      </in>
      <out>
         <store messageStore="CentralStorage"/>
         <send/>
      </out>
   </sequence>
   <messageStore name="CentralStorage"/>
</definitions>
```

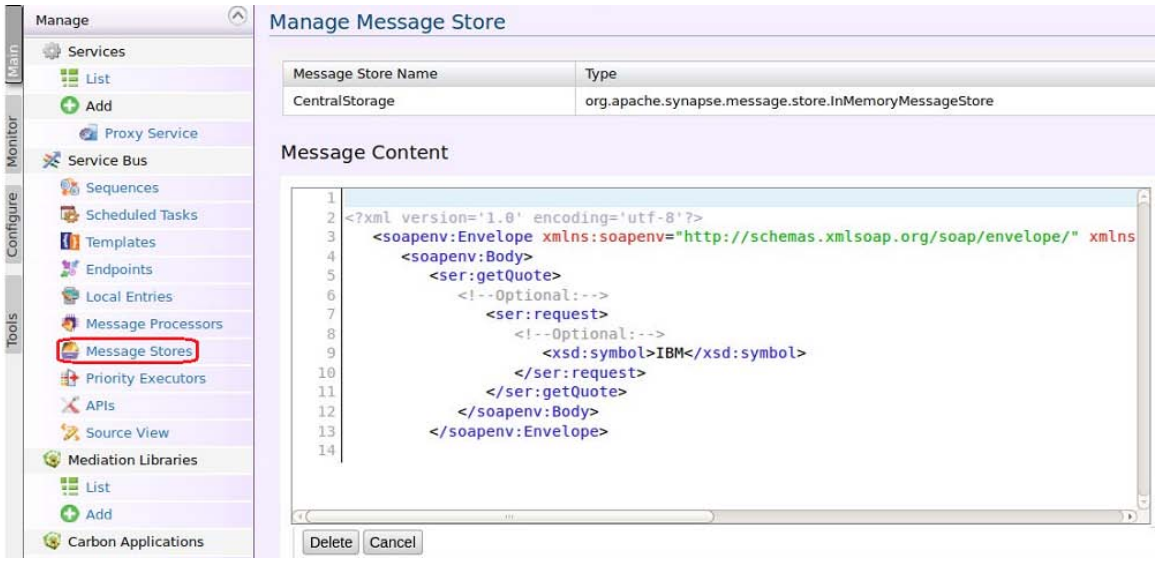
Simulating the sample scenario

1. Send the following request to the server several times using a SOAP client like SoapUI.

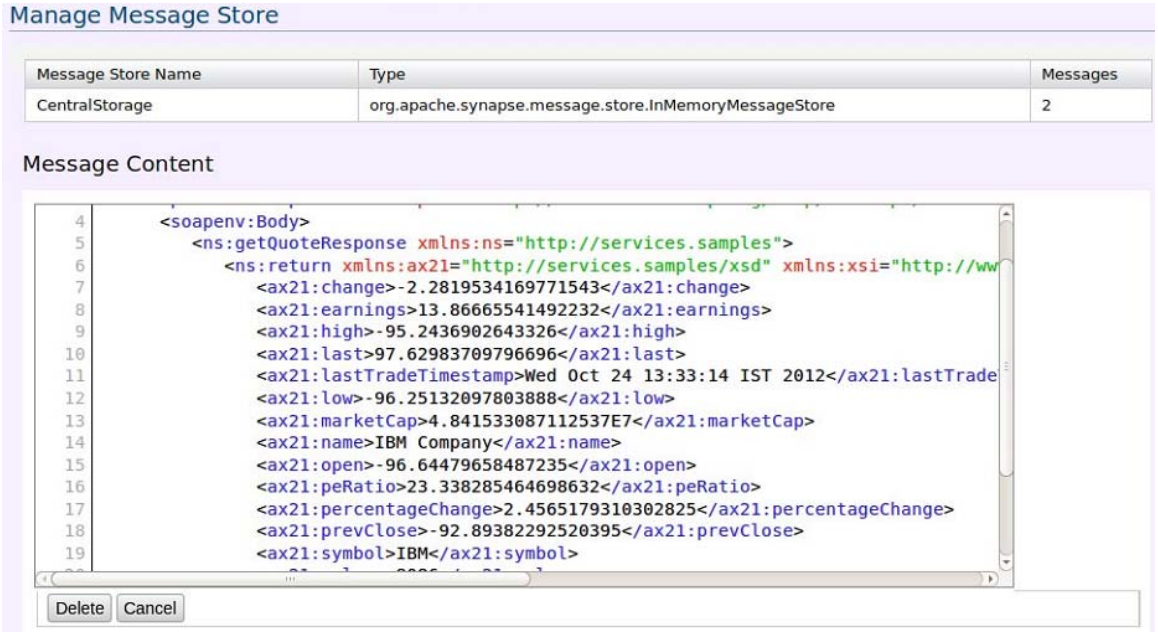
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:getQuote>
         <!--Optional:-->
         <ser:request>
            <!--Optional:-->
            <xsd:symbol>A</xsd:symbol>
         </ser:request>
      </ser:getQuote>
   </soapenv:Body>
</soapenv:Envelope>
```

2. In the ESB management console, click the **Main** menu and select **Message Stores** sub menu under the **Service Bus** menu. The messages you sent will be available under message stores. You can explore their content as follows.

   Message 1



   Message 2



How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **localEntry** [line 2 in ESB config] - Specifies the local entry that has the key `Location` and value `Foo`.
- **sequence** [line 11 in ESB config] - This is the main sequence invoked when a request is sent to the ESB. It has an `in` mediator, which defines the mediation to perform when a request is received, and an `out` mediator , which defines the mediation to perform when a response is received from a back-end service.
- **enrich** [line 13 in ESB config] - The Enrich mediator applies what is in its source element to what is in its target specified as an XPath expression. In this example, it applies the plain text value inside the `localEntry` with key `Location` as a text value into the `symbol` element in the SOAP body.
- **store** [line 17 in ESB] - The Store mediator stores the message inside the `CentralStorage` message store.

- **send** [line 18 in ESB config] - The message is then sent to the specified endpoint.
- **store** [line 25 in ESB config] - This store mediator is in the `out` sequence. It stores the message received as a response from the back-end service inside the `CentralStorage` message store.
- **messageStore** [line 29 in ESB config] - Defines a new message store with the name `CentralStorage.`

Smart Proxy

This section explains, through an example scenario, how the Smart Proxy EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Smart Proxy
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Smart Proxy

The Smart Proxy EIP tracks messages on a service that publishes reply messages to the Return Address specified by the requestor. It stores the Return Address supplied by the original requestor and replaces it with the address of

the `Smart Proxy`. When the service sends the reply message, the EIP routes it to the original Return Address. For more information, refer to http://www.eaipatterns.com/SmartProxy.html.

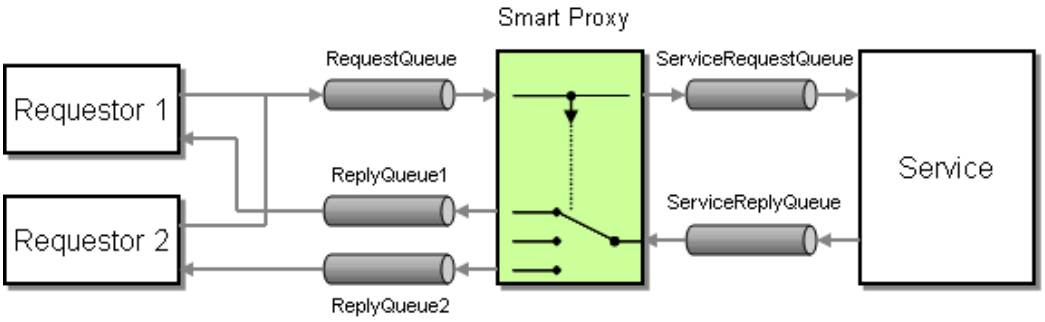


Figure 1: Smart Proxy EIP

Example scenario

This example scenario demonstrates a stock quote service, and a sample client sends a stock quote request to the ESB. The service that the client invokes is the `SmartProxy`, but through the ESB it manages to make calls to the back-end stock quote service. Smart Proxy simulates an address to the user. When the user sends a request, it will be diverted to another back-end server that will receive the response and send it back to the client. The ESB stores and manages information on what request the response should go to.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

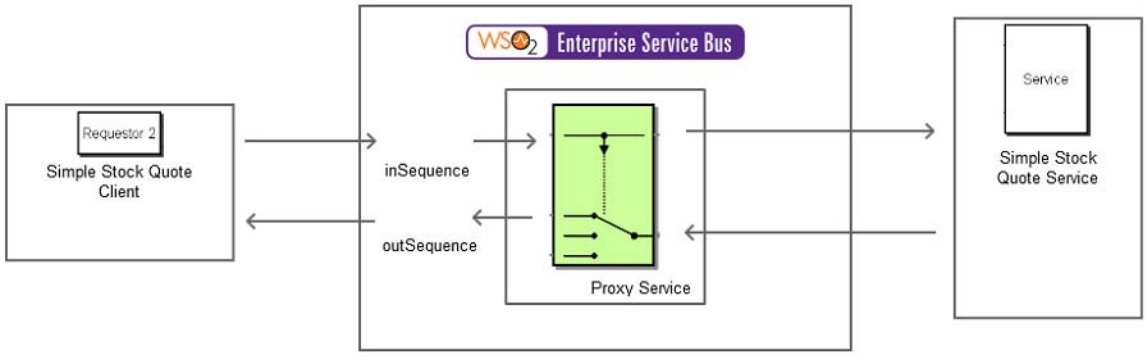


Figure 2: Example Scenario of the Smart Proxy EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Smart Proxy EIP by comparing their core components.

Smart Proxy EIP (Figure 1)	Smart Proxy Example Scenario (Figure 2)
Requestor	Simple Stock Quote Client
Smart Proxy	Proxy Service
Service	Simple Stock Quote Service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.

2. Start two Sample Axis2 server instances in ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <proxy name="SmartProxy" transports="http https" startOnLoad="true">
      <target>
         <inSequence>
            <send>
               <endpoint>
                  <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
               </endpoint>
            </send>
         </inSequence>
         <outSequence>
            <send/>
         </outSequence>
      </target>
   </proxy>
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in/>
      <out/>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send a request using the Stock Quote client to WSO2 ESB as follows. For information about the Stock Quote client, refer to the section Sample Clients in the WSO2 ESB documentation.

```
ant stockquote -Dtrpurl=http://localhost:8280/services/SmartProxy -Dsymbol=Foo
```

Note that the message is returned to the original requester.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **proxy** [line 2 in ESB config] - The proxy called `SmartProxy` passes incoming messages to the back-end

service.

- **send** [line 5 in ESB config] - The Send mediator sends the message to the back-end service. The ESB automatically sets the `Reply-To` header of the incoming request to itself before messages are forwarded to the back-end service.
- **send** [line 12 in ESB config] - The Send mediator inside the `outSequence` of the proxy sends response messages back to the original requester.

Test Message

This section explains, through an example scenario, how the Test Message EIP can be implemented using WSO2 ESB. The following topics are covered:

Introduction to Test Message

The Test Message EIP ensures the health of message processing components by preventing situations such as garbling outgoing messages due to an internal fault. For more information, refer to http://www.eaipatterns.com/TestMessage.html.

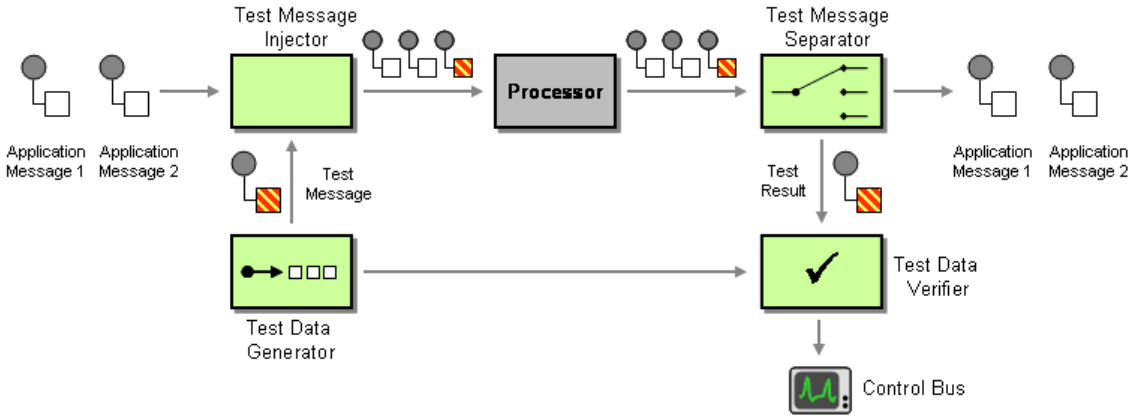


Figure 1: Test Message EIP

Example scenario

This example scenario demonstrates how to send a test message and determine the availability of a service. You can also see how the Scheduled Tasks in WSO2 ESB are used.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

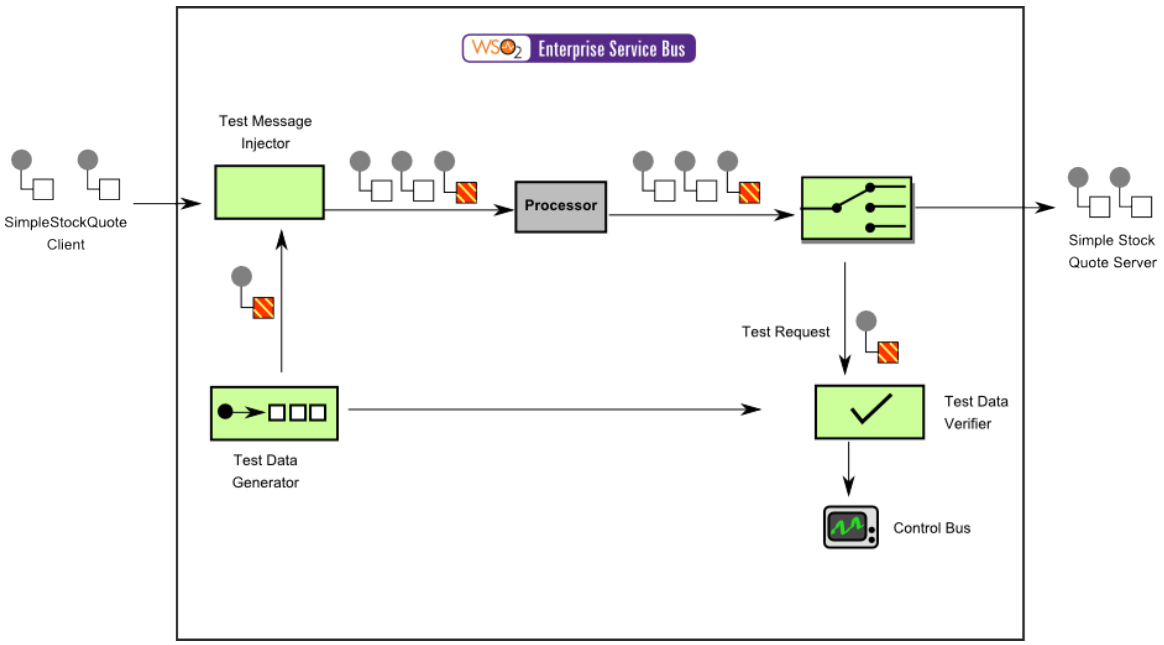


Figure 2: Example Scenario of the Test Message EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Detour EIP by comparing their core components.

Test Message EIP (Figure 1)	Example Scenario of the Test Message EIP (Figure 2)
Application Messages	Simple Stock Quote Client
Test Data Generator	Test data generator role is played by the Task Scheduler in the ESB. It creates a message after a given interval and hands over the message to the main sequence.
Test Message Injector	Main sequence injects messages it gets from Task scheduler into `sendSeq`, where the rest of the messages go.
Test Message Separator and Test Data Verifier	A Filter mediator inside `receiveSeq` filters out the test messages and prints the outcome of the test. There's another filter in fault c to monitor any failures in sending test messages.

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances in ports 9001 and 9002. For instructions, refer to the section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
```

```
 <!-- Proxy for normal message flow -->
 <proxy name="ServiceProxy" transports="https http" startOnLoad="true"
trace="disable">
  <target inSequence="sendSeq"/>
 </proxy>

 <!-- Normal flow of the messages -->
 <sequence name="sendSeq">
  <send receive="receiveSeq">
   <endpoint>
    <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
   </endpoint>
  </send>
 </sequence>

 <!-- Fault sequence handles failures -->
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default 'fault' sequence"/>
         <property xmlns:ns="http://org.apache.synapse/xsd"
                   name="ERROR_CODE"
                   expression="get-property('ERROR_CODE')"/>
         <property xmlns:ns="http://org.apache.synapse/xsd"
                   name="ERROR_MESSAGE"
                   expression="get-property('ERROR_MESSAGE')"/>
      </log>

  <!-- Filter the failed Test Messages -->
  <filter xmlns:ns="http://org.apache.synapse/xsd"
              xmlns:m0="http://services.samples"
              source="//m0:getQuote/m0:request/m0:symbol"
              regex="TEST">
   <then>
    <log>
     <property name="FAILURE" value="*** Test Message Failed ***"/>
     <property name="FAILED SERVICE" value="***
localhost:9000/services/SimpleStockQuoteService ***"/>
    </log>
   </then>
   <else/>
  </filter>
  <drop/>
 </sequence>

 <!-- Receiving messages from service -->
 <sequence name="receiveSeq">
  <log/>
  <!-- Filter the Test Messages -->
  <filter xmlns:ax21="http://services.samples/xsd"
    xmlns:ns="http://org.apache.synapse/xsd"
    source="//ax21:symbol"
    regex="TEST">
  <then>
   <log>
               <property name="TEST PASSED" value="*** Test Message Passed ***"/>
               <property name="TESTED SERVICE" value="***
localhost:9000/services/SimpleStockQuoteService ***"/>
            </log>
            <drop/>
```

```
          </then>
          <else>
             <send/>
          </else>
   </filter>
 </sequence>

<!-- main sequence used as the test Message injector -->
 <sequence name="main">
      <in>
         <sequence key="sendSeq"/>
      </in>
      <out>
         <send/>
      </out>
    </sequence>

 <!-- Task Scheduler act as Test data generator -->
 <task name="Testing"
  class="org.apache.synapse.startup.tasks.MessageInjector"
  group="synapse.simple.quartz">

  <!-- Interval between generating test messages, Cron Expression are also allowed
to use -->
  <trigger interval="25"/>

  <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
name="message">
   <m0:getQuote xmlns:m0="http://services.samples">
            <m0:request>
               <m0:symbol>TEST</m0:symbol>
            </m0:request>
   </m0:getQuote>
  </property>
  <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
               name="soapAction"
               value="urn:getQuote"/>
  <property xmlns:task="http://www.wso2.org/products/wso2commons/tasks"
               name="to"
               value="http://localhost:9000/services/SimpleStockQuoteService"/>
 </task>
</definitions>
```

Simulating the sample scenario

After setting this configuration, note that the Axis2 server is invoked every 25 seconds, and the following message on the WSO2 ESB console indicates that the test has passed.

```
INFO - LogMediator To: http://www.w3.org/2005/08/addressing/anonymous, WSAction: ,
SOAPAction: , MessageID: urn:uuid:fa29b7c1-98cb-48a2-87b3-54dc96b9c817, Direction:
response, TEST PASSED = *** Test Message Passed ***, TESTED SERVICE = ***
localhost:9000/services/SimpleStockQuoteService ***
```

If you stop the Axis2 server, you will get the following message indicating that the test has failed.

```
INFO - LogMediator To: http://localhost:9000/services/SimpleStockQuoteService,
WSAction: urn:getQuote, SOAPAction: urn:getQuote, MessageID:
urn:uuid:008c1cf5-a13b-44ec-8e1c-566a39275f67, Direction: request, FAILURE = ***
Test Message Failed ***, FAILED SERVICE = ***
localhost:9000/services/SimpleStockQuoteService ***
```

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **sequence** [line 8 in ESB config] - The sequence with key `sendSeq` defines the endpoint where messages will be sent and defines the sequence (receiveSeq) that will receive responses.
- **fault sequence** [line 17 in ESB config] - The fault sequence provides a channel for handling faults.
- **filter** [line 29 in ESB config] - A filter mediator is used within the fault sequence to determine whether the symbol property of the message passing through this filter contains `TEST`. If it does, it will be logged, and no fault message is passed onto the requesting client. If `TEST` does not exist, a fault message is passed as usual.
- **sequence** [line 66 in ESB config] - This is the main sequence invoked by default when a request is made to the ESB. This sequence calls the `sendSeq` sequence (line 8 in ESB config).
- **task** [line 76 in ESB config] - A task in the ESB runs periodically based on a given timer. This timer is specified using the trigger element. In this case, the task is set to run every 25 seconds. This task uses the Synapse `MessageInjector` class to inject a message into the Synapse environment.
- **property** [line 83 in ESB config] - This property, defined inside the task, specifies the message body.
- **property** [line 90 in ESB config] - This property, defined inside the task, sets the SOAP Action to `getQuote.`
- **property** [line 93 in ESB config] - This property, defined inside the task, specifies the address where the message generated by the task is sent.

Wire Tap

This section explains, through an example scenario, how the Wire Tap EIP can be implemented using WSO2 ESB. The following topics are covered:

- Introduction to Wire Tap
- Example scenario
  - Environment setup
  - ESB configuration
  - Simulating the sample scenario
  - How the implementation works

Introduction to Wire Tap

The Wire Tap EIP inspects messages that travel on a Point-to-Point Channel EIP. It inserts a simple Recipient List into the channel that publishes each incoming message to the main channel and a secondary channel. For more information, refer to http://www.eaipatterns.com/WireTap.html.

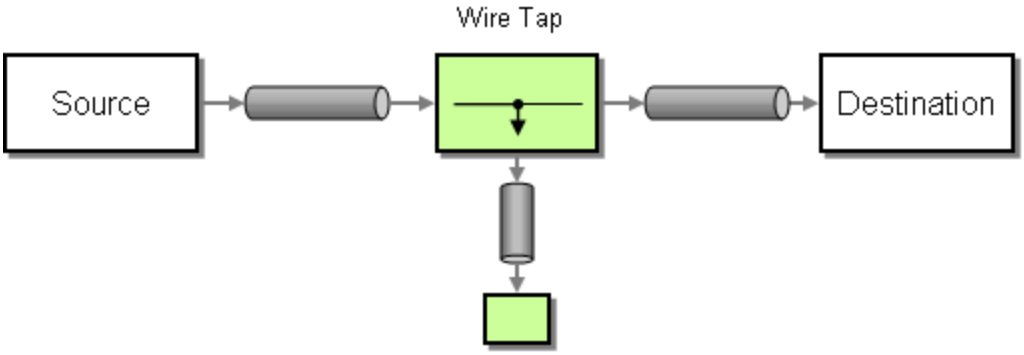


Figure 1: Wire Tap EIP

Example scenario

This example scenario demonstrates how the Log mediator in WSO2 ESB can be used to tap in between two message flows. The log mediator indicates the structure of the message in the ESB console each time it is called.

The diagram below depicts how to simulate the example scenario using the WSO2 ESB.

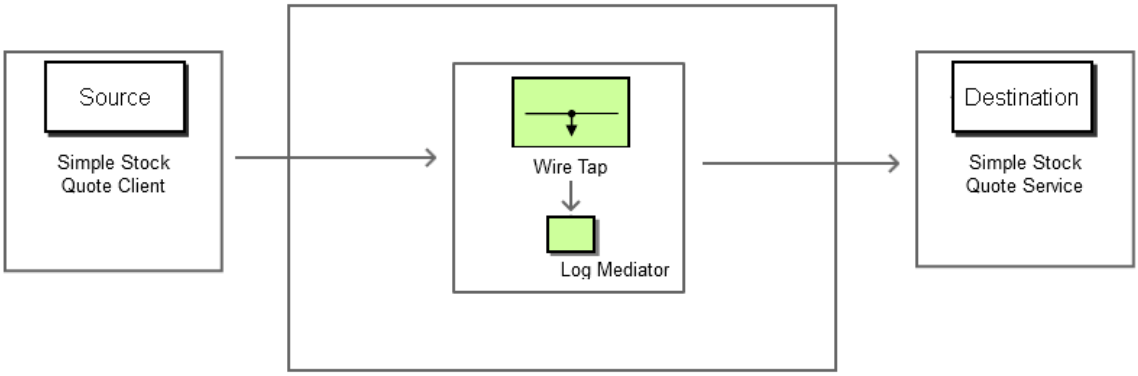


Figure 2: Example Scenario of the Wire Tap EIP

Before digging into implementation details, let's take a look at the relationship between the example scenario and the Wire Tap EIP by comparing their core components.

Wire Tap EIP (Figure 1)	Wire Tap Example Scenario (Figure 2)
Source	Simple Stock Quote Client
Wire Tap	Log Mediator
Destination	Simple Stock Quote Service

Environment setup

1. Download and install WSO2 ESB from http://wso2.com/products/enterprise-service-bus. For a list of prerequisites and step-by-step installation instructions, refer to Getting Started in the WSO2 ESB documentation.
2. Start two Sample Axis2 server instances in ports 9001 and 9002. For instructions, refer to section ESB Samples Setup - Starting Sample Back-End Services in the WSO2 ESB documentation.

ESB configuration

Start the ESB server and log into its management console UI (`https://localhost:9443/carbon`). In the management console, navigate to **Main** Menu, click **Service Bus** and then **Source View**. Next, copy and paste the following configuration, which helps you explore the example scenario, to the source view.

```
<definitions xmlns="http://ws.apache.org/ns/synapse">
   <sequence name="fault">
      <log level="full">
         <property name="MESSAGE" value="Executing default &#34;fault&#34;
sequence"/>
         <property name="ERROR_CODE" expression="get-property('ERROR_CODE')"/>
         <property name="ERROR_MESSAGE" expression="get-property('ERROR_MESSAGE')"/>
      </log>
      <drop/>
   </sequence>
   <sequence name="main">
      <in>
         <log level="full"/>
         <send>
            <endpoint>
               <address
uri="http://localhost:9000/services/SimpleStockQuoteService"/>
            </endpoint>
         </send>
      </in>
      <out>
         <log level="full"/>
         <send/>
      </out>
   </sequence>
</definitions>
```

Simulating the sample scenario

Send a request using the Stock Quote client to WSO2 ESB in the following manner. For information about the Stock Quote client, refer to the Sample Clients section in the WSO2 ESB documentation.

```
ant stockquote -Dtrpurl=http://localhost:8280/ -Dsymbol=Foo
```

Note that the ESB Log mediator taps into the message and displays it on the console.

How the implementation works

Let's investigate the elements of the ESB configuration in detail. The line numbers below refer to the ESB configuration shown above.

- **sequence** [line 10 in ESB config] - The main sequence is invoked when the a request is sent to WSO2 ESB.
- **log** [line 12 in ESB config] - The Log mediator with attribute level set to `full` logs the entire message that passes through.