Using JBI Components for integration – things you have to know about the component's concepts.

Adrien LOUIS, EBM WebSourcing

adrien . louis at ebmwebsourcing . com

http://www.ebmwebsourcing.com

10 avenue de l'Europe

31520 Ramonville Saint Agne

FRANCE

Abstract

This article presents the Java Business Integration specification (JSR 208 [1]), and describes more specifically the concept of "component" as defined in this specification.

First, we introduce the main goals of JBI, and then, we explain extensively the communication between components through the JBI environment, as well as component installation process.

1 JBI, a standard for SOA

Java Business Integration (JSR 208 [1]) specification defines a standard mean to assemble integration components in order to create integration solutions to enable the SOA (Service Oriented Architecture) in an Enterprise Information System.

Components are plugged into a JBI environment and can provide or consume services through it, in a loosely-coupled way. The JBI environment routes the exchanges between those components, and offers a set of technical services.

Two kinds of components can be plugged:

- Service Engines provide logic in the environment, such as XSL transformation, BPEL orchestration and so on.
- Binding components are sort of “connectors” to external services or applications. They allow communication with various protocols, such as SOAP, JMS, ebXML, …

JBI is built on top of state of the art SOA standards: service definitions are described in WSDL components exchanges XML messages following the document oriented model.

The JBI environment (also called JBI container) provides the glue between JBI components by acting as a Message Router to:

- Find a service provided by a component,
- Send request to a service-provider,
- Manage the message exchange life cycle (request, response, acknowledgements…).

It provides also a set of services (support of naming context, transactional context…)

On the other hand, the JBI container manages, through a rich management API, the installation and the life cycle of the components, and the deployment of artefacts to configure an installed component (for instance, deployment of XSL stylesheets to a transformation service engine).

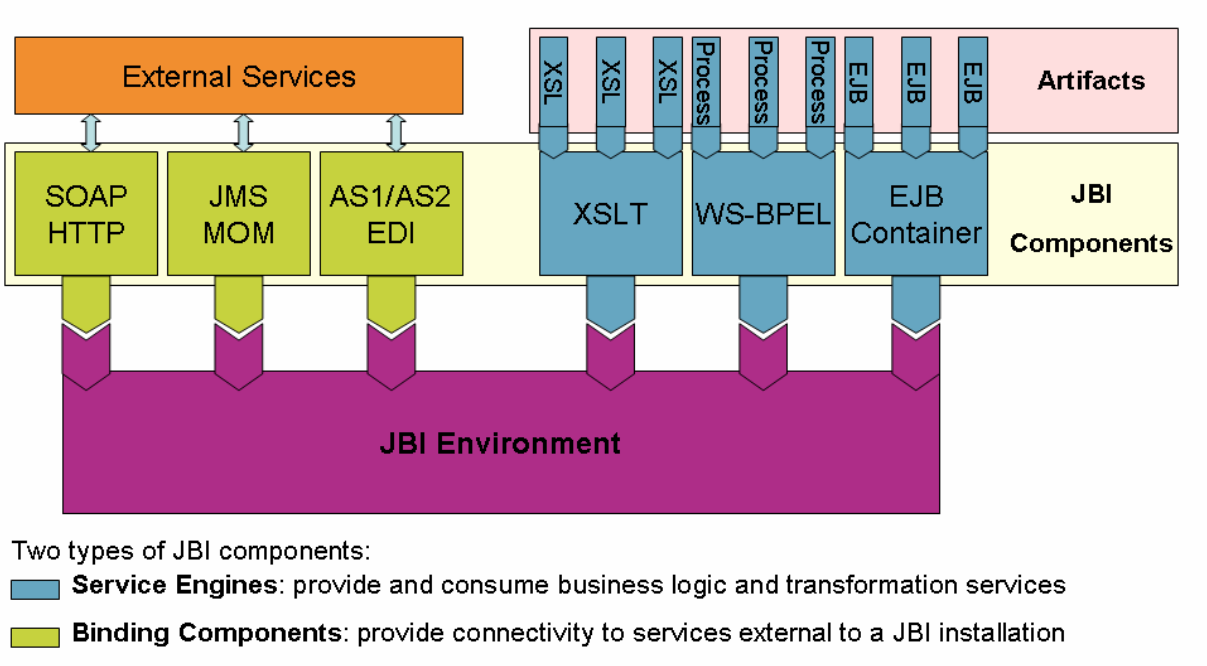

Figure 1 : JBI container

2 JBI Components

JBI Components are the base elements to be composed by the JBI container in order to create an integration solution.

The Components are plug-ins for the container, and are considered as “external”. As a J2EE container hosts EJBs, or a Portal hosts portlets, the JBI container hosts Integration Components. The Components have to be written in Java, or can be re-used, in the same way as you write or use EJBs or portlets.

2.1 Concepts

The notion of “JBI component” comes with several main concepts represented by the following SPI (System Programming Interface):

- The Component object, with which the container interacts to retrieve information about the component (services description…),
- The LifeCycle object, used by the container to manage the lifecycle of the component,
- The ComponentContext, given by the container to the component, for communication with the JBI environment.

Additional concepts are:

- The Bootstrap object, which provides all operations required at install/uninstall time,
- The ServiceUnitManager object, used to manage the deploying of artefacts on a component.

A Component is packaged as an archive (a Zip or Jar file).

This archive contains the classes of the component, the required libraries, and a descriptor file.

The way to plug a component into a JBI container is to use the management API provided by the container. This API allows you to provide to the container the location of your Component package. Then, the container processes the component archive and installs the Component.

From the Component point of view, two different phases are defined:

- The installation phase, in which the JBI-container installs the component and plug it to the Bus,
- The execution phase, in which the JBI-container interacts with the component.

2.2 Installation time

During the installation phase, the component must perform all extra processes needed for its execution like the creation of mandatory folders, the installation of a database…

This installation process is done by the Bootstrap object, which receives an *onInstall()* event from the container with an associated InstallationContext object, thus providing to the

Bootstrap some information about the installation (the path of the installation, a NamingContext, etc…).

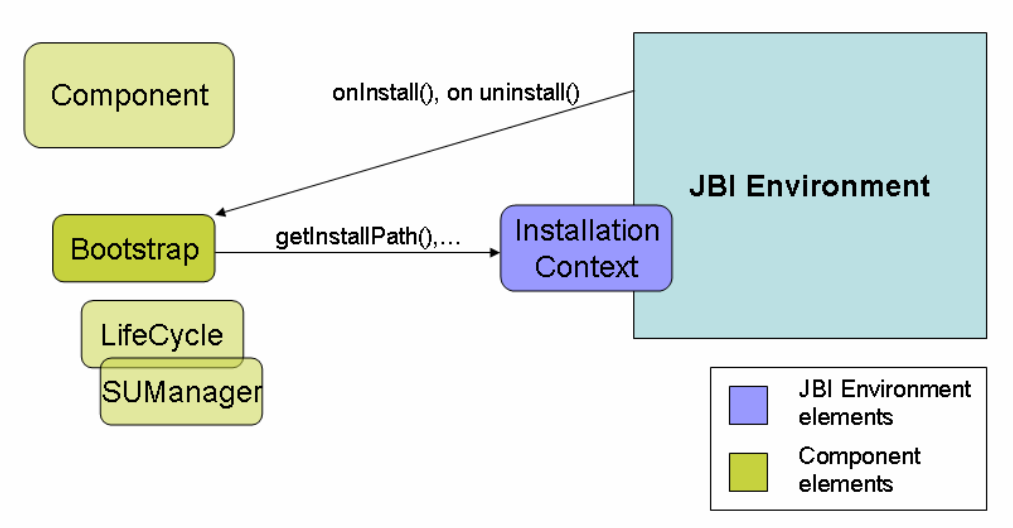

Figure 2 : interactions during the installation phase

2.3 Execution time

The Component is started, stopped, and shutdown by the container.

The container initializes the Component by passing it a ComponentContext, which is the entry point to the JBI environment. While the Component is running, it interacts with the JBI environment through this ComponentContext.

The Component can consume services (exposed on the bus by other component) by sending messages to a service-provider. As a service-provider, it can accept such messages, process them, and send an answer to the consumer through the Bus.

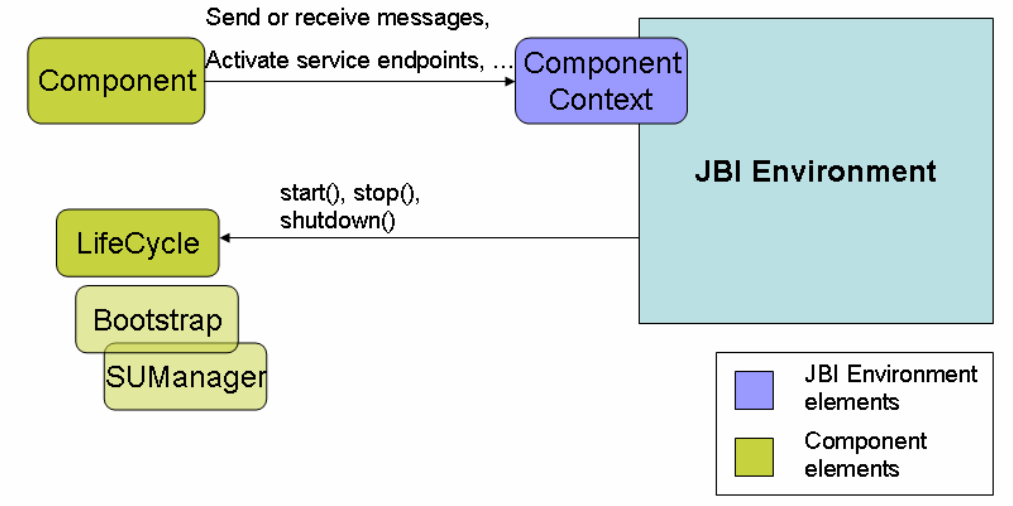

Figure 3 : interactions during the execution phase

3 Components interactions

To illustrate the interactions between a service consumer and a service provider, let's take a simple request-response exchange as example.

The corresponding message exchange pattern is an "InOut" exchange pattern, as defined in the JBI specification (see the 5.4.2 section of the JBI spec.).

JBI supports four WSDL pattern exchanges: In, InOut, InOptionalOut, and RobustIn. Each pattern defines a particular exchange sequence. As an extension, it is possible to support other MEP (Message Exchange Patterns).

3.1 Service consumer

Once a Component is running, it can find and consume the services that are registered in the JBI environment. Therefore, the component is in the role of a service consumer.

3.1.1 Find a service endpoint

An endpoint represents an address where a service provided by a component can be found.

Several components can provide the same service (with eventually different implementations), but each of those components have a unique endpoint.

To find a service, the consumer asks its ComponentContext for the list of all endpoints matching service name, by using the *getEndpointsForService(serviceName)* method.

The consumer can choose the provider that it wants to reach.

If the consumer already knows the address of the provider that it wants to contact, it can retrieve the provider Endpoint object by using a *getEndpoint(serviceName, endpointName)* method.

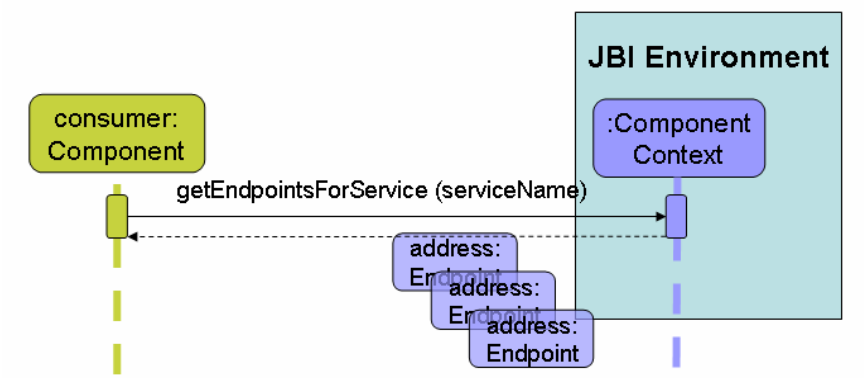

Figure 4 : get endpoints for a given service

3.1.2 Create a message exchange

To manipulate messages, the ComponentContext provides a DeliveryChannel object, which represents a bi-directional communication channel between the Component and the message-router of the JBI environment (called Normalized Message Router).

The DeliveryChannel is in charge of message-exchanges instantiation, and is the path through which the messages are sent to the NMR. Then, the NMR routes the message to the component which provides the requested service.

An exchange between the consumer and the provider is materialized by a MessageExchange object. This object serves during the whole life of the exchange.

When the consumer wants to initialize a new exchange, it asks a MessageExchangeFactory (provided by the DeliveryChannel object) to create a new MessageExchange.

This MessageExchange contains the actual content of the message and a set of meta-data, such as the provider-endpoint, the status of the exchange (active, done, in error), identification of the exchange "owner" (the consumer or the provider)… The MessageExchange is shared by the consumer and the provider during the exchange (the request, the response …).

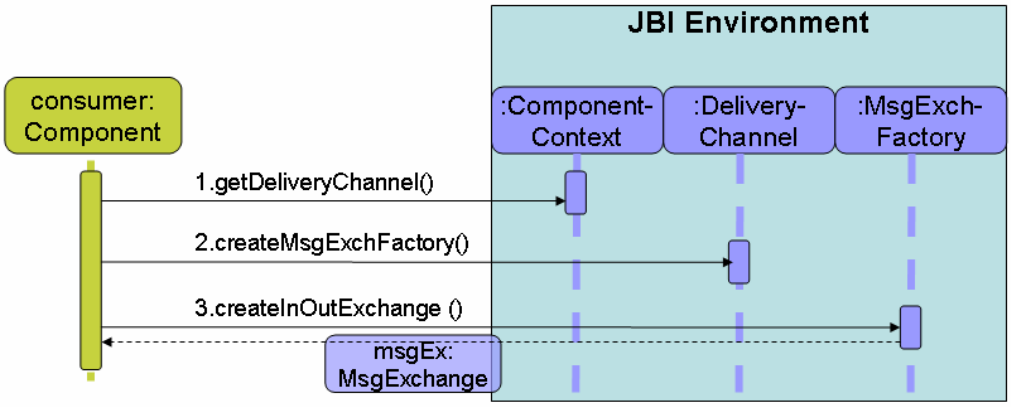

Figure 5 : create a message exchange

3.1.3 Send the message

Now that the consumer has instantiated a MessageExchange, it can set on this object the message it wants to send to the consumer.

The consumer has to set a NormalizedMessage as the "in" message of the exchange.

The NormalizedMessage is a JBI definition of a message. The consumer asks the MessageExchange to create a new NormalizedMessage.

Then, the consumer set on this NormalizedMessage the content of the message (an XML payload), and eventually some attachments.

The consumer has to set on the MessageExchange the Endpoint of the provider (previously retrieved), and the name of the operation to be performed.

Note: the consumer can omit to specify the Endpoint of the provider, and just specify a serviceName. In this case, the NMR will search all matching Endpoints, and choose one of them.

Finally, the consumer sends the MessageExchange using the DeliveryChannel.

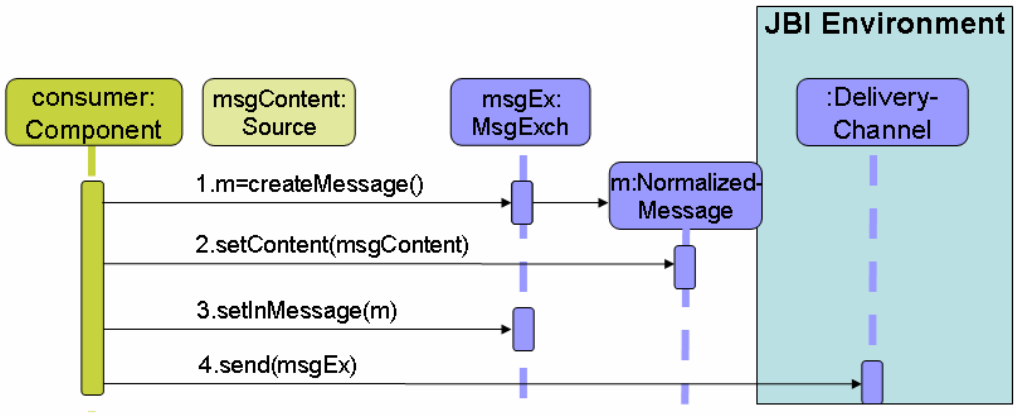

Figure 6 : send a message

3.2 Service provider

Once a Component is running, it can also provide services. This component acts as a service provider.

3.2.1 Activate an endpoint

The provider has to publish the services that it wants to offer.

The publication of a service is done by the *activateEndpoint(serviceName, endpointName)* method of the ComponentContext. This method returns the generated Endpoint object that references the new service in the JBI environment.

That done, other components can access the activated services, by finding the corresponding Endpoint with the *findEndpointForService()* method of their ComponentContext.

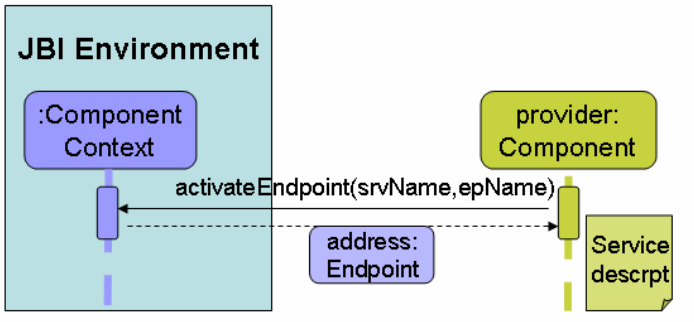

Figure 7 : activate a service - endpoint

3.2.2 Receive a message

Now that the provider has published some services, it can receive messages from other components (consumers).

When a consumer sends a message to a provider, the message (MessageExchange) is pushed in the message queue of the provider's DeliveryChannel.

The provider retrieves the received messages from the message queue with by doing *accept()* on its DeliveryChannel.

Once the provider obtains a MessageExchange, it can process it.

The provider gets the “in” message (the NormalizedMessage set by the consumer), the name of the operation to perform, the payload of the message…

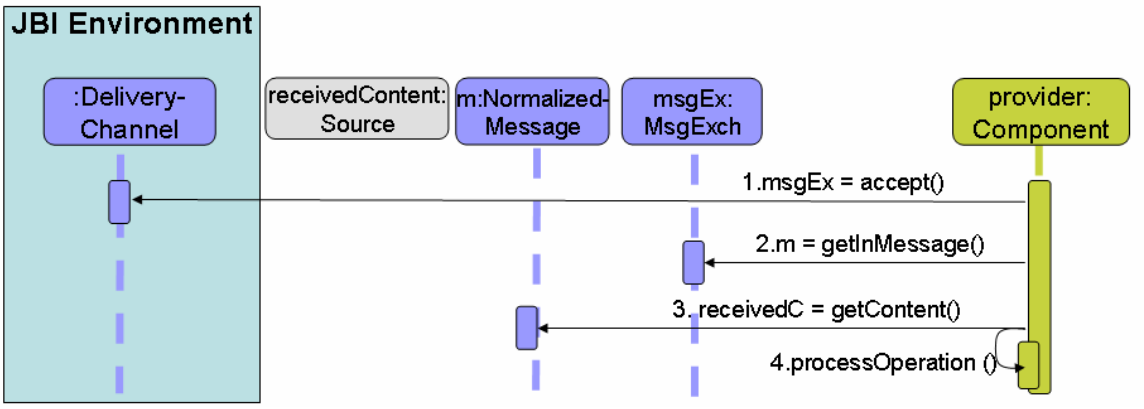

Figure 8 : receive a message

3.2.3 Send the response

If the operation requires an answer, the provider can set an “out” message on the MessageExchange, and send it again to the NMR, via its DeliveryChannel.

The NMR routes the MessageExchange to the consumer that previously initiated this MessageExchange.

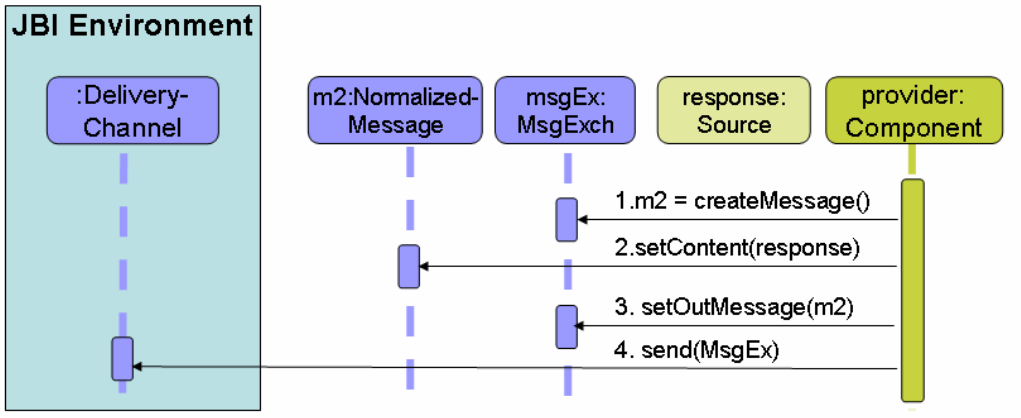

Figure 9 : send the response

3.3 Close the exchange

After the provider sent its response, the exchange is nearly complete.

The NMR routes the answer to the consumer, which receives it with an *accept()* on its DeliveryChannel.

The consumer process the “out” content of the MessageExchange, and has to close the exchange. To do this, the consumer set the status of the MessageExchange to DONE, and send it again to the NMR via the *send()* method of its DeliveryChannel.

The exchange is terminated, and the provider is notified of this status by receiving the MessageExchange.

The following schema describes the whole exchange process:

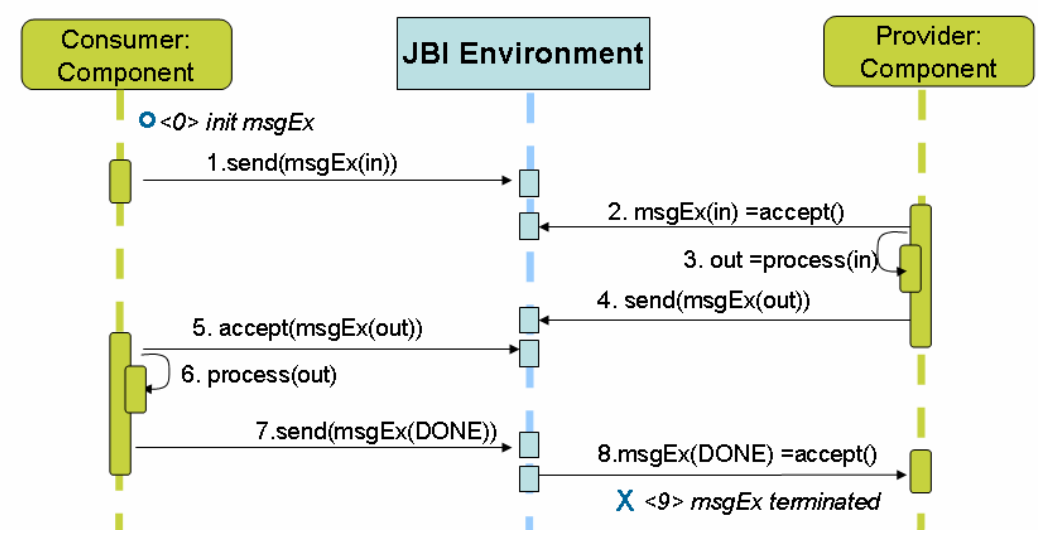

Figure 10 : the whole in-out exchange

4 HelloWorldService component

As seen before, a JBI component is a set of objects that have to implement some JBI interfaces. Additionally, a descriptor file has to be provided with this component.

In this section, we show most of the code to be implemented to create a simple HelloWorld ServiceEngine.

As a few lines of codes are necessary to implement the Component and the ComponentLifeCycle interfaces, a single object can implement these two interfaces.

To process requests, there is no “listener” mechanism proposed by the JBI specification. The way to receive a message is to block on an *accept()* method.

A good pattern is to create a separate object which is executed in another thread. This “listener” makes a loop on the *accept()* method, avoiding the Component to be blocked.

The complete sources of this example can be downloaded on the Wiki page of the **Petals** project [3].

4.1 Component – ComponentLifeCycle

The object that implements the Component and the ComponentLifeCycle interfaces just registers the “HelloWorldService” in the JBI environment, then creates and starts the HelloWorldListener thread, which is in charge of processing the incoming messages.

```
import javax.jbi.component.*;
public class Helloworld implements Component, ComponentLifeCycle {
   private ComponentContext context;
   private HelloworldListener listener;
   …
   public void init(ComponentContext context) throws JBIException{
         // keep a reference to the given ComponentContext
         this.context = context;
   }
   public void start() throws JBIException {
   // create our Listener and start it.
         listener = new
HelloworldListener(context.getDeliveryChannel());
          (new Thread(listener)).start();

         // register our service in the JBI environment
context.activateEndpoint(QName(“http://helloWorldService.com”) ,
 “HWendpoint”);
   }
```

```
    public void stop() throws JBIException {
          // just break the loop of our listener; the thread will stop
          listener.running= false;
    }
    public void shutDown() throws JBIException {
          // nothing
    }
    public ComponentLifeCycle getLifeCycle() {
          // this object acts as the LifeCycle, so it returns itself
          return this;
     }
…
}
```

4.2 HelloWorld listener

This object processes the MessageExchanges that pass through the NMR.

The received messages are retrieved from the DeliveryChannel object.

This listener checks that the specified operation name is “hello” and that the type of the exchange is an InOut exchange.

The listener retrieves the “in” content, processes the message (even if in this example there is no specific processing), and sets an “out” response to the MessageExchange.

Then, it sends the MessageExchange back to the NMR. The response is then conveyed to the consumer through the NMR.

```
import javax.jbi.messaging.*;

public class HelloworldListener implements Runnable {

    private DeliveryChannel channel;
    public boolean running;

    public HelloworldListener(DeliveryChannel channel){
          this.channel = channel;
    }
    /**
    * the main part of the listener
    */
    public void run()
    {
          running= true;
```

```
         while(running) {
               // block on the accept() method
               MessageExchange messageExchange = channel.accept();
               // a MessageExchange is received, so process it
               process(messageExchange);
         }
   }
/**
* process the received messages
*/
   private void process(MessageExchange msg)
   {
         if( new QName("hello").equals(msg.getOperation())  && msg
instanceof InOut)
         {
               // we received a InOut exchange, with the “hello”
operation
               InOut inOut = (InOut)msg;

               // Read the IN message
               NormalizedMessage in = inOut.getInMessage();
                           Source content = in.getContent();
               // … process the in-content (omitted)

               // Write the response
               NormalizedMessage out = inOut.createMessage();
               // … Set out content (omitted)
               InOut.setOut(out);

               // Send the response
               channel.send();
         }
   }
}
```

4.3 Bootstrap

No special operation is required during the installation of this HelloWorld component, so the class that implements the Bootstrap interface has nothing to do.

4.4 Package the component

Let assume that the 3 component classes (component, listener and bootstrap) are archived in a “helloComponent.jar” file.

A descriptor has to be provided to help the JBI container during the installation phase.

This descriptor file (jbi.xml) looks like this:

```
<jbi version="1.0" xmlns='http://java.sun.com/xml/ns/jbi'
   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'>
   <component type="service-engine">
     <identification>
         <name>HelloworldComponent</name>
         <description>A Helloworld Component</description>
     </identification>
     <component-class-name>hello.Helloworld</component-class-name>
     <component-class-path>
         <path-element>helloComponent.jar</path-element>
     </component-class-path>
     <bootstrap-class-name>hello.Bootstrap</bootstrap-class-name>
     <bootstrap-class-path>
         <path-element>helloComponent.jar</path-element>
     </bootstrap-class-path>
   </component>
</jbi>
```

The packaging of the component is a simple archive (Zip file or Jar file) with the following structure:

- helloComponent.jar
- <META-INF>
    - jbi.xml

4.5 Use the Helloworld service

A component that wants to access the service provided by this HelloWorld component can use the following code:

```
// Find the endpoint
ServiceEndpoint ep =
context.getEndpoint( new QName(“http://helloWorldService.com”) ,
                     “HWendpoint”);


// Create the exchange
MessageExchangeFactory factory =
deliveryChannel().createExchangeFactory();
```

```
MessageExchange msgEx = factory.createInOutExchange();
msgEx.setOperation( new QName("hello"));
msgEx.setEndpoint(ep);

// Create the message
NormalizedMessage nm = msgEx.createMessage();
// set the 'in' content - omitted
nm.setContent(source);
msgEx.setInMessage(nm);
deliveryChannel.send(msgEx);
```

When the HelloWorld service answers this request, the MessageExchange is sent back to the consumer. So, the listener of the consumer waits on the accept().

```
// wait for response
MessageExchange msgEx = deliveryChannel.accept();

// process the response from helloWorldService
if( msgEx.getEndpoint=ep && msg instanceof InOut)
   {
         InOut inOut = (InOut)msg;
         // Read the OUT message
         Source content = inOut.getOutMessage().getContent();
         // … process the out-content (omitted)

         // close the exchange
         msgEx.setStatus(ExchangeStatus.DONE);
         deliveryChannel.send(msgEx);
   }
```

Note: Each MessageExchange created has a unique ID. To be sure to process the response of a particular exchange, the consumer can keep the ID of the sent exchange, and compare it with the ID of the received message.

5 Install and start a JBI component

A JBI container offers administrative tools to manage components, through JMX MBean objects [4].

5.1 Install a component

The MBean that manage the installation of components is the InstallationServiceMBean. This service creates, for each component to install, another MBean object, an InstallerMBean.

The method to use is *loadNewInstaller(<archiveURL>)*. This method explodes and analyses the archive that pathname is specified as a parameter, and returns the name of the InstallerMBean created.

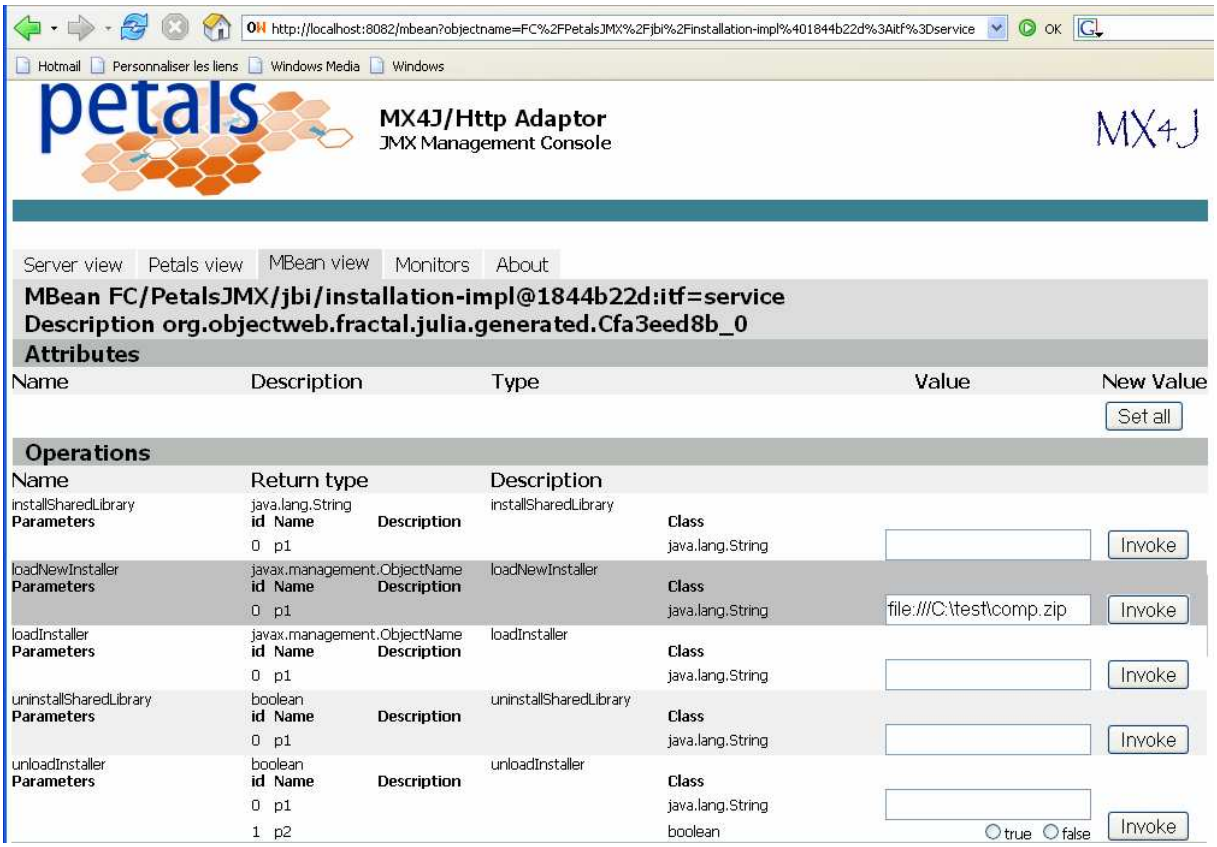

Figure 11 : InstallationService ::loadNewInstaller

The InstallerMBean performs the install/uninstall mechanism, with *install()* and *uninstall()* methods.

A call to *install()* instantiates the Bootstrap object of the component, as described in the jbi.xml descriptor file. Then, the *init()* and *onInstall()* methods are called on the Bootstrap.

Finally, the Component object is created, and a corresponding ComponentLifeCycleMBean object is also created. The *install()* method returns the name of this MBean.

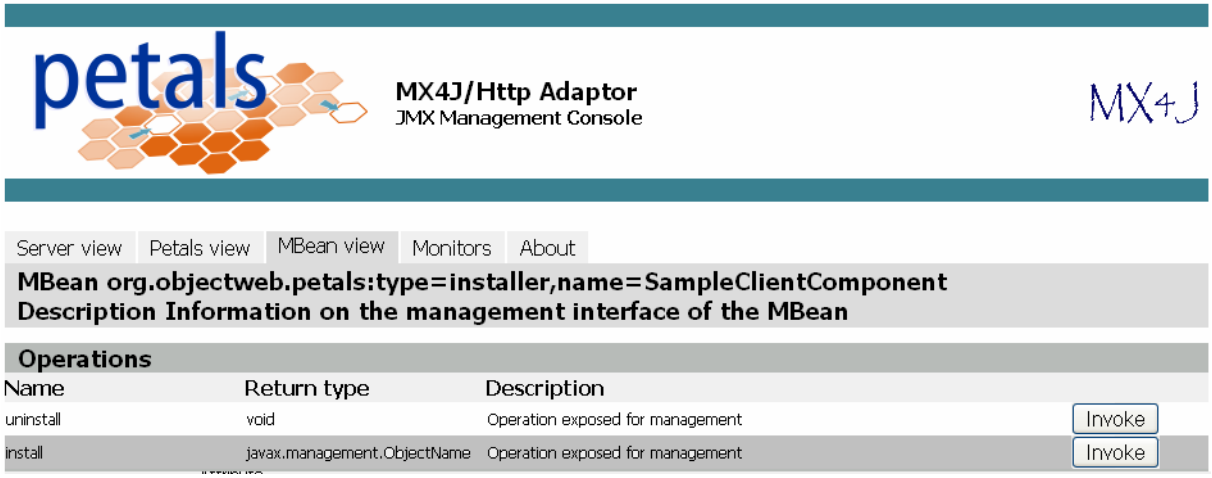

Figure 12 : Installer::install

Figure 13 : Installer::install - result

At this point, the component is installed, in a shutdown state.

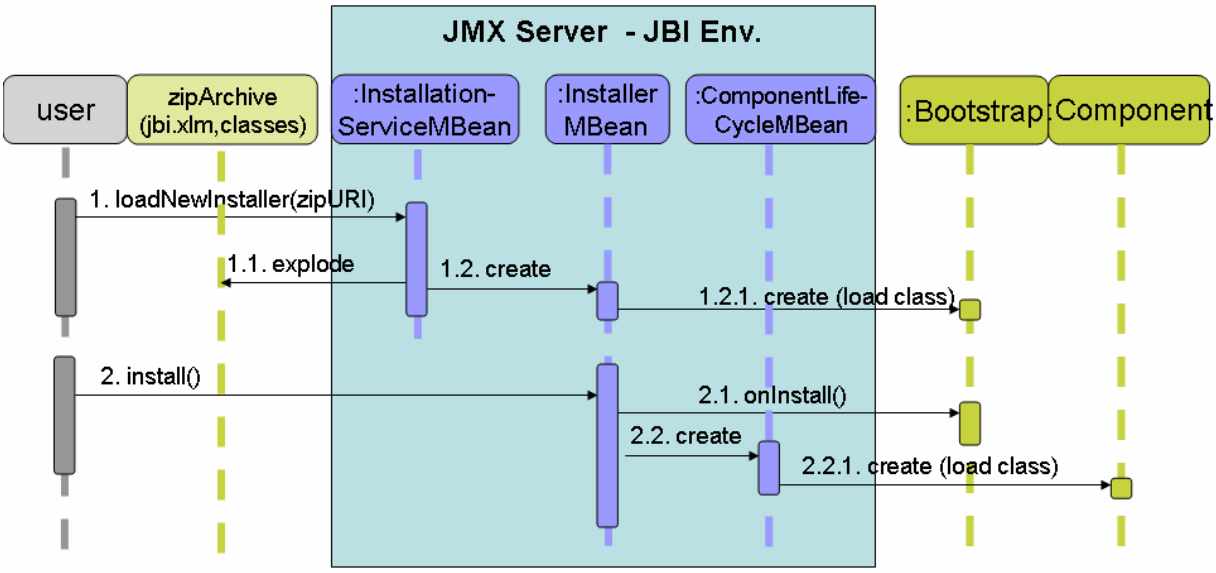

Figure 14 : install a component with JMX management

5.2 Start a component

The ComponentLifeCycleMBean manages the life cycle of the component.

The main methods are *start()*, *stop()* and *shutdown()*.

A call to the *start()* method of this MBean causes a *start()* on the corresponding component.

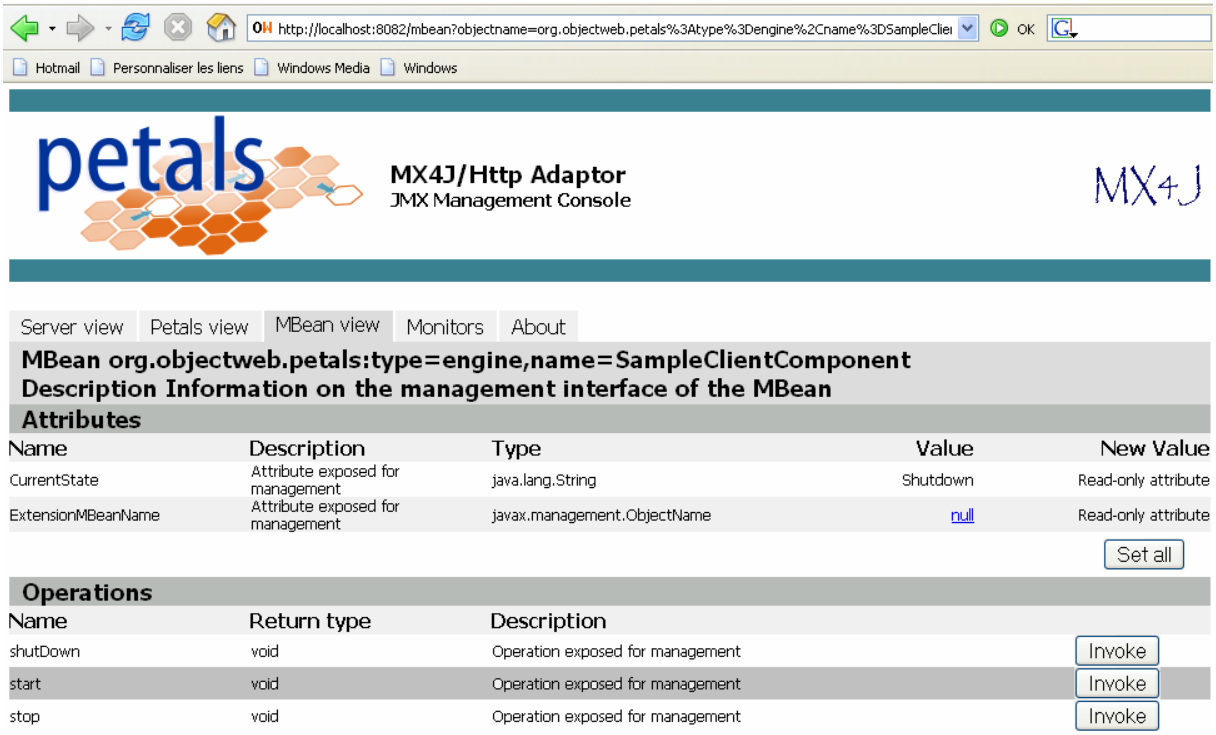

Figure 15 : LifeCycle::start

If the component is started for the first time, its *init()* method is called before the real start.

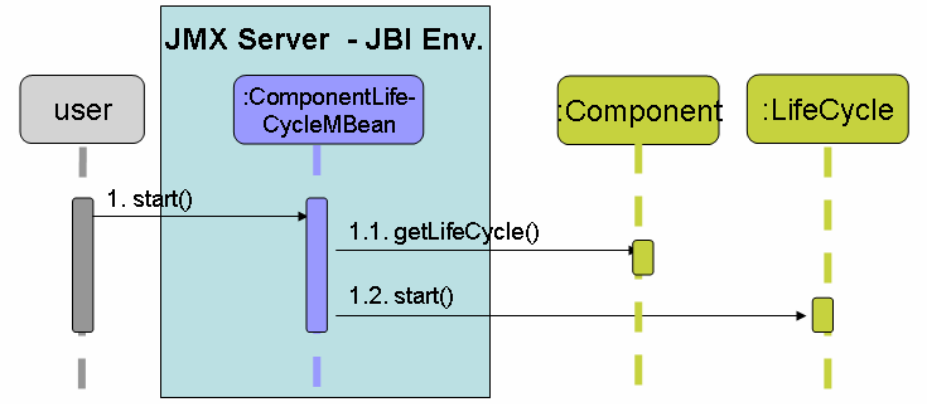

Figure 16 : start a component with JMX management

6 Conclusion

From a component point of view, using JBI and communicating with the environment is quite simple.

The use of WSDL for service description, XML for the payload of the messages and the JBI specification itself promotes the standardization of integration state of the art.

There are a lot of functionalities provided by the JBI environment that has not been discussed here, such as the WSDL definition of the provided services, synchronized or asynchronized exchanges, deployment of artefacts, use of Binding components to access external services…

The success of JBI will depend on the plethora of proposed components, either service engines that apply some integration logic to the messages, or binding components that open the JBI bus to specific protocols.

Providers of JBI containers have to propose a pertinent set of components with their container. Fortunately, as long as the components respect the JBI specification, they can be used on any JBI implementation.

7 Bibliography

[1] Java Business Integration specification http://www.jcp.org/en/jsr/detail?id=208

[2] Petals site http://petals.objectweb.org

[3] Petals gettingStarted https://wiki.objectweb.org/petals/Wiki.jsp?page=GettingStarted

[4] Java Management http://java.sun.com/products/JavaManagement