Developing a Service Engine Component

A Technical White Paper
May 2005

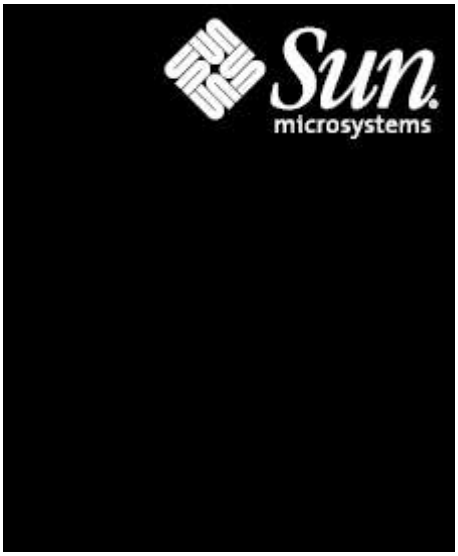


Table of Contents

Chapter 1

Document Purpose

Intent

The intention of this guide is to provide a hands-on example of the construction and deployment of a Java (TM) Business Integration (JBI) service engine component. The authors hope that this example will provide an understanding of the various elements and artifacts, along with an appreciation of the effort and process required to create a new service engine from scratch.

This guide is not intended to replace the documentation provided with the JBI Technology Preview Release , nor is it an addendum to the JBI Specification [ref 1].

Background

Before embarking on this guide, the reader should have a working understanding of JBI in general, as well as a view of the process involved and of the artifacts created when one develops a JBI component.

A JBI component can be thought of as a generic element of a JBI application, providing services and exposing business logic. These components are generally split into two camps: binding components and service engines. The first group is outside the scope of this document but can be considered to expose specific transport technologies in the form of services. For example, a SOAP binding component will expose services that are accessed by sending a SOAP XML message over HTTP(s).

The second group is the focus of this document and is usually used to expose pieces of business logic, business processes, or data transformation as services. These types of component will likely be the primary focus for bespoke development effort, based as they will be on existing services that may not be exposed with standard interfaces.

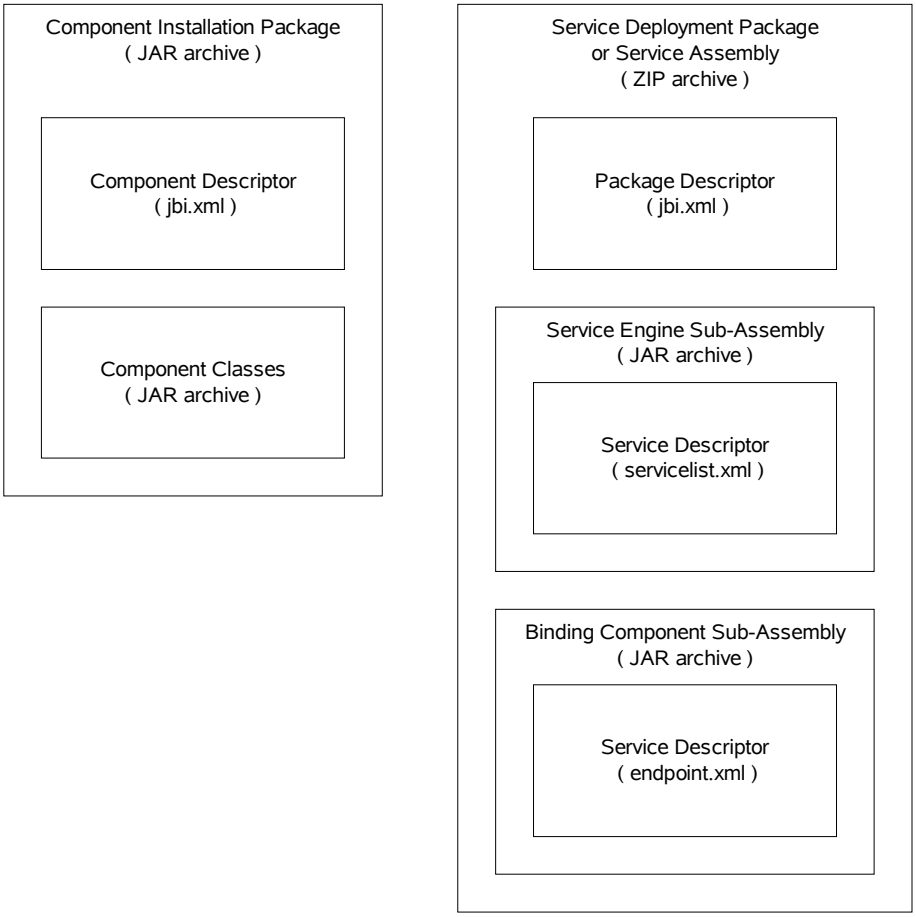
Think of the process as having three main stages: development of the component code, installation of the component in a JBI engine, and deployment of service definitions to that component.

- *Development*. The development phase is a fairly traditional Java coding exercise, with specific interfaces being implemented and Java classes being created and packaged. The end result of this phase is the core of a component—in this case, a service engine component.

- *Installation*. The installation phase requires the construction of an XML descriptor, which defines the component's identity component and describes a number of the elements within the component classes. The end result of this phase is a package (usually a JAR archive) containing the compiled component and the descriptor XML document. This package is then installed in the JBI engine.

- *Deployment*. The final phase, deployment, makes the installed component a usable JBI artifact. This phase requires constructing a number of further XML documents, both to describe the services provided and used by the installed component (usually called a subassembly) and to define the package (again, this is usually a JAR archive generally known as the service assembly) that will be deployed. The end result of this final phase will be a number of further packages: one or more subassemblies packaged together into a single service assembly.

JBI Artifacts and Their Relationships



Prerequisites

Before embarking on this guide, the reader should have the following:

- A working knowledge of the JBI Specification [ref 1]
- Java programming experience
- An installed copy of the Sun Java System Application Server, version 8.1 (PE)
- An installed copy of the JBI Technology Preview Release .

Chapter 2

Create a Service Engine Component

A number of Java classes must be created to construct a component, each based on one or more interfaces. This guide will show the absolute minimum amount of code required to build, install, and deploy a service engine component; where appropriate, it will also give pointers about adding further functionality. The reader should note that the end result of this section will *not* be a fully functional component but rather enough of a component to allow the developer to understand the effort required to create a component.

Code snippets illustrate some of the code required.

The reader should use the example applications provided with the Technology Preview release as a reference for constructing a fully operational component.

Create Service Engine Classes

Two external JAR archives are required in order to compile and build any classes created here:

- `jbi.jar` (found in `<install_dir>/appserver`)
- `j2ee.jar` (found in `JDK lib dir` or `AppServer lib dir`)

New Class Name	***Interfaces Implemented***
`TestBootstrap`	`javax.jbi.component.Bootstrap`

This class will be called by the JBI engine during component installation processes. Any application-specific behavior that is needed at install or uninstall time can be added to this class.

Each interface method must be implemented, but no code is required within each method. In order to aid debugging, however, it would be wise to produce log messages within each method, for example:

```
public void onInstall(){
System.out.println("Bootstrap instance installed.");
}
```

The method `getExtensionMBeanName()` can return a `null` reference without causing errors.

New Class Name	*Interfaces Implemented*
`TestServiceUnitManager`	`javax.jbi.component.ServiceUnitManager`

This class will be called by the JBI engine during service deployment processes. All the methods of this class will be used to deal with service unit operations: `start`, `stop`, `shutdown`, `init`, and so on. Any application-specific behavior that is needed at deploy or undeploy time can also be added to this class.

Each interface method must be implemented, but code is not required within all methods. In order to aid debugging, however, it would be wise to produce log messages within each method, for example:

```
public void start(String serviceUnitName){
System.out.println("SUM instance told to start SU "+serviceUnitName+".");
}
```

Extend the constructor to:

- Accept a parameter of type `javax.jbi.component.ComponentContext` and store this as a member variable.
- Use this variable to create another member variable of type `javax.jbi.management.ManagementMessageBuilder.`

```
public TestServiceUnitMgr(ComponentContext compCtx) {
mContext = compCtx;
mBuildManagementMessage =
mContext.getManagementMessageFactory().newBuildManagementMessage();
}
```

For the `deploy` method:

- Create an object of type `javax.jbi.management.ComponentMessageHolder` and set attributes of this object to indicate a successful deployment.
- Use the member variable of type `javax.jbi.management.ManagementMessageBuilder` to create a management message via the `buildComponentMessage` method. Ensure that the management message is returned from this method.

```
String retMsg = null;
try {
ComponentMessageHolder compMsgHolder =
new ComponentMessageHolder("STATUS_MSG");

compMsgHolder.setComponentName(mContext.getComponentName());
compMsgHolder.setTaskName("deploy");
compMsgHolder.setTaskResult("SUCCESS");

retMsg = mBuildManagementMessage.buildComponentMessage(compMsgHolder);
} catch(Exception e) {
...
}
return retMsg;
```

For the `start` method:

- The component must inform the JBI engine that it has an active endpoint.
- Create an object of type `javax.xml.namespace.QName` with values matching the service name and URI that your component will provide.
- These values *must* match those values placed in the deployment descriptor for your component
- Create an object of type `javax.jbi.servicedesc.EndpointReference` via the `activateEndpoint` method of the `ComponentContext` member variable captured in the previous constructor.
- The final parameter value *must* reference the endpoint name placed in the deployment descriptor for your component.

```
QName qn = new QName("http://www.payroll.org/payroll.wsdl","PayrollService");
try {
EndpointReference ref = mContext.activateEndpoint(qn, "SE_Endpoint");
} catch (JBIException JBIe) {
...
}
```

Important Note

Please note that the implementation suggested here is the bare minimum to enable a component to function. The values for the `activateEndpoint` method should *not* be hard-coded but extracted from the deployment package when it is deployed.

This is possible through the `deploy` method of the `TestServiceUnitMgr`, using the `ServiceUnit Path` parameter. This parameter will contain the full path for the deployment descriptor XML document `jbi.xml`, and this document can be read and parsed to extract the correct values.

The implication is that a more complex implementation is required to retrieve, store, and access these values because the `deploy` and `start` methods are called at separate points in the component lifecycle. The Technology Preview demo application provides an example of this kind of implementation..

New Class Name	*Interfaces Implemented*
`TestMsgReceiver`	`Runnable`

This class will be created not by the JBI engine but by the component class. As such, it is one way to respond to messages rather than a vital part of the component. The class will act as the receiver for all service messages from the JBI engine or Normalized Message Router (NMR).

Override the constructor to accept a parameter of type `javax.jbi.messaging.DeliveryChannel`.

Assign this parameter to a member variable.

```
public TestMsgReceiver(DeliveryChannel channel) {
this.mChannel = channel;
...
}
```

For the `run` method:

- Implement a loop that repeatedly calls the `accept` method of the `DeliveryChannel` variable. (The return type of this call should contain an `exchange`, the basis of a message exchange; add some debugging to interrogate the contents of this object.)
- Ensure that this loop checks a member variable to allow the loop to be ended.

```
public void run(){
...
mainLoop: while (this.mProcess) {
try {
this.mExchange = this.mChannel.accept(5000);

if (this.mExchange != null){
...
}
} catch (MessagingException me){
continue mainLoop;
}
}
...
}
```

Implement a `stopProcessing` method, which changes the variable checked by the loop in the `run` method.

```
public synchronized void stopProcessing(){
...
this.mProcess = false;
}
```

New Class Name	*Interfaces Implemented*
`TestComponent`	`javax.jbi.component.Component` `javax.jbi.component.ComponentLifecycle`

This class is the core of your component and will delegate the reception, processing, and returning of messages.

The component interface provides the JBI engine with methods to access parts of the service engine, for example, the `ServiceUnitManager`. The `ComponentLifecycle` interface allows the JBI engine to control the service engine, with `start`, `stop`, and `shutdown` methods. It is these methods that are called when one attempts to start or stop a component through the management tools.

Each interface method must be implemented, but code is not required within all methods. In order to aid debugging, however, it would be wise to produce log messages within each method, for example:

```
public ComponentLifeCycle getLifeCycle(){
System.out.println("Component instance asked to provide a LifeCycle.");
return this;
}
```

The method `getExtensionMBeanName()` can return a `null` reference without causing errors.

For the `init` method:
- Store the `ComponentContext` parameter in a member variable.
- Use this variable to create another member variable of type `TestServiceUnitManager`.

```
public void init(ComponentContext compCtx){
if (compCtx != null){
this.mContext = compCtx;
this.mServiceUnitMgr = new TestServiceUnitMgr(compCtx);
}
...
}
```

For the `getLifecycle` method:
- Return a reference to `this`.

For the `getServiceUnitManager` method:
- Return a reference to the member variable of type `TestServiceUnitManager`.

For the `start` method:
- Retrieve a `DeliveryChannel` object from the `ComponentContext` variable.
- Create a new instance of `TestMsgReceiver`, passing in this `DeliveryChannel` object.
- Create a new `Thread` object based on the `TestMsgReceiver` object and `start` it.

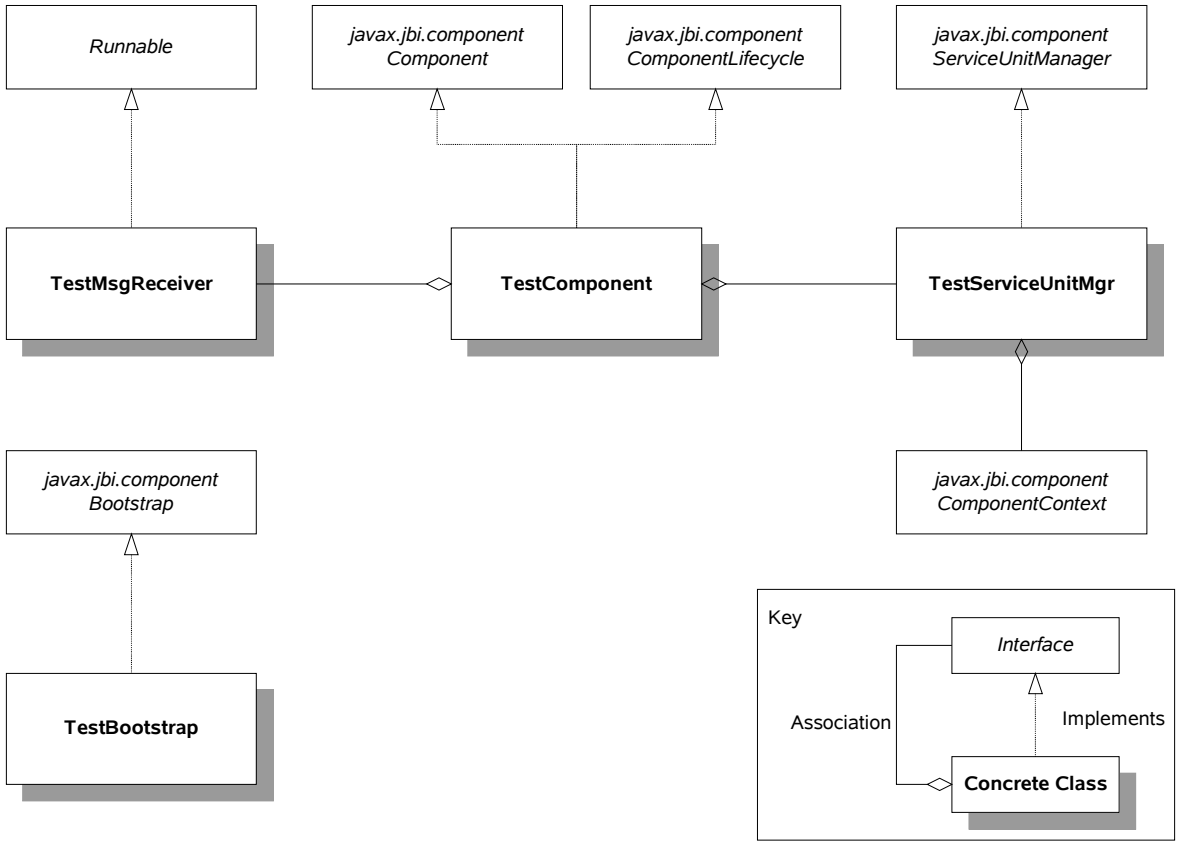
```
try {
this.mChannel = this.mContext.getDeliveryChannel();

this.mMsgReceiver = new TestMsgReceiver(this.mChannel);
Thread recThrd = new Thread(this.mMsgReceiver);
recThrd.start();
} catch (MessagingException me) {
...
}
```

For the `stop` and `shutdown` methods
- Make a call to the `stopProcessing` method of the `TestMsgReceiver` member variable.

```
public void stop(){
...
this.mMsgReceiver.stopProcessing();
}
```

Class Diagram



Chapter 3

Package Service Engine Classes

Build a single JAR archive from all these classes, referencing any other required JARs.

Create a JBI Component Descriptor

Create a new XML document called `jbi.xml` (based on the schema `jbi.xsd` found in the `<install_dir>/schemas` directory).

Place the `jbi.xml` file in a `META-INF` subdirectory.

Contents should be similar to the following:

```
<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>TestSeqEng</name>
<description>Description of your component.</description>
</identification>
<component-class-name description="[your component description]">
[your package].[your component class name]
</component-class-name>
<component-class-path>
<path-element>[your jar file path and name]</path-element>
</component-class-path>
<bootstrap-class-name>
[your package].[your bootstrap class name]
</bootstrap-class-name>
<bootstrap-class-path>
<path-element>[your jar file path and name]</path-element>
</bootstrap-class-path>
</component>
</jbi>
```

Understanding the Descriptor

The following table explains the most important elements of the `jbi.xml` document.

Element	***Description***
`jbi`	The root element, requiring the attribute `version` with a value of `1.0`.
`component`	The `type` attribute denotes whether the component will be a binding or service engine. In this case, it is set to `service engine`.
`identification`	This element block contains a unique name for thecomponent, along with a textual description.
`name`	All management tools and UIs will use this string to identify the component, so this should be meaningful but not excessively long.
`component-class-name`	This denotes the core class for the component, that is, the class that implements the component interface. The value of this element should be a *fully qualified* name of the component (for example, `com.sun.jbi.demo.TestComponent`).
`component-class-path`	This indicates the location of the class within the component deployment package. This will usually indicate a JAR archive packaged within the deployment package. Thus, the value of this element will indicate the location *within the deployment archive* of the class (for example, `dist/JBIDemo.jar`).
`bootstrap-class-name`	This denotes the class to be used at installation time, that is, the class that implements the `Bootstrap` interface. This must be the *fully qualified* name of the class.
`bootstrap-class-path`	This indicates the location of the class within the component deployment package. This will usually denote a JAR archive within the deployment package.

Package the JBI Component

Create a new JAR archive for your component and include both the JAR you created in the previous section and the `jbi.xml` document.

The resulting JAR file should have contents like this:

```
META-INF/jbi.xml
[your classes' jar file path]/[your classes' jar file].jar
```

Chapter 4

Install a Service Engine Component

Install the Component

The component package can now be installed as a JBI component, either through the NetBeans GUI or the `jbiadmin` command-line tool (see Appendix B for details).

Once the service engine package has been created, use the `jbiadmin` tool to install the component, using the following command:

```
install-component [component jar]
```

Verify the installation with the `jbiadmin` command `list-service-engines`. This should produce output similar to the following:

```
================================
List of Service Engines
================================
Name: TestSeqEng
State: Installed
--------------------------------
```

Once installed, the component can then be started to begin polling for messages, with the following command:

```
start-component [component name]
```

Verify this with the `list-service-engines` command again. The output should be similar to the following:

```
================================
List of Service Engines
================================
Name: TestSeqEng
State: Started
--------------------------------
```

You have now created and installed a service engine that will continually poll for messages.

Chapter 5

Define Services for a Component

Create a Component Subassembly

This step requires the creation of a new package for the component, which will contain details of the services that the component provides; this is called a subassembly. This package will not be deployed directly but will be used to create a superpackage or service assembly that will be deployed.

Create a new XML document named `servicelist.xml` (based on the schema `servicelist.xsd` found in the `<install_dir>/schemas` directory).

```
<service-list xmlns="http://www.sun.com/ns/jbi/engines/sequencer/deploy/service-config">
<list-attributes>
<list-service>
<namespace-uri>http://www.examples.org/service.wsdl</namespace-uri>
<local-part>SE_TestService</local-part>
</list-service>
<endpoint-name>SE_Endpoint</endpoint-name>
<list-operation>
<namespace-uri>http://www.examples.org/service.wsdl</namespace-uri>
<local-part>DoTest</local-part>
</list-operation>
<list-mep>http://www.w3.org/2004/08/wsdl/in-only</list-mep>
</list-attributes>
<service>
<service-name>
 <namespace-uri>"http://timecard.com/service.wsdl"</namespace-uri>
 <local-part>Timecard</local-part>
</service-name>
<endpoint-name>endpoint</endpoint-name>
<service-id>first_service</service-id>
<description>This defines a Service called by the SE</description>
<operation>
 <namespace-uri>"http://operation.com/service.wsdl"</namespace-uri>
 <local-part>transform</local-part>
</operation>
<timeout>5000</timeout>
<mep>http://www.w3.org/2004/08/wsdl/in-out</mep>
</service>
</service-list>
```

Understanding the Descriptor

The following table explains the most important elements of the `servicelist.xml` document.

Element	***Description***
`service-list`	This is the root element.
`list-attributes`	This section of the document describes the services that the component provides.
`list-service`	This section uniquely identifies the service.
`local-part`	This element's value denotes the name of the service; it will be shown and used both in management tools and within other descriptor files.
`endpoint-name`	This element provides a unique name for the endpoint that this service represents.
`list-operation`	This element's value details the name of the operation that this service represents.
`local-part`	This element's value denotes the name of the operation; it will be shown and used both in management tools and within other descriptor files.
`service`	This section defines those services that may be called by the component .
`service-name`	This section identifies the service that may be called.
`local-part`	This element's value denotes the name of the service; it will be shown and used both in management tools and within other descriptor files.
`endpoint-name`	This element provides a unique name for the endpoint that this service represents.
`operation`	This element's value details the name of the operation that this service represents.

Package the Subassembly

Create a new zip archive containing the `servicelist.xml` document created previously; make sure that the document is within a subdirectory named `META-INF`.

The resulting zip file should have contents like this:

```
META-INF/servicelist.xml
```

Create a Component Service Assembly

This step will create another package, containing one or more subassemblies that will be deployed to the JBI engine.

Create a new XML document named `jbi.xml` (based on the schema `jbi.xsd` found in the `<install_dir>/schemas` directory), as in the example below.

Place the `jbi.xml` file in a `META-INF` subdirectory.

```
<jbi version="1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/jbi">
<service-assembly>
<identification>
<name>JBIDemo_SA</name>
<description>Service Assembly for JBI Demo Component</description>
</identification>
<service-unit>
<identification>
<name>JBIDemoSE_SA</name>
<description>Service Engine Sub-Assembly for JBI Demo</description>
</identification>
<target>
<artifacts-zip>JBIDemoSE_SA.zip</artifacts-zip>
<component-name>TestSeqEng</component-name>
</target>
</service-unit>
</service-assembly>
</jbi>
```

Understanding the Descriptor

The following table explains the most important elements of the `jbi.xml` document.

Element	***Description***
`jbi`	The root element, requiring the attribute `version` with a value of `1.0`.
`service-assembly`	This is the main identifier for the type of package described by this file—in this case, a service assembly.
`identification`	This element block retains details of the service assembly as a whole or of each subassembly, depending on its location in the document.
`name`	This element is the identifying name for the service assembly or subassembly. The value of this element will be the identifier used within all administration tools (that is, to undeploy this service assembly).
`description`	This is a textual description of the service assembly or subassembly that is being identified (that is, that may be displayed in administrative applications).
`service-unit`	This element block identifies a subassembly, the zip archive that contains its service definitions, and the JBI component that it relates to (see also `<identification>`). This block may be repeated for each subassembly being deployed.
`target`	This element block identifies the zip archive and component to which services are being deployed.
`artifacts-zip`	The value of this element contains the fully qualified name of the zip archive containing subassembly service definitions.
`component-name`	The value of this element contains the unique identifier of the component to which this set of services are to be deployed.

Package the Service Assembly

Create a new zip archive containing the subassembly packages (zip files) you have built and the `META-INF` subdirectory.

The resulting zip file should have contents like this:

```
META-INF/jbi.xml
[your sub-assembly].zip
```

Chapter 6

Deploy Services to a Component

Deploy the Service Assembly Package

Once the service assembly package has been created, the `jbiadmin` tool can be used to deploy the package, using the following command:

```
deploy-service-assembly [service assembly.zip]
```

Verify the deployment with the `jbiadmin` command `list-service-assemblies`., which should produce output similar to the following:

```
================================
List of Service Assemblies
================================
Name: JBIDemo_SA
Status: DEPLOYED
--------------------------------
```

Chapter 7

Testing the Service Engine Component

Preparation

The following steps will provide the basis for testing a new component. This scenario is extremely simple and will serve only to prove that the new service engine component has been properly installed and has service deployed for it. For more complex or realistic examples, follow the Technology Preview examples and demo.

In essence, an XML file will be read from a directory and passed to the service engine, which will write entries to the application server log file showing that the file's contents were received.

Add a FileBinding Component

The default installation of the Technology Preview will provide a number of binding components that can be reused for testing. In this case, a file-based binding component will be used (called `SunFileBinding`); this will read a file from a given directory and pass the contents of that file to the service engine.

Because the component is already installed, a new service can be deployed to it, requiring only a new service descriptor and subassembly. This subassembly can be added to the existing service assembly.

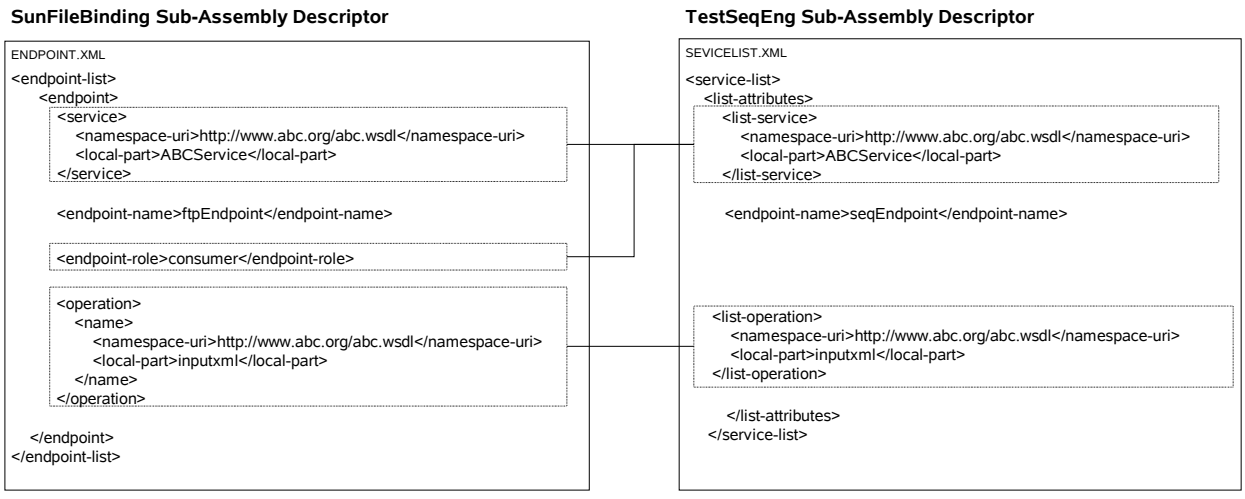
Create a New Subassembly

As before, you must create a new service decriptor file, either through the NetBeans IDE or by hand. This file will be named `endpoint.xml` and conform to the `endpoint.xsd` schema. Again, this document should be in a `META-INF` subdirectory.

This document will specify the location in which the `filebinding` component will look to read documents and any services it expects to provide or call—in this case, the service defined for our service engine component.

The values you enter into this document *must* match values entered into the component subassembly created earlier. The JBI engine (or the normalized message router) will use these values to forward a message from one component (`SunFileBinding`) to another (`TestSeqEng`).

The following diagram illustrates the relationships between elements of these two files:



Package the Subassembly

Create a new zip archive containing the `servicelist.xml` document you created previously.

Deploy New Services to the Binding Component

The existing service assembly can now be extended to include the subassembly archive just created. In addition, the `jbi.xml` descriptor must be extended to include details of the new subassembly.

The service assembly can then be redeployed (in fact, the existing service assembly will be undeployed and the new service assembly deployed in its place).

Test the Service Engine Component

The following steps should exercise both the file binding component and the new test service engine, with the contents of an XML document being passed to the service engine, which should output some debug messages to show that the exchange has occurred.

1. Place a test file in a directory for the `SunFileBinding` component to read.
2. Check the application server log file.
3. Verify that a message has been passed from one component to the other.

Appendixes

Appendix A: Useful jbiadmin commands

Command	*Description*
`List-service-engines`	Show current state of all installed service engine components.
`list-binding-components`	Show current state of all installed binding components.
`install-component [component jar]`	Install a component from the given JAR archive name.
`uninstall-component [component name]`	Uninstall a component with the given name (see JBI Component Descriptor section)
`start-component [component name]`	Start a component with the given name.
`stop-component [component name]`	Stop a component with the given name.
`deploy-service-assembly [service-assembly.zip]`	Deploy a service assembly (or assembly unit) to the JBI server.
`undeploy-service-assembly [sa name]`	Undeploy a service assembly with the given name.

Appendix B: Glossary

Term	***Description***
component	Generic name for any JBI application element.
service engine	A type of JBI component, primarily providing services such as orchestration or transformation.
binding component	A type of JBI component, primarily providing access to existing applications, via protocols such as SOAP or JMS.
install	The process by which a new component is delivered to the JBI engine.
deploy	The process by which the services provided by a component are exposed to the JBI engine (see also *service assembly*).
package	A logical grouping of elements to allow the installation or deployment of JBI artifacts.
JBI component descriptor	An XML document defining the identity and contents of a JBI component or package.
service assembly	A package of service definitions (see *subassembly*) provided to the JBI engine at deployment time (see also *deploy*.)
assembly unit	See *service assembly*.
subassembly	A package containing an XML descriptor of the services provided by or used by a specific component. See also *service assembly*.
service unit	See *subassembly*.

Appendix C: References

Reference	***Referenced Item***	***Version***	***Date***
1	Java Business Integration Specification Public Review Draft	0.75	