Building SOA-Based Composite Applications Using NetBeans IDE 6
Design, build, test, and debug service-oriented applications with ease using XML, BPEL, and Java web services

Chapter 3
"Service Engines"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter 3 “Service Engines”
A synopsis of the book’s content
Information on where to buy this book

About the Authors

David Salter is an enterprise software architect who has been developing software professionally since 1991. His relationship with Java goes right back to the beginning, using Java 1.0 for writing desktop applications and applets for interactive web sites. David has been developing Enterprise Java Applications using both the J2EE standards and open source solutions for the last five years. David runs the Java community web site Develop In Java (http://www.developinjava.com), a web site for all levels of Java developers.

Frank Jennings works in the Information Products Group of Sun Microsystems Inc. He has more than 9 years of experience in Java, SOA and System Design. He is an Electronics Engineer from Madras University and has worked for several Open Source projects. Frank has written regular columns for leading Java journals including Java Developer’s Journal and Linux Developer’s Week. Frank is also the co-author of the book SOA Approach to Integration focusing on SOA design pattern for enterprises. Frank also is involved in the technical publication of Sun Microsystems in the fields of Solaris and Developer AMP Stack. His blog can be read at http://blogs.sun.com/phantom and he can be reached at theghost@sun.com. He also holds a Post Graduate Diploma in Computer Science and an Advance Diploma in Computer Integrated Management from University of Indianapolis.

Building SOA-Based Composite Applications Using NetBeans IDE 6

Composite applications aid businesses by stitching together various componented business capabilities. In the current enterprise scenario, empowering business users to react quickly to the rapidly changing business environment is the top most priority. With the advent of composite applications the 'reuse' paradigm has moved from the technical aspect to the business aspect. You no longer re-use a service but re-use a business process. Now, enterprises can define their own behaviors optimized for their businesses through metadata and flows. This business process composition has become increasingly important for constructing business logic.

The ability of composite applications to share components between them nullifies the distinction between actual applications. Business users should be able to move between the activities they need to do without any actual awareness that they are moving from one domain to another.

The composite application design enables your company to combine multiple heterogeneous technologies into a single application, bringing key application capability within reach of your business user. Enterprises creating richer composite applications by leveraging existing interoperable components increase the development organization's ability to respond quickly and cost-effectively to emerging business requirements. While there are many vendors offering various graphical tools to create composite applications, this book will focus on OpenESB and NetBeans IDE for designing and building composite applications.

This book introduces basic SOA concepts and shows how you can use NetBeans and OpenESB tools to design and deploy a composite application. After introducing the SOA concepts, you are introduced to various NetBeans Editors and aids that you need to understand and work with to design a composite application. The last part of the book deals with a full fl edged incremental example on how you can build a complex composite application with necessary screen shots accompanied by the source code available on the website.

What This Book Covers

Chapter 1 introduces SOA and BPEL to the readers with simple examples and gives an overview of the JBI components and runtime required to build composite applications. This chapter also gives you an overview of the need for SOA-based applications in companies by depicting an example of an imaginary AirlinesAlliance system.

Chapter 2 shows you how you can quickly setup NetBeans IDE and other runtime environments including OpenESB runtime and BPEL engine. There are many software/tools mentioned in this chapter that you need to download and configure to get started building composite applications using NetBeans.

Chapter 3 provides an overview of Java Business Integration (JBI) and the Enterprise Service Bus (ESB). You will learn about JBI Service Engines and how they are supported within the NetBeans IDE.

Chapter 4 introduces JBI Binding Components and how they provide protocol independent communication between JBI components. You will also learn about the support that the NetBeans IDE provides for Binding Components.

Chapter 5 introduces the NetBeans BPEL Designer that comes bundled with the NetBeans IDE. You will also be introduced to the graphical tools/wizards and palettes available for creating BPEL files.

Chapter 6 provides an overview of WSDL and how WSDL documents are formed. You will learn about the use of WSDL in enterprise applications and the WSDL editor within the NetBeans IDE.

Chapter 7 covers the XML schema designer and shows how it aids rapid development and testing of XML schema documents.

Chapter 8 provides you an overview of the Intelligent Event Processor (IEP) module and the IEP Service Engine that can be acquired from the OpenESB software bundle. This chapter also shows the need for an event processing tool through simple composite application examples.

Chapter 9 provides details of fault handling within a BPEL process and shows how these can be managed within the NetBeans IDE by using graphical tools.

Chapter 10 shows you how you can build simple to complex composite applications and BPEL processes using the NetBeans IDE. The examples in this chapter are divided into several parts and the source code for all parts is available in the code bundle.

Chapter 11 gives you the overall picture of the composite application and the need for a composite application to deploy your BPEL processes. The composite application support provided in NetBeans IDE comes with a visual editor for adding and configuring WSDL ports and JBI modules.
In Chapter 1, we introduced the concept of SOA applications, and introduced BPEL processes and JBI applications. To gain a greater understanding of these concepts and to enable us to develop enterprise level SOA applications, we need to understand JBI in further depth, and how JBI components can be linked together. This chapter will introduce the JBI Service Engine and how it is supported within the NetBeans Enterprise Pack.

In this chapter, we will discuss the following topics:

- Need for Java Business Integration (JBI)
- Enterprise Service Bus
- Normalized Message Router
- Introduction to Service Engines
- NetBeans Support for Service Engines
- BPEL Service Engine
- Java EE Service Engine
- SQL Service Engine
- IEP Service Engine
- XSLT Service Engine

**Need for Java Business Integration (JBI)**
To have a good understanding of Service Engines (a specific type of JBI component), we need to first understand the reason for Java Business Integration.

In the business world, not all systems talk the same language. They use different protocols and different forms of communications. Legacy systems in particular can use proprietary protocols for external communication. The advent and acceptance of XML has been greatly beneficial in allowing systems to be easily integrated, but XML itself is not the complete solution.

When some systems were first developed, they were not envisioned to be able to communicate with many other systems; they were developed with closed interfaces using closed protocols. This, of course, is fine for the system developer, but makes system integration very difficult. This closed and proprietary nature of enterprise systems makes integration between enterprise applications very difficult. To allow enterprise systems to effectively communicate between each other, system integrators would use vendor-supplied APIs and data formats or agree on common exchange mechanisms between their systems. This is fine for small short term integration, but quickly becomes unproductive as the number of enterprise applications to integrate gets larger. The following figure shows the problems with traditional integration.

As we can see in the figure, each third party system that we want to integrate with uses a different protocol. As a system integrator, we potentially have to learn new technologies and new APIs for each system we wish to integrate with. If there are only two or three systems to integrate with, this is not really too much of a problem. However, the more systems we wish to integrate with, the more proprietary code we have to learn and integration with other systems quickly becomes a large problem.

To try and overcome these problems, the Enterprise Application Integration (EAI) server was introduced. This concept has an integration server acting as a central hub. The EAI server traditionally has proprietary links to third party systems, so the application integrator only has to learn one API (the EAI server vendors). With this architecture however, there are still several drawbacks. The central hub can quickly become a bottleneck, and because of the hub-and-spoke architecture, any problems at the hub are rapidly manifested at all the clients.

To help solve this problem, leading companies in the integration community (led by Sun Microsystems) proposed the Java Business Integration Specification Request (JSR 208) (Full details of the JSR can be found at http://jcp.org/en/jsr/detail?id=208). JSR 208 proposed a standard framework for business integration by providing a standard set of service provider interfaces (SPIs) to help alleviate the problems experienced with Enterprise Application Integration.

The standard framework described in JSR 208 allows pluggable components to be added into a standard architecture and provides a standard common mechanism for each of these components to communicate with each other based upon WSDL. The pluggable nature of the framework described by JSR 208 is depicted in the following figure. It shows us the concept of an Enterprise Service Bus and introduces us to the Service Engine (SE) component:

![Enterprise Service Bus Diagram](image)

JSR 208 describes a service engine as a component, which provides business logic and transformation services to other components, as well as consuming such services. SEs can integrate Java-based applications (and other resources), or applications with available Java APIs.

Service Engine is a component which provides (and consumes) business logic and transformation services to other components. There are various Service Engines available, such as the BPEL service engine for orchestrating business processes, or the Java EE service engine for consuming Java EE Web Services. We will discuss some of the more common Service Engines later in this chapter.

The Normalized Message Router

As we can see from the previous figure, SE's don't communicate directly with each other or with the clients, instead they communicate via the NMR. This is one of the key concepts of JBI, in that it promotes loose coupling of services.

So, what is NMR and what is its purpose? NMR is responsible for taking messages from clients and routing them to the appropriate Service Engines for processing. (This is not strictly true as there is another standard JBI component called the Binding Component responsible for receiving client messages. Binding Components are discussed in Chapter 4. Again, this further enhances the support for loose coupling within JBI, as Service Engines are decoupled from their transport infrastructure).

NMR is responsible for passing normalized (that is based upon WSDL) messages between JBI components. Messages typically consist of a payload and a message header which contains any other message data required for the Service Engine to understand and process the message (for example, security information). Again, we can see that this provides a loosely coupled model in which Service Engines have no prior knowledge of other Service Engines. This therefore allows the JBI architecture to be flexible, and allows different component vendors to develop standard based components.

Normalized Message Router enables technology for allowing messages to be passed between loosely coupled services such as Service Engines.

The figure below gives an overview of the message routing between a client application and two service engines, in this case the EE and SQL service engines.

In this figure, a request is made from the client to the JBI Container. This request is passed via NMR to the EE Service Engine. The EE Service Engine then makes a request to the SQL Service Engine via NMR. The SQL Service Engine returns a message to the EE Service Engine again via NMR. Finally, the message is routed back to the client through NMR and JBI framework. The important concept here is that NMR is a message routing hub not only between clients and service engines, but also for intra-communication between different service engines.

The entire architecture we have discussed is typically referred to as an Enterprise Service Bus.

Enterprise Service Bus (ESB) is a standard-based middleware architecture that allows pluggable components to communicate with each other via a messaging subsystem.

Now that we have a basic understanding of what a Service Engine is, how communication is made between application clients and Service Engines, and between Service Engines themselves, let’s take a look at what support the NetBeans IDE gives us for interacting with Service Engines.

**Service Engine Life Cycle**

Each Service Engine can exist in one of a set of predefined states. This is called the Service Engine life cycle.

- Started
- Stopped
- Shutdown
- Uninstalled

The figure below gives an overview of the life cycle of Service Engines:

Service Engines can be managed from the command line utility, `asadmin`, that is supplied as part of the Sun Java System Application Server. The table below shows some of the common commands that can be used to manage Service Engines:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>asadmin list-jbi-service-engines</code></td>
<td>Obtains a list of installed Service Engines</td>
</tr>
<tr>
<td><code>asadmin show-jbi-service-engine</code></td>
<td>Shows the status of an installed Service Engine</td>
</tr>
<tr>
<td><code>$&gt;./asadmin show-jbi-service-engine sun-bpel-engine</code></td>
<td>Component Information: Name: sun-bpel-engine, State: Shutdown, Description: This is a bpel service engine.</td>
</tr>
<tr>
<td><code>asadmin start-jbi-component</code></td>
<td>Starts a Service Engine</td>
</tr>
<tr>
<td><code>$&gt;./asadmin start-jbi-component sun-bpel-engine</code></td>
<td>Started component sun-bpel-engine.</td>
</tr>
<tr>
<td><code>asadmin stop-jbi-component</code></td>
<td>Stops a Service Engine</td>
</tr>
<tr>
<td><code>$&gt;./asadmin stop-jbi-component sun-bpel-engine</code></td>
<td>Stopped component sun-bpel-engine.</td>
</tr>
</tbody>
</table>

Service Engines can also be managed from within the NetBeans IDE instead of using the `asadmin` application. We will look at that in the next section.

**Service Engines in NetBeans**

As we discussed in Chapter 2, the NetBeans Enterprise Pack provides a version of the Sun Java System Application Server 9.0 which includes several Service Engines from the Open ESB project.

All of these Service Engines can be administered from within the NetBeans IDE from the **Services** explorer panel. Within this panel, expand the **Servers | Sun Java System Application Server 9 | JBI | Service Engines** node to get a complete list of Service Engines deployed to the server.
Service Engines

The NetBeans Enterprise Pack 5.5 and the NetBeans 6.0 IDE have different Service Engines installed. The following table lists which Service Engines are installed in which version of the NetBeans Enterprise Pack:

<table>
<thead>
<tr>
<th>Service Engine Name</th>
<th>Description</th>
<th>NetBeans 6.0</th>
<th>NetBeans 5.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>sun-aspect-engine</td>
<td>Aspect Service Engine</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>sun-bpel-engine</td>
<td>BPEL Service Engine</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>sun-dtel-engine</td>
<td>DTEL Service Engine</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>sun- etl-engine</td>
<td>ETL (Extract, Transform and Load) Service Engine</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>sun-iep-engine</td>
<td>IEP (Intelligent Event Processor) Service Engine</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>sun-javaee-engine</td>
<td>Java EE Service Engine</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>sun-script-engine</td>
<td>Scripting Service Engine</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>sun-sql-engine</td>
<td>SQL Service Engine</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>sun-wlm-engine</td>
<td>WLM (Work List Manager) Service Engine</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>sun-xslt-engine</td>
<td>XSLT Service Engine</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

In the previous section, we discussed the life cycle of Service Engines and how this can be managed using the `asadmin` application. Using the NetBeans IDE, it is easy to manage the state of a Service Engine. Right-clicking on any of the Service Engines within the Services explorer shows a menu allowing the life cycle to be managed as shown in the figure below:

To illustrate the different states in a Service Engine life cycle, a different icon is displayed:

Now that we have a good understanding of what Service Engines are, and what support the NetBeans IDE provides, let's take a closer look at some of the more common Service Engines provided with the NetBeans Enterprise Pack.

**BPEL Service Engine**

Similar to all the other Service Engines deployed to the JBI Container within the Sun Java System Application Server and accessible through NetBeans, the BPEL Service Engine is a standard JBI Compliant component as defined by JSR 208.

The BPEL Service Engine enables orchestration of WS-BPEL 2.0 business processes. This enables a work flow of different business services to be built as shown in the following figure:

---

Within NetBeans, we can create BPEL modules which consist of one or more BPEL processes. BPEL modules are built into standard JBI component, and then deployed to the JBI container where the BPEL Service Engine allows the processes within the module to be executed. In JBI terms, this is called a Service Unit.

A Service Unit is a deployable component (jar file) that can be deployed to a Service Engine.

New BPEL modules are created in NetBeans by selecting the File | New Project menu option and then selecting BPEL Module from the SOA category as shown in the following figure:

Within a BPEL module project, we add BPEL Processes. These processes describe the orchestration of different services.

All the standard operations specified by WS-BPEL 2.0 Specification (like Providing and Consuming Web Services, Structuring the processing logic, and performing basic activities such as assignments and waiting) are available within the BPEL Service Engine. The NetBeans designer provides simple drag-and-drop support for all of these activities.

Consider, for example, a service for generating license keys for a piece of software. In a Service Oriented Architecture, our system may consist of two services:

1. A Customer Service: this service would be responsible for ensuring that license requests are only made by valid customers.
2. A License Generation Service: this service would be responsible for generating valid license keys.

Within NetBeans, we can create a BPEL process that ties these services together allowing us to return valid license keys to our customers and details of purchasing options to non-customers.
Java EE Service Engine

The Java EE service engine acts as a bridge between the JBI container allowing Java EE web services to be consumed from within JBI components. Without the Java EE service Engine, JBI components would have to execute Java EE Web Services via remote calls instead of via in-process communication. The Java EE Service Engine allows both servlet and EJB-based web services to be consumed from within JBI components.

The Java EE Service Engine provides several benefits when executing Java EE Web Services.

- Increased performance
- Transaction support
- Security support

These are explained in the following subsections.

Increased Performance
Using the Java EE service engine enables Java EE web services to be invoked in process within the same JVM, as the services are running. This eliminates the need for any wire-based transmission protocols and provides increased performance.

Transaction Support
Using an in-process Communication Model between Java EE Application Server and JBI container allows both web services and JBI modules to use the same transaction model. Through multiple web service calls and calls to other JBI modules. For example, BPEL processes can all use the same transaction.

Security Support
When executing Java EE Web Services from within the JBI container, the Java EE Service Engine allows security contexts to propagate between components. This removes the need to authenticate against each service.

SQL Service Engine
SQL service engine allows SQL statements to be executed against relational databases and allows the results of SQL statements to be returned to the client application or other Service Engines for further processing.

SQL service engine allows SQL DDL (Data Definition Language), SQL DML (Data Manipulation Language), and stored procedures to be executed against a database. This, therefore, allows different scenarios to be executed against the database. For example, obtaining a customer's address or the number of outstanding invoices a customer may have.

Within NetBeans, the SQL module is used to interact with the SQL Service Engine. The SQL module project consists of three artifacts as follows:

- configuration xml file (connectivityInfo.xml)
- one or more SQL files containing distinct SQL statements
- WSDL file describing the SQL operations.

SQL Modules are created by choosing File | New Project and then selecting the SQL Module option from within the SOA projects category.

Within a SQL Module, there is a configuration file called connectivityInfo.xml which contains connection details for the database. This can either be specified as a driver connection or as a JNDI name for a data source.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<connection>
  <database-url value='jdbc:derby://localhost:1527/db_name'/>
  <jndi-name value=''/>
</connection>
```

Each SQL statement that is to be presented to client applications as a new operation must be stored in a separate SQL file. Using the example scenarios above, we would have two SQL files with contents shown in the following table:

<table>
<thead>
<tr>
<th>SQL File</th>
<th>SQL Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer_address.sql</td>
<td>select address1, address2, zip from customer where customer_id=?</td>
</tr>
<tr>
<td>outstanding_invoices.sql</td>
<td>select count(*) from invoices where customer_id=? and isPayed='n'</td>
</tr>
</tbody>
</table>

In order for other JBI components to be able to access our SQL module, we must have a WSDL file which describes the operations we have defined (\texttt{customer_address.sql} and \texttt{outstanding_invoices.sql}). NetBeans will generate this file for us when we select the \textbf{Generate WSDL} option from right-clicking on the project in the \texttt{Projects} explorer.

SQL Service assembly units cannot be executed directly from within the JBI container. To execute the SQL Service Unit, it needs to be added as part of a composite application. This is then called a Service Assembly. Composite applications are further discussed in Chapter 4.

\textit{Service Assembly}: a deployable component (jar file) that consists of a collection of Service Units.

\section*{IEP Service Engine}

The Intelligent Event Processing service engine allows data to be read from an input source and then processed into a format that can be used for a variety of different purposes such as reporting or business intelligence information.

For example, an IEP project could be created that takes sales information from a retail system, collects all information made over the last hour, and then outputs it to a database table for reporting purposes. This would enable fast reporting based upon a periodically updated subset of the business data. Any reporting queries performed would therefore be "off-line" to the business database. This way different reporting queries could be performed as and when necessary without any performance impact on the business database.

Depending on the version of NetBeans that you have installed, you may not automatically have support for creating and editing IEP projects. If you do not have IEP project support within NetBeans, both the IEP service engine and NetBeans editor support for IEP projects can be downloaded from http://www.glassfishwiki.org/jbiwiki/attach/IEPSE/InstallationGuide.zip.

New IEP modules can be created within NetBeans by selecting the **File | New Project** menu option and then selecting the **Intelligent Event Processing Module** option within the **SOA** category as shown in the following figure:

![New Project](image)

After making the above selections, the second stage of the **New Project** wizard allows the **Project Name** and the **Project Location** to be specified.

Finally, after creating the new IEP module, new Intelligent Event Processors can be added to the project. This is achieved by right-clicking on the newly created IEP project within the NetBeans **Project** pane and selecting the **New | Intelligent Event Processor** menu option. Selecting this option displays the **New Intelligent Event Processor** wizard which includes one page allowing the IEP **File Name** and **Folder** to be specified.

Service Engines

The IEP Process Editor within NetBeans allows many different processing actions to be performed on data. IEP Processes are defined using a drag-and-drop editor. The Palette, which shows all of the operations that can be performed on data, is shown in the following figure:

IEP Processes (Service Assemblies) cannot be executed directly from within the JBI container. To execute IEP Processes, they need to be deployed into a Service Assembly and added as part of a composite application. Composite applications are further discussed in Chapter 4.

XSLT Service Engine

XSLT Service Engine enables transformations of XML documents from one format to another using XSL stylesheets. The service engine allows XSL transformations to be deployed as web services which can then be used by external clients.

New XSLT modules can be built to run against the XSLT service engine by selecting the File | New Project menu option and then selecting the XSLT Module option from within the SOA category as shown in the following figure:

Several different types of files can be created within an XSLT Module to allow the service engine to transform XML files from one format to another. XML Schema files can be used to define XML within the transformation process. WSDL files are used to define the operations that are transformed within the service engine. We won’t discuss how WSDL files and XML Schema files are created and maintained in this chapter, however, we will discuss them in full detail later in this book.

The final type of file that can be specified within an XSLT Module is an XSLT Service. These types of files can be created by right-clicking on the XSLT Module within the Project explorer in NetBeans and selecting the New | XSLT Service menu option. The result is shown in the next screenshot.

When creating an XSLT Service Unit, two different processing modes (Service type) are available:

- Request-Reply Service
- Service Bridge

Service Engines

The **Request-Reply Service** mode enables an XML message to be received from a client, transformed, and then sent back to the original client.

The **Service Bridge** mode enables an XML message to be received from a client and transformed into a different format. The transformed message is then used as an input for invoking a service. The output of this service is then transformed using a second XSL stylesheet and returned to the original caller. The **Service Bridge** mode is therefore acting as a bridge between two services. This is an implementation of the Adapter Pattern as defined in Design Patterns—Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

When creating a **Request-Reply Service**, the **New XSLT Service** wizard allows the web service for the XSLT transformation to be specified including details of the port, the operation being executed and the input and output types of the operation as shown in the following two screenshots:

Chapter 3

Service Engines

When creating a Service Bridge service, the two web services to be bridged are specified by first selecting the WSDL for the implemented web service and then for the invoked web service.

Having selected the web services to bridge, the wizard allows the implemented and invoked web services to be fully specified. Here we need to specify the operation from our implemented service and the operation to call on the invoked service.
Summary

In this chapter, we have introduced the concept of a Service Engine and given an overview of the Service Engines installed with the NetBeans Enterprise Pack (the BPEL, Java EE, SQL, IEP, and XSLT Service Engines). We've learned that Service Engines:

- provide business logic functionality to their clients
- can be consumers and/or providers
- run within a Java Business Integration (JBI) Server
- expose their interfaces via WSDL
- communicate within an Enterprise Service Bus via messaging

We've also discussed some basic concepts about JBI such as the Normalized Message Router, Service Assemblies, and Service Units. We have a good understanding of JBI, some of the problems with Enterprise Application integration and why JBI is useful. In the next chapter, we extend our knowledge of JBI and SOA application development with NetBeans by describing another standard JBI component—the binding component.
Where to buy this book


Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.