5

Enabling Enterprise-Class Web Services

In the last chapter, we talked about the business requirements and the business reasons for choosing a web services architecture for the solution. Before you start designing the web services to meet these business requirements, you have to take a close look at the factors to be considered for enabling enterprise-class web services. Enterprise-class web services represent a level of technical and operational maturity that is consistent with the increasingly stringent requirements of today's enterprise. As we progress toward building such web services, in this chapter, we take a close look at designing interoperability, publishing an enduring web services contract, exposing the business-tier components, and planning for a robust production environment.

The WebLogic Workshop web services framework abstracts the developer from the plumbing code required to build enterprise-class web services. WebLogic Workshop combines the visual development environment with the runtime framework to help developers and nondevelopers effectively and efficiently build these services.

Designing Interoperability

The two dominant web services development platforms are on J2EE and .NET. Most often, the client accessing the web services cannot be predetermined. Therefore, it is important to make certain design choices—choice of standards, WSDL design, and SOAP messaging style—that enhance the interoperability of the web services implementation across different platforms.

Conforming to Standards

Using the web services standards—XML, WSDL, SOAP, and UDDI, as well as WS-Security (security), Web Services Distributed Management (manageability), and the Business Process Execution Language (orchestration)—strictly as defined by the standard bodies is a necessary

requirement for interoperability. Following the recommendation from bodies such as the WS-I not only improve interoperability, but it also facilitates fast implementation with business partners.

Standards are important at all levels of web services communication. By using a standard way of exchanging data using XML, choosing a common SOAP message protocol, and describing services through WSDL, you enable the users of your service to communicate with your application regardless of the platform they are using. This approach helps you to integrate your web services with those of your business partners without relying on proprietary vendor technology.

In WebLogic Workshop, exposing an operation through these standards is extremely easy. Workshop introduces a simple javadoc-based annotation `@jws:operation` for marking up standard Java code. If you simply put this annotation before a method in a standard Java class file, Workshop exposes that method as a web service that fully supports the web services standards.

Designing WSDL First

In general, designing the WSDL first is the right way to start your web services. The WSDL represents the contract between you and the user of your service. Not unlike the real world, the contract needs to be well crafted. It needs to spell out in the clearest possible terms what the service offers—and, ideally, what it does not do.

Most often, developers seldom design the WSDL because many SOAP implementations provide tools to automatically generate a WSDL file from a COM object or Java class files. Surprisingly, this can cause issues of interoperability between different Java application servers and between .NET and Java implementations. This is because the automatically generated WSDLs might not be interpretable across different platforms. For example, some application servers cannot use WSDLs that have imported or included XML Schemas. Designing the WSDL first and then creating the web service avoids the issue.

The recommendation is that you generate WSDL using techniques based on the Unified Modeling Language (UML) or using WSDL editors (see Figure 5.1). We cover in detail how to design WSDL using XMLSPY, a WSDL editor, in the next chapter.

WSDL modeling in UML can be another way of creating WSDL rather than using WSDL editors. The Object Management Group (OMG) has established a Model Driven Architecture (MDA) and Unified Modeling Language (UML) for modeling software applications. UML is a graphical language for specifying, visualizing, constructing, and documenting software systems. Because of its wide acceptance and capabilities, UML is the natural choice for any modeling activities. UML profiles provide a generic extension mechanism for building UML models in particular domains such as Web Modeling Profile, XSD Schema Profile, and Business Profile Modeling. A *profile* consists of stereotypes, constraints, and tagged values that allow modeling of all WSDL definitions.

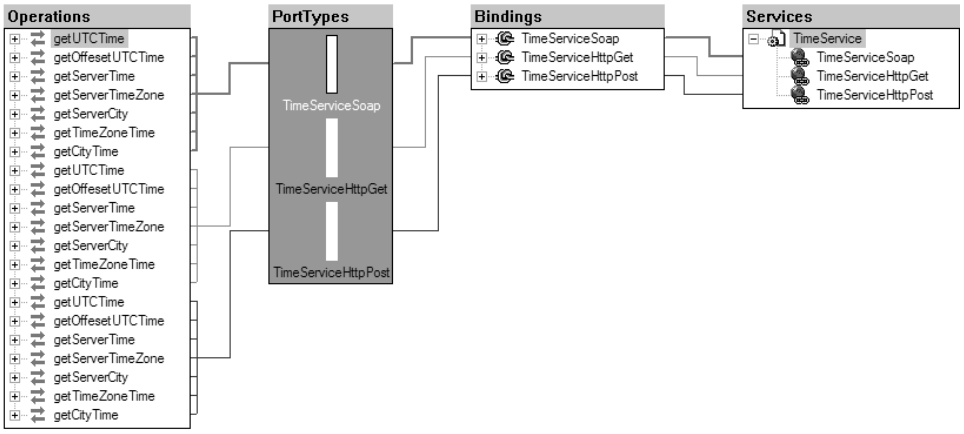

Figure 5.1 XMLSPY, a sample WSDL editor

Deriving from the MDA, the modelling of the WSDL can be divided into a platform-independent model (PIM) and a platform-specific model (PSM). A WSDL file has two sections, the abstract section and the bindings section. The PIM represents the abstract section of the WSDL. This includes `Definitions`, `Service`, `PortType(s)`, `Messages`, `Parts`, and `PartType(s)`. The PSM completes the bindings section of the WSDL. The elements modelled here include `Service`, `Ports`, and `Binding(s)`.

Selecting SOAP Messaging

The style, or binding style, decision controls how the elements just under the SOAP body are constructed. The two choices are RPC and document. The use, or encoding, concerns how types are represented in XML. The two choices are literal and SOAP encoded. Literal follows an XML Schema definition. SOAP encoded follows special encoding rules detailed in the SOAP 1.1 specification to produce XML that can contain references pointing within the message. The WS-I Basic Profile Version 1.0a specification recommends the use of literal encoding; it actually discourages the use of SOAP encoding.

Theoretically, you should be able to mix and match any of the styles and uses for SOAP messaging. However, practically, the RPC-encoded and document-literal choices are the ones used. In addition, many web service platforms don't directly support RPC-literal. WebLogic Workshop uses document-literal as its default message format, but it also supports the RPC-encoded message format. To specify the message format that a web service uses to communicate with the web service control, you can set the `soap-style` attribute of the `@jc:protocol` annotation to `document` for the document-literal message format or `rpc` for the SOAP-RPC format.

RPC-encoded seemed reasonable in a world without XML Schema. Now that XML Schema is here to stay, document-literal is quickly becoming the de facto standard among developers and toolkits, thanks to its simplicity and interoperability results.

Styles: Document and RPC

Document style indicates that the SOAP body contains an XML document. In this approach, you the send XML document as is. The sender and receiver must agree on the format of the document ahead of time. Here is an example of a document-style message:

```
<SOAP-ENV: Envelope xmlns:SOAP-
➥ENV=http://schemas.xmlsoap.org/soap/envelope/
➥xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
➥mlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
➥xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <SOAP-ENV:Body>
        <m:inValidateConfig xmlns:m="http://production.psg.hp.com/types">
            <m:Config>
                <m:Model/>
                <m:Cstic/>
            </m:Config>
            <m:User AccountID=0000000000>
                <m:Password>String</m:Password>
            <m:IpcVersion>OLD</m:IpcVersion>
            <m:ConfigID>String</m:ConfigID>
        </m:inValidateConfig>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Observe that all elements are defined within the same namespace. With a document-style web service, you are free to define the XML however you want because all the XML is explicitly specified.

RPC style indicates that the SOAP body contains an XML representation of a method call. RPC style uses the names of the method and its parameters to generate structures that represent a method's call stack. Here's an example of a RPC-style message:

```
    <SOAP-ENV:Body>
        <m:validate xmlns:m="local
➥host:7001/services/validate" SOAP-ENV:encodingStyle=
➥"http://schemas.xmlsoap.org/soap/encoding/">
            <in0 xsi:type="xsd:string">String</in0>
        </m:validate>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

RPC relies on some rules. Observe that the root element is qualified with a namespace, but the child element uses an unqualified name. The root element is named after the operation: `validate` for a SOAP request and `opNameResponse` for a SOAP response. Each child is a parameter, or a return value named `opNameReturn`.

The binding style is specified in the WSDL. There actually is a style attribute that can be `RPC` or `Document` in the `<soapbind:binding>` element, which is the `<wsdl:binding>` element's first child. Table 5.1 illustrates the differences between `RPC` and `Document`.

Table 5.1 Binding Style in a WSDL

	RPC	**Document**
`<soapbind:binding>` element	`style="rpc"`	`style="document"`
`<wsdl:part>` element(s)	`<part>` element can be many; each contains a "type" attribute.	Only one `<part>` element containing an "element" attribute.
`<wsdl:part>` element(s) explained	Specify the type so that the SOAP engine can use RPC rules to make it either a parameter or a return value.	The SOAP body has an XML element that is fully specified in `<wsdl:types>`.

Use: Literal and Encoding

The use, or encoding, concerns how types are represented in XML. The two choices are literal and SOAP encoded. Literal follows an XML Schema definition. SOAP encoded follows special encoding rules detailed in the SOAP 1.1 specification (section 5) to produce XML that can contain references pointing within the message. Now to compare the literal and encoded use, let's compare the example from the document style that also uses the literal with another example showing the encoding use.

Here's the literal use:

```
<m:User AccountID=0000000000>
        <m:Password>xyz </m:Password>
<m:User>
```

Here's the encoding use:

```
< AccountID xsi:type ="xsd:string">0000000000</ AccountID r>
<Password xsi:type="xsd:string">xyz</Password>
```

SOAP encoding automatically maps all eligible fields to elements, not attributes. With the literal use of an XML Schema, on the other hand, you can specify what items are attributes and what items are elements. In the previous example, the simple `AccountID` field can be written more compactly as an attribute.

Now we can examine how use is specified in the WSDL. Table 5.2 shows the difference in literal versus encoded in the WSDL document.

Table 5.2 Use/Encoding Specified in a WSDL

	Encoded	Literal
`<soapbind:body>` use attribute	`use="encoded"`	`use="literal"`
Other `<soapbind:body>` attributes	`encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"`	—

Publishing Enduring Web Services Contracts

A web service contract, usually expressed via a WSDL, describes the relationship between the web services that you publish and the manner in which an external client can interact with them. These touch upon what XML messages to expect, what business operations you can perform, and what XML messages you generate. You can do so without letting the other party know about the technical details of your application and how they change over time. By the same token, you can use other developers' public contracts to consume their applications without needing to know the details or even the language and platform they use to implement their web service.

These contracts can be made more durable by following certain principles, such as loose coupling, XML strategy selection, and versioning.

Integrating Through Loose Coupling

An important aspect of web services design is to ensure that the web services are loosely coupled. Web services are loosely coupled when you separate the internal implementation from the interface exposed to the user of your service. The web service has an interface defined by the WSDL that is exposed to the outside world. The internal implementation of a web service is the specific application server that it runs on or the EJBs that do the business logic. If you need to change your application server, the user of the service is not affected because there is a change only to the internal implementation. When this is ensured, the user of the web service does not need to get involved in testing changes unless the interface changes as defined by the WSDL. You might still want to test with the client even if the WSDL doesn't change, to make sure that SLAs and performance requirements can still be met.

Using the WSDL, however, is not enough to guarantee loose coupling. If your WSDL is automatically generated from the code and you change the code, the WSDL changes, thus making it tightly coupled to the code. WebLogic Workshop provides XQuery Maps which is a simple, declarative way of describing how your Java code relates to the WSDL of your web service. If your Java code changes over time, you can change your XQuery Map so that your WSDL stays intact. XQuery Map maps the Java code to your WSDL. When the Java code changes, the mapping can be alerted to keep the WSDL the same. This enables you to truly realize the promise of loose coupling. WebLogic also provides the facility of modeling the WSDL in XMLSpy and then importing it into WebLogic Workshop. With this WSDL, you can create a web service in WebLogic Workshop. See Figures 5.2 and 5.3 for an illustration of the modeling and importing of WSDL.

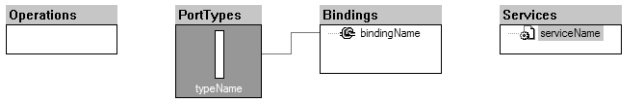

Figure 5.2 Modeling of WSDL

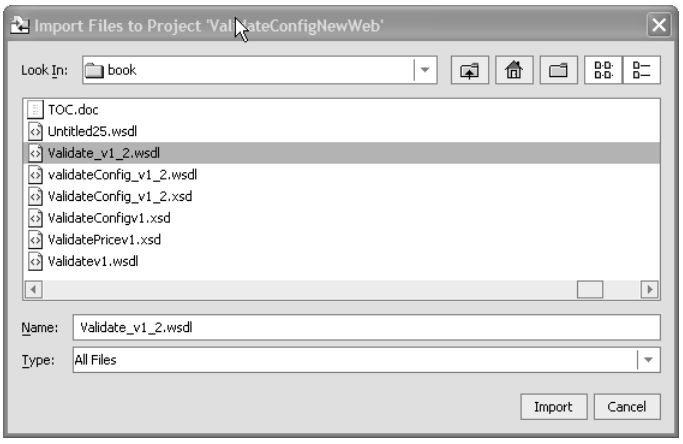

Figure 5.3 Importing of WSDL in WebLogic Workshop

Here is an example of using an XQuery map in WebLogic Workshop. The start and end of the XML map is shown by the annotation:

```
  * @jws:operation
  * @jws:parameter-xml xml-map::
  *   < order>
  * <item xm:multiple="String name in nameA,
➥int amount in amountA, float price in priceA">
  *       <name>{name}</name>
  *       <amount>{amount}</amount>
  *       <price>{price}</price>
  *     </item>
  *   </order>
  * ::
  * @jws:return-xml xml-map::
  *   <totalPrice>{return}</totalPrice>
  * ::
  */
  public float getTotalPrice(String [] nameA,
➥int [] amountA, float [] priceA)
  {
      float totalPrice = 0.0f;
      if( nameArr != null )
      {
          // for each item, compute subtotal and add to total
         for (int i = 0; i < nameArr.length; i++)
          {
               totalPrice += amountArr[i] * priceArr[i];
          }
       }
       return totalPrice;
  }
```

This map specifies that the `getTotalPrice` Java method receives an XML document that contains an order with multiple line items, each with a name, amount, and price. These fields are extracted from the XML message and mapped into Java arrays. Similarly, the float value that is returned from the method is placed in the context of a simple XML document that has a `<totalPrice>` tag.

Choosing an XML Strategy

XML is the messaging standard for web services. You have to choose the right strategy for handling XML messages in web services, depending on how the web services you are designing fit into the overall application.

The XML strategy that you will use depends on two criteria. One is whether you have any application logic and whether you are exposing the application logic. The second criterion is whether the interface to the web service you are developing has a predefined XML Schema and whether you want to define your own.

This can lead to four different strategies for handling XML Schemas:

- **Defined application logic and predefined XML Schema**—In this case, the strategy that you use is mapping between defined incoming messages to fields in your internal data types. XQuery maps in WebLogic Workshop can be used to map the data.
- **No internal classes and predefined XML Schema**—If schemas are defined in the WSDL file, you should import the WSDL in WebLogic Workshop and let XML Beans accept the schema file and return a set of Java classes that a developer can conveniently leverage to process any XML document that conforms to the original schema file. Having schema definitions at the core of the XML Beans system provides a variety of benefits. For example, when you first receive an XML document for processing, you can validate the data based on the schema definition. Any time you manipulate the XML document via the schema-inspired Java classes, the XML Beans system can always ensure that all changes remain consistent with the prevailing schema definition. This unambiguously disallows the creation of invalid XML documents.
- **When internal classes are defined and the schema is not defined, you should define your schema according to your internal classes.** If both are not defined, first define the schemas and then derive the Java classes from there. This follows the concept of designing WSDL first, as described in the earlier section.

Versioning New Releases

You need to consider versioning of your web service to facilitate managing multiple versions of a web service. Versioning should minimize code replication and maximize code reuse. It should also put a logical and manageable naming paradigm in place. This allows for upgrades and improvements to be made to existing web services, while continuously supporting previously released versions of that web service.

Two areas should be considered for versioning a web service: the public interface, as described by the WSDL file, and the web service implementation, including its conversational state.

Versioning the Public Interface

You can version the public interface—specifically, your WSDL—in different ways:

- Use a different endpoint for your service:

```
<wsdl:service name="Validate_v1_2">
        <wsdl:port name="Validate" binding="tns1:ValidateSoapBinding">
            <wsdlsoap:address location="localhost:7001/ValidateConfigNewWeb/
➥ValidateConfigNew/validate_v1_2ControlTest.jws"/>
        </wsdl:port>
    </wsdl:service>
```

In this example, you can change to the next version `validate_v1_3` by changing the address of the endpoint for the service as follows:

```
<wsdlsoap:address location="localhost:7001/ValidateConfigNewWeb
➥/ValidateConfigNew/validatev_1_3ControlTest.jws"/>
```

In this approach, the XML Schema does not change and there is no change in your SOAP message. The advantage in this option is that you can insulate your users from changing schemas. The drawback is that there is no reference to the version `validate_v1_2` within the SOAP message, so you cannot use management tools that can direct the message to the right version:

```
<SOAP-ENV:Body>
        <m:inValidateConfig xmlns:m="http://production.psg.hp.com/types">
            ...
    </SOAP-ENV:Body>
```

- Use a date stamp as part of the target namespace of your XML Schema. This is in compliance with the W3C XML Schema specification. The disadvantage here is that your schema changes every time the version changes. The advantage of this approach is that you can see the version in your SOAP messages, and you can write code or use management tools to direct the web service to the right version, according to the SOAP message:

```
<SOAP-ENV:Body>
        <m:inValidateConfigv1_2
➥xmlns:m="http://production.psg.hp.com/types/2004/02/04">
            ...
    </SOAP-ENV:Body>
```

- Add new operations to the WSDL and support old ones until the user moves to the new operation. You can add new operations to the public interface of a web service to reflect the new versions, keeping the existing operations. The advantage of doing so is that you do not disrupt clients that rely on the web service. By adding new operations and making them known to clients, you can gradually shift clients over to a new set of operations, but you should leave the original operations intact for backward compatibility. For example, the `operation` section of the WSDL will look as shown here:

```
<wsdl:operation name="validateConfig">
         <wsdlsoap:operation soapAction="validate_v1_2"/>
         <wsdl:input>
              <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output>
              <wsdlsoap:body use="literal"/>
         </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="validatePrice">
         <wsdlsoap:operation soapAction=" validate_v1_3 "/>
         <output>
              <wsdlsoap:body use="literal"/>
         </output>
         <input>
              <wsdlsoap:body use="literal"/>
         </input>
    </wsdl:operation>
```

The `types` section of the WSDL will look as shown here:

```
<wsdl:types>
         <xsd:schema targetNamespace="http://production.psg.hp.com/types"
➥xmlns="http://production.psg.hp.com/types" elementFormDefault="qualified">
              <xsd:include schemaLocation="ValidateConfigv1_2.xsd"/>
              <xsd:include schemaLocation="ValidateConfigv1_3.xsd"/>
         </xsd:schema>
    </wsdl:types>
```

The SOAP message for each of the operations will look as shown next. Again, you can use management tools to direct messages to different versions:

```
<SOAP-ENV:Body>
        <m:inValidateConfigv1_2 xmlns:m="http://production.psg.hp.com/types">
            ...
    </SOAP-ENV:Body>
<SOAP-ENV:Body>
        <m:inValidateConfigv1_3 xmlns:m="http://production.psg.hp.com/types">
            ...
    </SOAP-ENV:Body>
```

- Use UDDI and versioning UDDI in conjunction with WSDL to provide for versioning. The UDDI data model is rich enough that the current best practices can be enhanced to include service versioning. A given service can advertise more than one interface that represents its different versions. Different interfaces have different tModels. The service can reference the tModel for each of the interface in its `tModelInstanceDetails` collection. `tModelInstanceDetails` is a class that has `tModelKey` as a mandatory attribute.

Versioning the Implementation

WebLogic Workshop uses Java serialization to persist the state associated with web service requests and conversations. For this reason, the primary requirement for supporting versioned implementations is maintaining backward serialization compatibility. Specifically, it must be possible to load state into the current version of a class that was stored using any older versions of that class.

Versioning Lifecycle

Understanding the versioning lifecycle will help you to implement the new versions of your web service and deprecate the older versions effectively. First, you need to put together a plan for the lifecycle of the web services you are supporting. The main aspects of the plan of the versioning lifecycle are listed here:

1. Your plan should contain the frequency of the release of your versions. For example, you could release one version per year. Consider how many versions of the web service you want to support in parallel.
2. The plan should also contain the time frame for your users to move to the new version. This would be the same as the time you would support an older version after the new one is released.
3. Consider using a pilot for the new version of the web services with an early release of version.

4. Consider releasing new functionality and conformance to new web service specifications only through this versioning strategy, as with software releases.
5. Communicate the versioning strategy to the users of your web service.

After you lay out the versioning strategy for each version, follow these steps to release each version of your web service:

1. Make changes to the services that you are supporting, as detailed in the previous sections.
2. Do unit and functional testing of the service.
3. Deploy the new service either through new WSDLs for users of the service or to UDDI registries.
4. Notify the consumers of your new service and pilot the new service with one of your consumers.
5. Run the new and old versions in parallel for the time frame you have allocated in your versioning plan.
6. Notify the consumers of the date when the old service is obsoleted.
7. Remove the old service version from descriptions, registries, and so on to keep new consumers from discovering and using the old web service. Remove the functional behavior of the old service; return only an appropriate error message.
8. Retire the old service. Physically remove the old service version.

Effectively Using Business-Tier Systems

Web services are usually built on existing business-tier systems. To effectively interface with these systems, you have to take into consideration the communication, frequency of interaction, and degree of exposure to the business components. For example, a purchase order web service must effectively interface with the underlying ERP systems.

Facilitating Asynchronous Communication

Web services can be designed to be synchronous or asynchronous in nature. In synchronous mode, a web service implements a call-and-response RPC mode of interaction. Here, when the service is invoked, the business logic behind the web service executes the logic while the client is waiting for a response. For example, this can be used in getting price quotes and stock quotes.

For complex processes such as orders and order changes, processing might take minutes or even days to complete. This is particularly true when the web service implementation depends on batch processing or manual steps that require human intervention. This is an asynchronous

web service. The client does not wait for the response to do the next steps. It either checks back periodically to see if the process is completed on the web service producer side, or acts as a listener to any updates.

Web services standards include the infrastructure and mechanisms on which asynchronous operations can be based. Handling web services interactions asynchronously is useful to accommodate user interactions, legacy IT systems, and partner IT systems. In these instances, systems must not block one another, and relevant systems and data must be easy to connect. This is also essential for scalability.

To effectively build composite, asynchronous applications, Workshop provides a model for web service conversations. A *conversation* is a series of message exchanges in both directions that are related and share some context. The Workshop framework automatically manages a unique ID for each conversation, and messages in a conversation are related through a conversation ID. In addition, any class member variables are automatically persisted and available later as the conversation continues. To use conversations in Workshop, methods must be annotated as starting, continuing, or finishing a conversation. To start a conversation, you can use the following tag:

```
* @jws:operation
* @jws:conversation phase="start"
```

SOAP conversation is a SOAP- and WSDL-based specification that defines long-running and asynchronous interactions between SOAP-based senders and receivers.Workshop web service conversations enable you to easily build asynchronous web services without having to write the underlying infrastructure typically associated with these types of applications (see Figure 5.4).

Figure 5.4 Asynchronous web service

Using a Coarsely Grained Approach

Much of the design of a web service interface involves designing the service's operations. After you determine the service's operations, you define the parameters for these operations, their return values, and any errors or exceptions that they can generate. That is, you define the method signatures of the service.

You should define the web service's interface for optimal granularity of its operations. Although finely grained service operations, such as an operation to browse a catalog by categories, products, or items, offer greater flexibility to the client, they also result in greater network overhead and reduced performance. More coarsely grained service operations, such as returning catalog entries in a set of categories, reduce network overhead and improve performance, although they are less flexible. Generally, you should consolidate finely grained operations into more coarsely grained ones to minimize expensive remote method calls. The key benefits of coarsely grained web services are that they provide a uniform interface, reduce coupling, increase manageability and reusability, centralize security management and transaction control, and improve performance.

With WebLogic Workshop, the XML documents that you exchange are business-level documents, such as entire invoices or purchase orders. The coarsely grained web services can access finely grained business logic through the control framework of WebLogic Workshop. You can have web services, such as purchase orders, that access finely grained controls for business logic such as EJB controls and database controls.

Exposing Business Logic Components

The Model-View-Controller (MVC) architecture is one of the well-known architecture frameworks used in many web-based applications. The MVC design is modular, separating the key components of the architecture:

- **Model**—Contains data required by the application
- **View**—Manages the presentation of the data to the user
- **Controller**—Acts as an intermediary between the client and the data

The MVC approach (see Figure 5.5) can allow an organization to expose multiple interfaces to the same set of data. For example, consider a web service in which EJBs provide the data from a database (Model), JSPs are used to present the data (View), and servlets mange the interface between the client and the data (Controller). Now, if the client device changes from a browser to a PDA, the View can be changed from HTML to WML without affecting the EJBs and servlets. Also, if web services can be considered as just another View on the model, a web services presentation can easily be plugged into an existing MVC architecture. In this case, instead of HTML or WML being sent back to the client application, the web services View would construct XML or SOAP messages through interactions by interacting with the Controller.

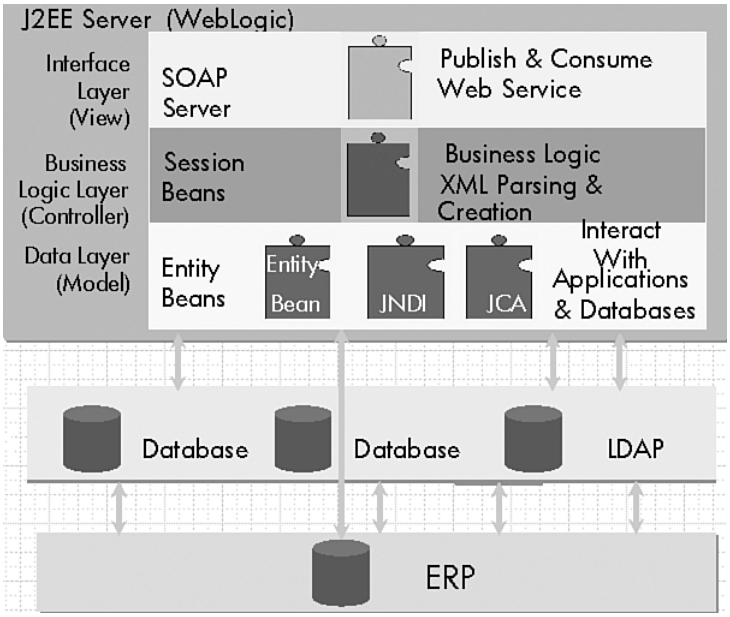

Figure 5.5 MVC architecture

A Java class or an EJB can perform the business logic for fulfilling a web service request. The EJBs that support web services are session- and message-driven beans. Stateless session beans facilitate RPC-style web services that result in component operation invocations. Message-driven beans facilitate document-oriented web services. They do asynchronous messaging through a JMS consumer. Java classes can also be used for the business logic and are simpler to implement.

WebLogic Workshop has a Controls Framework with many built-in controls, such as database, EJB, and web service, that can wrap the back-end components and help expose them as web services. We talk more about the Controls Framework in Chapter 8, "Using Controls, Bindings, and Parsers."

The Web Flow Framework is a visual representation of presentation scenarios in WebLogic Workshop. The web flow framework is based on Struts, which is based on the MVC design paradigm. WebLogic Workshop facilitates the Web Flow Framework, which extends the Struts framework to provide a simplified development model with numerous additional features.

Companies such as HP are using frameworks such as SSA, which is described in the accompanying sidebar to build well-designed robust, scalable, and extensible web services. SSA complements standard J2EE web service toolkits such as WebLogic Workshop and helps HP IT in creating a consistent, repeatable approach to developing web services in J2EE.

Shared Services Architecture

The Shared Services Architecture (SSA) is a framework based on J2EE standards that was developed at HP to meet developers' needs for reusable services. The first version of the SSA was released in the year 2000, and support for creating web services was added in 2002. The SSA was developed by and is maintained by a small team within HP IT; it is the corporate standard framework for all internal Java-based web services development.

As we have discussed in this chapter, there is more to programming of web services, in production, you need modular, reusable, high-performance web services. This is where the SSA framework codifies the use of best practices, including design patterns, into a ready-to-use architecture and offers a robust, flexible framework for implementing enterprise-class web services.

The SSA offers developers a set of higher-level abstractions such as Request, Result, Feature, and Business Policy that developers can start with instead of having to start with a clean slate for every new service that they implement.

The SSA offers help in several areas:

- Designing and developing a layered service implementation
- Developing a modular, reliable, scalable, high-performance service
- Designing and developing a modular, easily repurposable user experience implementation
- Enforcing business rules in an easy-to-change fashion
- Dealing with error handling and metrics gathering
- Providing scalability and availability, ERP/legacy integration, authentication, and authorization
- Specifying and controlling the overall flow and calling service components at predefined points in time using predefined interfaces. Conceptually, this is very similar to what other frameworks such as Struts offer for building web UIs in Java.

The SSA architecture complements the frameworks, describes the overall structure of a shared service, and provides a design blueprint for organizing a service, including the logical layers that a service is recommended to have, and the responsibilities of those layers.

continues

As complements of the core SSA framework itself, there are several plug-ins, utilities, and tools. The plug-ins expand the base functionality available in SSA and are analogous to web browser plug-ins. The list of available plug-ins includes the Apache Axis and the WebLogic SOAP plug-in for exposing SSA services as web services with different web services toolkits. The utilities include APIs for logging, metrics gathering, and lifecycle management. The tools include development tools such as code-generation wizards and deployment tools such as scripts and Ant build files.

Several key business benefits are associated with using SSA, including faster time to market, reduced total cost of ownership (TCO), reduced support costs, improved developer productivity, and improved capability to react to change. The SSA framework presently is being used in HP's identity-management solution for external users, HP Passport. It allows all external users to HP to be registered through web services built with the SSA framework in the BEA WebLogic plug-in. The SSA framework has helped make this solution robust and scaleable. It has high performance and is very flexible in adding operations to the existing web services.

Planning a Robust Production Environment

Enterprise-class web services require application servers to provide clustering, caching, transaction coordination, messaging, high-performance XML processing, database connection pooling, and other features that have made J2EE the de facto standard for enterprise computing. WebLogic Workshop is built on top of WebLogic Server. As a developer, you can automatically leverage much of the robustness of the WebLogic Server platform (see Figure 5.6).

In addition to leveraging the robust foundation provided by WebLogic server, you need to pay particular attention to security, scalability, peformance, and manageability to ensure a robust production environment.

Security

Securing the web services that you build is an important aspect of building enterprise-class web services. You can use traditional techniques such as two-way SSL. However, there is performance degradation using two-way SSL because the whole message will be encrypted even though only parts of it might need to be encrypted. For example, you might want to encrypt just the customer name, address, and credit-card information in a purchase order, not the purchase order items and their descriptions.

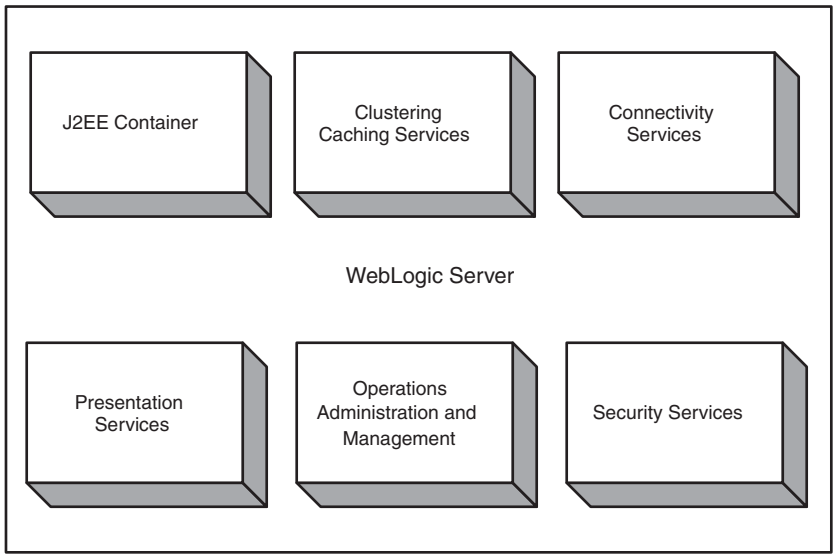

Figure 5.6 WebLogic Server

WS-Security secures the SOAP message. The OASIS Technical Committee has ratified a specification with a standard set of SOAP extensions that can be used when building secure web services to implement message content integrity and confidentiality.

WS-Security is controlled using policy files. One part of a WS-Security policy determines the security requirements for SOAP messages coming into a web service or web service control. This part of the policy determines the security mechanisms for an inbound SOAP message to pass the security gate. The other part of a WS-Security policy determines the security enhancements to be added to outgoing SOAP messages before they are sent out to the client. This part of the policy file determines the kinds of security mechanisms that a web service or a web service control adds to SOAP messages.

In WebLogic Workshop, WS-Security policies are configured in WSSE files, an XML file with the `.wsse` extension. The `<wsSecurityIn>` element describes the security requirements for incoming SOAP messages; the `<wsSecurityOut>` element describes the security enhancements added to outgoing SOAP messages.

To apply a WS-Security policy to a web service, add the annotations `@jws:ws-security-service` and `@jws:ws-security-callback` to the web service file. If the web service communicates synchronously with its clients, you need to use only the `@jws:ws-security-service` annotation. If the web service sends callbacks to its clients, you must use both annotations. The Web Service CallBack Protocol (WS-CallBack) specification consists of the `CallBack` SOAP header and an associated WSDL definition. WS-CallBack is used to dynamically specify where to send asynchronous responses to a SOAP request:

```
/**
 * @jws:ws-security-service file="MyWebServicePolicy.wsse"
 * @jws:ws-security-callback file="MyWebServicePolicy.wsse"
 */
public class MyWebService implements com.bea.jws.WebService
```

Scalability and Performance

Enterprise-class web services need to be scalable and to perform at an optimum level. WebLogic Server provides clustering, caching, transaction coordination, messaging, high-performance XML processing, and database connection pooling, which makes the web services running on it highly scalable. You can improve your performance by tuning the WebLogic Server and by using JRockit instead of Sun JVM as your JVM (covered in detail in Chapter 12, "Enhancing Performance of Web Services").

Manageability

Management of web services is extremely important in an enterprise-class implementation. You need to establish the way you want to manage your web services while designing your web services. This is crucial to the success of business operations. Applications today are increasingly distributed, complex, and integral to the enterprise. For example, fault analysis now often involves gathering and reporting information across many applications, services, and platforms. As applications have grown more complex, so have their manageability needs.

BEA WebLogic with HP's OpenView uses Java/J2EE management via JMX. To manage the web services, WebLogic Server leverages Java Management Extensions (JMX). JMX defines a standard for developing managed beans (MBeans). The first step is to develop the MBeans as a wrapper to your Java web service that you want to manage and monitor. Then you need to register this MBean in the WebLogic JMX Server. JMX Server is responsible for decoupling management applications from the managed resources. It is important to note that MBeans are always accessed via the methods of the MBeanServer interface.

HP enhances the WebLogic Server management world by providing a WebLogic Server–specific Smart Plug-In (SPI) module that integrates the management of the WebLogic Server and any associated applications that depend on it. This SPI integrates several sources of information into one screen. Simultaneously, it gathers data from any supplied MBeans, performs calculations on that data to be more meaningful to the end operator, and brings filtered application logging messages to bear on the management task

This provides highly desirable management integration. Many IT operations departments want to monitor the health of their computers, networks, application servers and any applications that depend on them from the same console and using the same familiar tools.

Web services manageability goes beyond Java/J2EE management via JMX. In practice, this means that IT operations can deploy and configure your application, monitor its health and performance, predict and reduce failure, and analyze, correct, and report failures. At the business level, this means that your enterprise users can do more than just monitor: They can control and adapt your web service based on management data received.

The importance of a standardized management model for web services has driven industry leaders in web services and management technologies and applications to form a technical committee in OASIS called the Web Services Distributed Management (WSDM) Technical Committee.

Web services manageability is defined by this committee as a set of capabilities for discovering the existence, availability, health, performance, and usage (as well as the control and configuration) of a web service within the web services architecture. This implies that web services can be managed using web services technologies.

Web Services Networking

Functions such as load balancing, failover, routing, and monitoring and access control can also be done by an intermediary (see Figure 5.7). This unburdens each producing and consuming entity of the web service. This intermediary can also do the function of translation, as with messaging protocols, formats of XML messages, and security standards. They can also manage versioning of web services by directing consumers of the web service to different versions. All web service messages are routed first to the intermediary, where network policy is enforced. This class of solutions is called web services networking. Blue Titan and Grand Central are two vendors that offer such a capability. We cover web services manageability in detail in Chapter 14, "Managing Web Services."

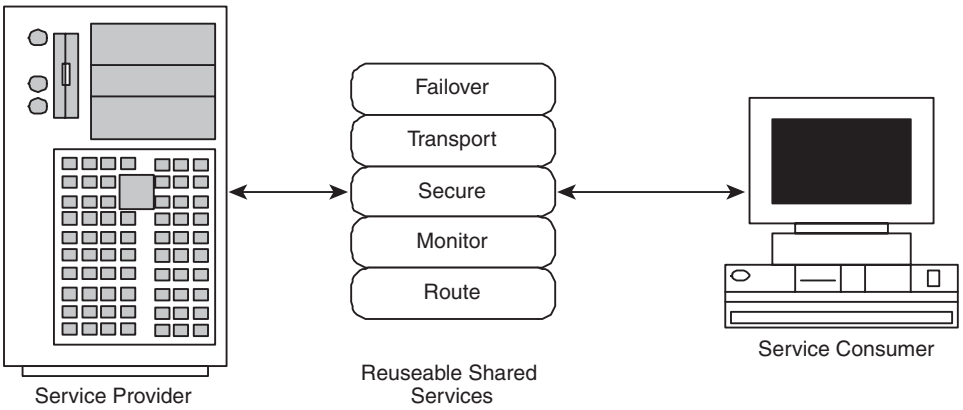

Figure 5.7 Web services networks

Summary

In this chapter, we focused on the considerations for enabling enterprise-class web services. These are web services that meet the level of technical and operational maturity consistent with the increasingly stringent requirements of today's enterprise.

J2EE web services must be designed for interoperability. This requires conforming to the specifications of XML, SOAP, WSDL, UDDI, and WS-I interoperability recommendations; designing the WSDL before implementing SOAP; and using document-literal as the SOAP messaging format.

Web services are published via WSDL. These represent contracts between the consumer and the producer of web services. To make these contracts durable, it is important to integrate using loose coupling, choose the right XML strategy, and use either versioning of the public interface or versioning of the implementation.

Web services are typically layered on top of enterprise business systems. To effectively interface with these business systems, we recommend using asynchronous communication using Workshop's web service conversations, using WebLogic's control framework to develop coarsely grained web services for reduced network overhead and improved performance, and using WebLogic's control framework and the MVC architecture to wrap the back-end components and help expose them as web services.

Security, scalability, and manageability considerations are crucial for enterprise-class web services. WS-Security secures SOAP messages and can be implemented using WebLogic Workshop through WS-Security policy files. Leverage the robust foundation of the WebLogic Server and improve performance by fine-tuning the WebLogic Server and using JRockit as the JVM. Use BEA WebLogic along with HP OpenView to manage web services. The former leverages JMX for developing MBeans, used for managing and monitoring web services. The latter offers the WebLogic Server–specific Smart Plug-In module for enhanced management capabilities. Intermediaries called Web Services Networks offload load balancing, failover, routing, monitoring, and access control from the web services infrastructure.