CHAPTER 3

Service Endpoint Design

WEB services interact with clients to receive the clients' requests and return responses. In between the request and the response, a Web service applies appropriate business logic to process and fulfill a client's request. Designing an effective Web service starts with understanding the nature of the service to be provided—that is, how the service is going to interact with the client and how it is going to process and fulfill the client's request—coupled with how users of the service find and make their requests. This chapter examines Web services from the perspective of a service's interaction and processing functionality.

The chapter describes the key issues you must consider when designing a Web service, then shows how these considerations drive the design and implementation of a service's Web service interface and functionality. In particular, the chapter examines the interactions between a service and its clients and the business processing that the service performs. It illustrates these considerations by drawing from examples using three typical Web service scenarios.

The chapter covers most of the decisions that must be made when designing and implementing a Web service, including identifying the different possibilities that give rise to different solutions. It describes how to receive requests, delegate requests to business logic, formulate responses, publish a Web service, and handle document-based interactions.

Along the way, the chapter makes recommendations and offers some guidelines for designing a Web service. These recommendations and key points, marked with check boxes, include discussions of justifications and trade-offs. They are illustrated with the example service scenarios. Since Web services basically expose interoperable interfaces for new as well as existing applications, a large segment of the audience of this book may have existing applications for

which they have already implemented the business logic. For that reason, and since the primary interest of most readers is on Web services, this chapter keeps its focus on Web service development and does not delve into the details of designing and implementing business logic.

3.1 Example Scenarios

Let's revisit the scenarios introduced in "Typical Web Service Scenarios" on page 11—the adventure builder enterprise scenario and the examples illustrating when Web services work well for an enterprise—from the point of view of designing a Web service. This chapter, rather than discussing design issues abstractly, expands these typical scenarios to illustrate important design issues and to keep the discussion in proper perspective.

In this chapter, we focus on three types of Web services:

1. An informational Web service serving data that is more often read than updated—clients read the information much more than they might update it. In our adventure builder example, a good scenario is a Web service that provides interested clients with travel-related information, such as weather forecasts, for a given city.
2. A Web service that concurrently completes client requests while dealing with a high proportion of shared data that is updated frequently and hence requires heavy use of EIS or database transactions. The airline reservation system partner to adventure builder is a good example of this type of Web service. Many clients can simultaneously send details of desired airline reservations, and the Web service concurrently handles and conducts these reservations.
3. A business process Web service whose processing of a client request includes starting a series of long-running business and workflow processes. Adventure builder enterprise's decision to build a service interface to partner travel agencies is a good example of this type of Web service. Through this service interface, partner agencies can offer their customers the same services offered in adventure builder's Web site. The partner agencies use adventure builder's business logic to fulfill their customer orders. A service such as this receives the details of a travel plan request from a partner agency, and then the service initiates a series of processes to reserve airlines, hotels, rental cars, and so forth for the specified dates.

Discussions of Web service design issues in this chapter include references to these examples and scenarios. However, the discussions use only appropriate characteristics of these scenarios as they pertain to a particular design issue, and they are not meant to represent a complete design of a scenario.

3.2 Flow of a Web Service Call

In a Web service scenario, a client makes a request to a particular Web service, such as asking for the weather at a certain location, and the service, after processing the request, sends a response to the client to fulfill the request. When both the client and the Web service are implemented in a Java environment, the client makes the call to the service by invoking a Java method, along with setting up and passing the required parameters, and receives as the response the result of the method invocation.

To help you understand the context within which you design Web services, let's first take a high-level view at what happens beneath the hood in a typical Web services implementation in a Java environment. Figure 3.1 shows how a Java client communicates with a Java Web service on the J2EE 1.4 platform.

Note: Figure 3.1 changes when a non-Java client interacts with a Java Web service. In such a case, the right side of the figure, which reflects the actions of the Web service, stays the same as depicted here, but the left side of the figure would reflect the actions of the client platform. When a Java client invokes a Web service that is on a non-Java platform, the right side of the figure changes to reflect the Web service platform and the left side, which reflects the actions of the client, remains as shown in the figure.

Once the client knows how to access the service, the client makes a request to the service by invoking a Java method, which is passed with its parameters to the client-side JAX-RPC runtime. With the method call, the client is actually invoking an operation on the service. These operations represent the different services of interest to clients. The JAX-RPC runtime maps the Java types to standard XML types and forms a SOAP message that encapsulates the method call and parameters. The runtime then passes the SOAP message through the SOAP handlers, if there are any, and then to the server-side service port.

The client's request reaches the service through a port, since a port provides access over a specific protocol and data format at a network endpoint consisting of a host name and port number.

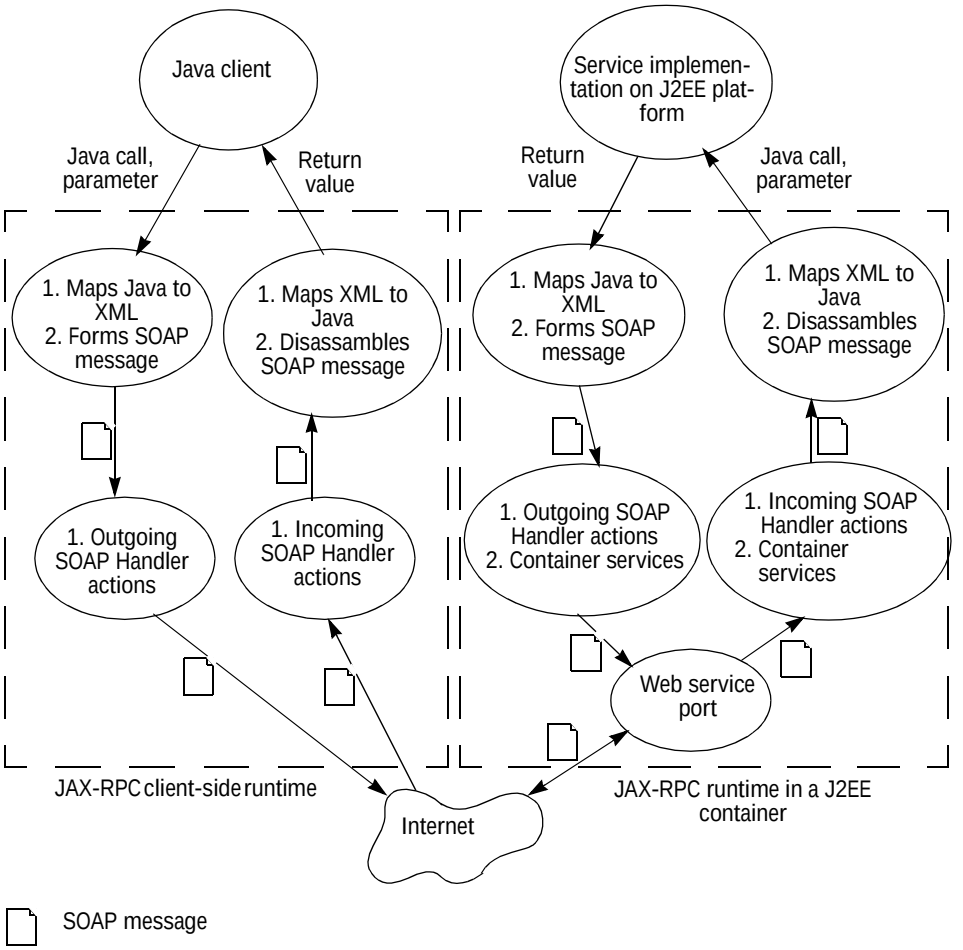

Figure 3.1 Flow of a Web Service on Java Platform

Before the port passes the request to the endpoint, it ensures that the J2EE container applies its declarative services (such as security checks) to the SOAP request. After that, any developer-written SOAP handlers in place are applied to the request. Note that SOAP handlers, which are optional, let developers apply application-specific processing logic common to all requests and responses that flow through this endpoint. After the handlers operate on the SOAP message, the message is passed to the service endpoint.

The J2EE container extracts the method call invoked by the client along with the parameters for the call, performs any XML-to-Java object mapping necessary, and hands the method to the Web service interface implementation for further processing. A similar set of steps happens when the service sends back its response.

Note: All the details between the method invocation and response just described happen under the hood. The platform shields the developer from these details. Instead, the developer deals only with typical Java programming language semantics, such as Java method calls, Java types, and so forth.

3.3 Key Web Services Design Decisions

Now that you understand what happens in a Web service interaction, let us look further at the issues involved in the design and implementation of a Web service. We first look at what goes into designing a Web service, examining the issues for which decisions are required and, when possible, making recommendations. (Similarly, Chapter 5 examines the issues to consider when designing a Web service client.) Before doing so, it is worthwhile to repeat this point:

- Web service technologies basically help you expose an interoperable interface for a new or an existing application. That is, you can add a Web service interface to an existing application to make it interoperable with other applications, or you can develop a completely new application that is interoperable from its inception.

It is important to keep in mind that designing Web service capabilities for an application is separate from designing the business logic of the application. In fact, you design the business logic of an application without regard to whether the application has a Web service interface. To put it another way, the application's business logic design is the same regardless of whether or not the application has a Web service interface. When you design a Web service interface for an application, you must consider those issues that pertain specifically to interoperability and Web services—and not to the business logic—and you make your design decisions based on these issues.

When designing a Web service, consider the logic flow for typical Web services and the issues they address. In general, a Web service:

- Exposes an interface that clients use to make requests to the service
- Makes a service available to partners and interested clients by publishing the service details
- Receives requests from clients
- Delegates received requests to appropriate business logic and processes the requests
- Formulates and sends a response for the request

Given this flow of logic, the following are the typical steps for designing a Web service.

1. **Decide on the interface for clients. Decide whether and how to publish this interface.**

   You as the Web service developer start the design process by deciding on the interface your service makes public to clients. The interface should reflect the type and nature of the calls that clients will make to use the service. You should consider the type of endpoints you want to use—EJB service endpoints or JAX-RPC service endpoints—and when to use them. You must also decide whether you are going to use SOAP handlers. Last, but not least, since one reason for adding a Web service interface is to achieve interoperability, you must ensure that your design decisions do not affect the interoperability of the service as a whole.

   Next, you decide whether you want to publish the service interface, and, if so, how to publish it. Publishing a service makes it available to clients. You can restrict the service's availability to clients you have personally notified about the service, or you can make your service completely public and register it with a public registry. Note that it is not mandatory for you to publish details of your service, especially when you design your service for trusted partners and do not want to let others know about your service. Keep in mind, too, that restricting service details to trusted partners does not by itself automatically ensure security. Effectively, you are making known the details about your service and its access only to partners rather than the general public.

2. **Determine how to receive and preprocess requests.**

   Once you've decided on the interface and, if needed, how to make it available, you are ready to consider how to receive requests from clients. You need to design your service to not only receive a call that a client has made, but also to

do any necessary preprocessing to the request—such as translating the request content to an internal format—before applying the service's business logic.

3. **Determine how to delegate the request to business logic.**

   Once a request has been received and preprocessed, then you are ready to delegate it to the service's business logic.

4. **Decide how to process the request.**

   Next, the service processes a request. If the service offers a Web service interface to existing business logic, then the work for this step may simply be to determine how the existing business logic interfaces can be used to handle the Web service's requests.

5. **Determine how to formulate and send the response.**

   Last, you must design how the service formulates and sends a response back to the client. It's best to keep these operations logically together. There are other considerations to be taken into account before sending the response to the client.

6. **Determine how to report problems.**

   Since Web services are not immune from errors, you must decide how to throw or otherwise handle exceptions or errors that occur. You need to address such issues as whether to throw service-specific exceptions or whether to let the underlying system throw system-specific exceptions. You must also formulate a plan for recovering from exceptions in those situations that require recovery.

After considering these steps, start designing your Web service by devising suitable answers to these questions:

- How will clients make use of your services? Consider what sort of calls clients may make and what might be the parameters of those calls.
- How will your Web service receive client requests? Consider what kind of endpoints you are going to use for your Web service.
- What kind of common preprocessing, such as transformations, translations, and logging, needs to be done?
- How will the request be delegated to business logic?
- How will the response be formed and sent back?

- What kinds of exceptions will the service throw back to the clients, and when will this happen?
- How are you going to let clients know about your Web service? Are you going to publish your service in public registries, in private registries, or some way other than registries?

Before exploring the details of these design issues, let's look at a service from a high level. Essentially, a service implementation can be seen as having two layers: an interaction and a processing layer. (See Figure 3.2.)

❒ It is helpful to view a service in terms of layers: an interaction layer and a processing layer.

The service interaction layer consists of the endpoint interface that the service exposes to clients and through which it receives client requests. The interaction layer also includes the logic for how the service delegates the requests to business logic and formulates responses. When it receives requests from clients, the interaction layer performs any required preprocessing before delegating requests to the business logic. When the business logic processing completes, the interaction layer sends back the response to the client. The interaction layer may have additional responsibilities for those scenarios where the service expects to receive XML documents from clients but the business logic deals with objects. In these cases, you map the XML documents to equivalent object representations in the interaction layer before delegating the request to the business logic.

The service processing layer holds all business logic used to process client requests. It is also responsible for integrating with EISs and other Web services. In the case of existing applications adding a Web service interface, the existing application itself typically forms the service processing layer.

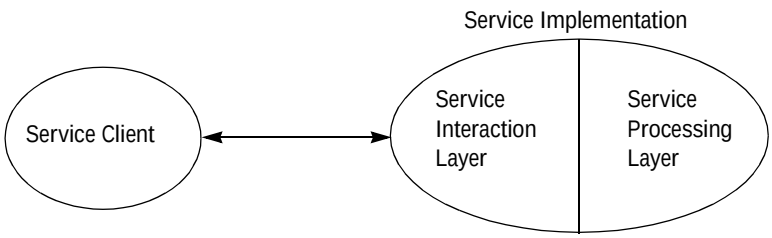

Figure 3.2 Layered View of a Web Service

Viewing your service implementation in terms of layers helps to:

- Clearly divide responsibilities
- Provide a common or single location for request processing (both pre- and post-processing) logic in the interaction layer
- Expose existing business logic as a Web service

To put this notion of a layered view in the proper context, let's look at an example such as adventure builder's business process Web service scenario. In this scenario, a partner travel agency uses adventure builder enterprise's Web service to build a travel itinerary for its clients. Through the service interface it exposes to these travel agencies, adventure builder enterprise receives business documents (in XML format) containing all required details for travel itinerary requests. Adventure builder uses its existing workflow systems to process and satisfy these partner requests. The interaction layer of adventure builder's exposed Web service interface validates these incoming business documents, then converts the incoming XML documents to its internal format or maps document content to Java objects. Once the conversion is finished, control passes to the workflow mechanisms in the processing layer where travel requests are completed. The interaction layer generates responses for completed travel requests, converts responses to XML documents or other appropriate formats, and ensures that responses are relayed to the partner agencies.

It is important to clarify the extent of the preprocessing performed at the interaction layer, since it differs from the JAX-RPC runtime processing. Adventure builder's interaction layer—its exposed Web service interface—applies service-specific preprocessing to requests coming in to the service. This service-specific preprocessing is performed only if required by the service logic, and it includes converting incoming XML documents to a suitable form or mapping the document contents to Java objects. This mapping of incoming XML documents to business objects is not the same as the JAX-RPC runtime mapping between XML documents and Java objects. Although the container performs the JAX-RPC runtime mapping for all requests and responses, the developer chooses the mapping of incoming XML documents to business objects.

❒ Although there are advantages, as noted previously, to viewing a service in terms of interaction and processing layers, a Web service may opt to merge these two layers into a single layer. There are times when multiple layers make

a service unnecessarily complicated and, in these cases, it may be simpler to design the service as one layer. Typically, this happens in scenarios where the logic in either layer is too small to merit a separate layer.

The weather service scenario is one such service that might benefit from merging the interaction and processing layers into a single layer. This type of service does not need to preprocess incoming requests. A client request to the service for weather information simply includes a name or zip code to identify the location. The service looks up the location's weather information, forms a response containing the information, and returns it to the client. Since incoming requests require no preprocessing, a layered view of the weather service only complicates what otherwise should be a simple service.

3.4 Designing a Service's Interaction Layer

A service's interaction layer has several major responsibilities, and chief among them is the design of the interface the service presents to the client. Since clients access the service through it, the interface is the starting point of a client's interaction with the service. The interaction layer also handles other responsibilities, such as receiving client requests, delegating requests to appropriate business logic, and creating and sending responses. This section examines the responsibilities of the interaction layer and highlights some guidelines for its design.

3.4.1 Designing the Interface

There are some considerations to keep in mind as you design the interface of your Web service, such as issues regarding overloading methods, choosing the endpoint type, and so forth. Before examining these issues, decide on the approach you want to take for developing the service's interface definition.

Two approaches to developing the interface definition for a Web service are:

1. Java-to-WSDL—Start with a set of Java interfaces for the Web service and from these create the Web Services Description Language (WSDL) description of the service for others to use.
2. WSDL-to-Java—Start with a WSDL document describing the details of the Web service interface and use this information to build the corresponding Java interfaces.

How do these two approaches compare? Starting with Java interfaces and creating a WSDL document is probably the easier of the two approaches. With this approach, you need not know any WSDL details because you use vendor-provided tools to create the WSDL description. While these tools make it easy for you to generate WSDL files from Java interfaces, you do lose some control over the WSDL file creation.

❒ With the Java-to-WSDL approach, keep in mind that the exposed service interface may be too unstable from a service evolution point of view.

With the Java-to-WSDL approach, it may be hard to evolve the service interface without forcing a change in the corresponding WSDL document, and changing the WSDL might require rewriting the service's clients. These changes, and the accompanying instability, can affect the interoperability of the service itself. Since achieving interoperability is a prime reason to use Web services, the instability of the Java-to-WSDL approach is a major drawback. Also, keep in mind that different tools may use different interpretations for certain Java types (for example, `java.util.Date` might be interpreted as `java.util.Calendar`), resulting in different representations in the WSDL file. While not common, these representation variations may result in some semantic surprises.

On the other hand, the WSDL-to-Java approach gives you a powerful way to expose a stable service interface that you can evolve with relative ease. Not only does it give you greater design flexibility, the WSDL-to-Java approach also provides an ideal way for you to finalize all service details—from method call types and fault types to the schemas representing exchanged business documents—before you even start a service or client implementation. Although a good knowledge of WSDL and the WS-I Basic Profile is required to properly describe these Web services details, using available tools helps address these issues.

After you decide on the approach to take, you must still resolve other interface design details, which are described in the next sections.

3.4.1.1 Choice of the Interface Endpoint Type

In the J2EE platform, you have two choices for implementing the Web service interface—you can use a JAX-RPC service endpoint (also referred to as a Web tier endpoint) or an EJB service endpoint (also referred to as an EJB tier endpoint). Using one of these endpoint types makes it possible to embed the endpoint in the same tier as the service implementation. This simplifies the service implementation, because

it obviates the need to place the endpoint in its own tier where the presence of the endpoint is solely to act as a proxy directing requests to other tiers that contain the service's business logic.

When you develop a new Web service that does *not* use existing business logic, choosing the endpoint type to use for the Web service interface is straightforward. The endpoint type choice depends on the nature of your business logic—whether the business logic of the service is completely contained within either the Web tier or the EJB tier:

- ❒ Use a JAX-RPC service endpoint when the processing layer is within the Web tier.
- ❒ Use an EJB service endpoint when the processing layer is only on the EJB tier.

When you add a Web service interface to an existing application or service, you must consider whether the existing application or service preprocesses requests before delegating them to the business logic. If so, then keep the following guideline in mind:

- ❒ When you add a Web service interface for an existing application, choose an endpoint type suited for the tier on which the preprocessing logic occurs in the existing application. Use a JAX-RPC service endpoint when the preprocessing occurs on the Web tier of the existing application and an EJB service endpoint when preprocessing occurs on the EJB tier.

If the existing application or service does not require preprocessing of the incoming request, choose the appropriate endpoint that is present in the same tier as the existing business logic. Besides these major considerations for choosing an endpoint type, there are other, more subtle differences between an EJB service endpoint and a JAX-RPC service endpoint. You may find it helpful to keep in mind these additional points when choosing a Web service endpoint type:

- **Multi-threaded access considerations**—An EJB service endpoint, because it is implemented as a stateless session bean, need not worry about multi-threaded access since the EJB container is required to serialize requests to any particular instance of a stateless session bean. For a JAX-RPC service

endpoint, on the other hand, you must do the synchronization yourself in the source code.

❒ A JAX-RPC service endpoint has to handle concurrent client access on its own, whereas the EJB container takes care of concurrent client access for an EJB service endpoint.

- **Transaction considerations**—The transactional context of the service implementation's container determines the transactional context in which a service implementation runs. Since a JAX-RPC service endpoint runs in a Web container, its transactional context is unspecified. There is also no declarative means to automatically start the transaction. Thus, you need to use JTA to explicitly demarcate the transaction.

  On the other hand, an EJB service endpoint runs in the transaction context of an EJB container. You as the developer need to declaratively demarcate transactions. The service's business logic thus runs under the transactional context as defined by the EJB's `container-transaction` element in the deployment descriptor.

❒ If the Web service's business logic requires using transactions (and the service has a JAX-RPC service endpoint), you must implement the transaction-handling logic using JTA or some other similar facility. If your service uses an EJB service endpoint, you can use the container's declarative transaction services. By doing so, the container is responsible for handling transactions according to the setting of the deployment descriptor element `container-transaction`.

- **Considerations for method-level access permissions**—A Web service's methods can be accessed by an assortment of different clients, and you may want to enforce different access constraints for each method.

❒ When you want to control service access at the individual method level, consider using an EJB service endpoint rather than a JAX-RPC service endpoint.

  Enterprise beans permit method-level access permission declaration in the deployment descriptor—you can declare various access permissions for different enterprise bean methods and the container correctly handles access to these methods. This holds true for an EJB service endpoint, since it is a stateless ses-

sion bean. A JAX-RPC service endpoint, on the other hand, does not have a facility for declaring method-level access constraints, requiring you to do this programmatically. See Chapter 7 for more information.

- **HTTP session access considerations**—A JAX-RPC service endpoint, because it runs in the Web container, has complete access to an `HttpSession` object. Access to an `HttpSession` object, which can be used to embed cookies and store client state, may help you build session-aware clients. An EJB service endpoint, which runs in the EJB container, has no such access to Web container state. However, generally HTTP session support is appropriate for short duration conversational interactions, whereas Web services often represent business processes with longer durations and hence need additional mechanisms. See "Correlating Messages" on page 359 for one such strategy.

3.4.1.2 Granularity of Service

Much of the design of a Web service interface involves designing the service's operations, or its methods. You first determine the service's operations, then define the method signature for each operation. That is, you define each operation's parameters, its return values, and any errors or exceptions it may generate.

❒ It is important to consider the granularity of the service's operations when designing your Web service interface.

For those Web services that implement a business process, the nature of the business process itself often dictates the service's granularity. Business processes that exchange documents, such as purchase orders and invoices, by their nature result in a Web service interface that is coarse grained. With more interactive Web services, you need to carefully choose the granularity of these operations.

You should keep the same considerations in mind when designing the methods for a Web service as when designing the methods of a remote enterprise bean. This is particularly true not only regarding the impact of remote access on performance but also with Web services; it is important with Web services because there is an underlying XML representation requiring parsing and taking bandwidth. Thus, a good rule is to define the Web service's interface for optimal granularity of its operations; that is, find the right balance between coarse-grained and fine-grained granularity.

❒ Generally, you should consolidate related fine-grained operations into more coarse-grained ones to minimize expensive remote method calls.

More coarse-grained service operations, such as returning catalog entries in sets of categories, keep network overhead lower and improve performance. However, they are sometimes less flexible from a client's point of view. While finer-grained service operations, such as browsing a catalog by products or items, offer a client greater flexibility, these operations result in greater network overhead and reduced performance.

❒ Keep in mind that too much consolidation leads to inefficiencies.

For example, consolidating logically different operations is inefficient and should be avoided. It is much better to consolidate similar operations or operations that a client is likely to use together, such as querying operations.

❒ When exposing existing stateless session beans as Web service endpoints, ensure that the Web service operations are sufficiently coarse grained.

If you are planning to expose existing stateless session beans as Web service endpoints, remember that such beans may not have been designed with Web services in mind. Hence, they may be too fine grained to be good Web service endpoints. You should consider consolidating related operations into a single Web service operation.

Good design for our airline reservation Web service, for example, is to expect the service's clients to send all information required for a reservation—destination, preferred departure and arrival times, preferred airline, and so forth—in one invocation to the service, that is, as one large message. This is far more preferable than to have a client invoke a separate method for each piece of information comprising the reservation. To illustrate, it is preferable to have clients use the interface shown in Code Example 3.1.

```
public interface AirlineTicketsIntf extends Remote {
    public String submitReservationRequest(
           AirReservationDetails details) throws RemoteException;
}
```

Code Example 3.1 Using Consolidation for Greater Efficiency (Recommended)

Code Example 3.1 combines logically-related data into one large message for a more efficient client interaction with the service. This is preferable to receiving the data with individual method calls, as shown in Code Example 3.2.

```
public interface AirlineTicketsIntf extends Remote {
    public String submitFlightInformation(FlightDetails fltInfo)
                    throws RemoteException;
    public String submitPreferredDates(Date depart, Date arrive)
                  throws RemoteException;
    // other similar methods
}
```

Code Example 3.2 Retrieving Data with Separate Method Calls (Not Recommended)

However, it might not be a good idea to combine in a single service invocation the same reservation with an inquiry method call.

Along with optimal granularity, you should consider data caching issues. Coarse-grained services involve transferring large amounts of data. If you opt for more coarse-grained service operations, it is more efficient to cache data on the client side to reduce the number of round trips between the client and the server.

3.4.1.3 Parameter Types for Web Service Operations

A Web service interface exposes a set of method calls to clients. When invoking a service interface method, a client may have to set values for the parameters associated with the call. When you design an interface's methods, choose carefully the types of these parameters. Keep in mind that a method call and its parameters are sent as a SOAP message between the client and the service. To be part of a SOAP message, parameters must be mapped to XML. When received at the client or service end, the same parameters must be mapped from XML to their proper types or objects. This section describes some guidelines to keep in mind when defining method call parameters and return values.

Note: Since each call potentially may return a value, the discussion in this section about parameter values applies equally to return values.

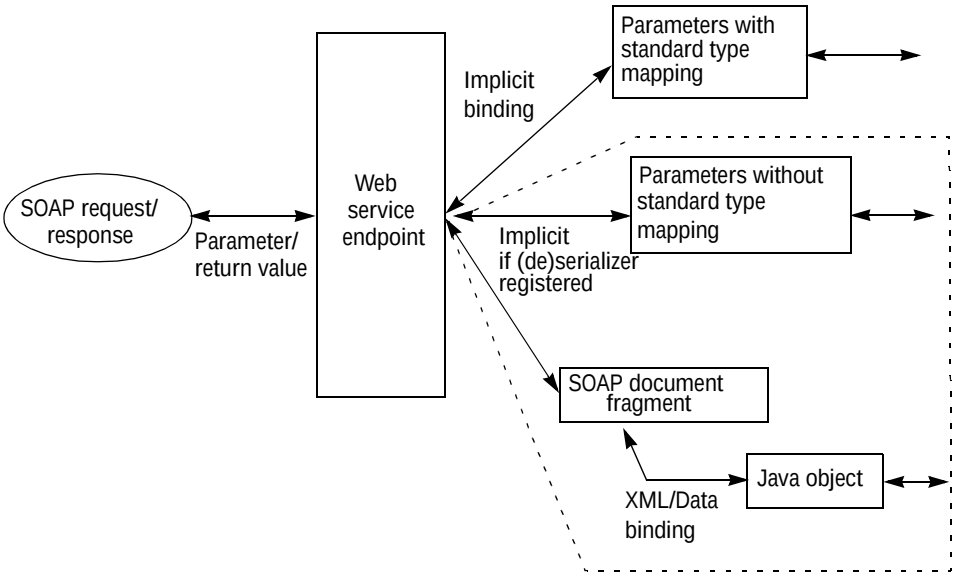

Figure 3.3 Binding Parameters and Return Values with JAX-RPC

Parameters for Web service method calls may be standard Java objects and types, XML documents, or even nonstandard types. Whether you use the Java-to-WSDL approach or the WSDL-to-Java approach, each type of parameter must be mapped to its XML equivalent in the SOAP message. Figure 3.3 shows how the binding happens for various types of parameters.

3.4.1.3.1 Java Objects as Parameters

Parameters for Web service calls can be standard Java types and objects. If you use the Java-to-WSDL approach, you specify the parameter types as part of the arguments of the method calls of your Java interface. If you use the WSDL-to-Java approach, you specify the parameter types as the `type` or `element` attributes of the `part` element of each `message` in your WSDL. The type of a parameter that you use has a significant effect on the portability and interoperability of your service.

The platform supports the following Java data types. (Refer to the JAX-RPC specification at `http://java.sun.com/xml/jaxrpc/` for the equivalent WSDL mappings for these Java data types.)

- Java primitive types `boolean`, `byte`, `short`, `int`, `long`, `float`, and `double`, along with their corresponding wrapper Java classes
- Standard Java classes: `String`, `Date`, `Calendar`, `BigInteger`, `BigDecimal`, `QName`, and `URI`
- Java arrays with JAX-RPC-supported Java types as members
- JAX-RPC value types—user-defined Java classes, including classes with JavaBeans™ component-like properties

❒ When designing parameters for method calls in a Web service interface, choose parameters that have standard type mappings. (See Figure 3.3.) Always keep in mind that the portability and interoperability of your service is reduced when you use parameter types that by default are not supported.

As Figure 3.3 shows, parameters that have standard type mappings are bound implicitly. However, the developer must do more work when using parameters that do not have standard type mappings. See "Handling Nonstandard Type Parameters" on page 76 for more details on using nonstandard Java types and possible side effects of such use.

Here are some additional helpful points to consider when you use Java objects with standard type mappings as parameters.

1. Many applications and services need to pass lists of objects. However, utilities for handling lists, such as `ArrayList` and `Collection`, to name a few, are not supported standard types. Instead, Java arrays provide equivalent functionality, and have a standard mapping provided by the platform.
2. JAX-RPC value types are user-defined Java classes with some restrictions. They have constructors and may have fields that are public, private, protected, static, or transient. JAX-RPC value types may also have methods, including set and get methods for setting and getting Java class fields.

   However, when mapping JAX-RPC value types to and from XML, there is no standard way to retain the order of the parameters to the constructors and other methods. Hence, avoid setting the JAX-RPC value type fields through the constructor. Using the get and set methods to retrieve or set value type fields avoids this mapping problem and ensures portability and interoperability.
3. The J2EE platform supports nested JAX-RPC value types; that is, JAX-RPC value types that reference other JAX-RPC value types within themselves. For

clarity, it is preferable to use this feature and embed value type references rather than to use a single flat, large JAX-RPC value type class.

4. The J2EE platform, because of its support for the SOAP message with attachment protocol, also supports the use of MIME-encoded content. It provides Java mappings for a subset of MIME types. (See Table 3.1.)

Table 3.1 Mapping of MIME Types

MIME Type	Java Type
`image/gif`	`java.awt.Image`
`image/jpeg`	`java.awt.Image`
`text/plain`	`java.lang.String`
`multipart/*`	`javax.mail.internet.MimeMultipart`
`text/xml` or `application/xml`	`javax.xml.transform.Source`

Since the J2EE container automatically handles mappings based on the Java types, using these Java-MIME mappings frees you from the intricacies of sending and retrieving documents and images as part of a service's request and response handling. For example, your service, if it expects to receive a GIF image with a MIME type of `image/gif`, can expect the client to send a `java.awt.Image` object. A sample Web service interface that receives an image might look like the one shown in Code Example 3.3:

```
import java.awt.Image;
public interface WeatherMapService extends Remote {
    public void submitWeatherMap(Image weatherMap)
                    throws RemoteException, InvalidMapException;
}
```

Code Example 3.3 Receiving a `java.awt.Image` Object

In this example, the `Image` object lets the container implementation handle the image-passing details. The container provides `javax.activation.DataHandler` classes, which work with the Java Activation Framework to accomplish the Java-MIME and MIME-Java mappings.

- Considering this mapping between Java and MIME types, it is best to send images and XML documents that are in a Web service interface using the Java types shown in Table 3.1. However, you should be careful about the effect on the interoperability of your service. See "Interoperability" on page 86 for more details.

3.4.1.3.2 XML Documents as Parameters

There are scenarios when you want to pass XML documents as parameters. Typically, these occur in business-to-business interactions where there is a need to exchange legally binding business documents, track what is exchanged, and so forth. Exchanging XML documents as part of a Web service is addressed in a separate section—see "Handling XML Documents in a Web Service" on page 105 for guidelines to follow when passing XML documents as parameters.

3.4.1.3.3 Handling Nonstandard Type Parameters

JAX-RPC technology, in addition to providing a rich standard mapping set between XML and Java data types, also provides an extensible type mapping framework. Developers can use this framework to specify pluggable, custom serializers and deserializers that support nonstandard type mappings.

- Extensible type mapping frameworks, which developers may use to support nonstandard type mappings, are not yet a standard part of the J2EE platform.

Vendors currently can provide their own solutions to this problem. It must be emphasized that if you implement a service using some vendor's implementation-specific type mapping framework, then your service is not guaranteed to be portable and interoperable.

- Because of portability limitations, you should avoid passing parameters that require the use of vendor-specific serializers or deserializers.

Instead, a better way is to pass these parameters as SOAP document fragments represented as a DOM subtree in the service endpoint interface. (See Figure 3.3.) If so, you should consider binding (either manually or using JAXB) the SOAP fragments to Java objects before passing them to the processing layer to avoid tightly coupling the business logic with the document fragment.

3.4.1.4 Interfaces with Overloaded Methods

In your service interface, you may overload methods and expose them to the service's clients. Overloaded methods share the same method name but have different parameters and return values. If you do choose to use overloaded methods as part of your service interface, keep in mind that there are some limitations, as follows:

- If you choose the WSDL-to-Java approach, there are limitations to representing overloaded methods in a WSDL description. In the WSDL description, each method call and its response are represented as unique SOAP messages. To represent overloaded methods, the WSDL description would have to support multiple SOAP messages with the same name. WSDL version 1.1 does not have this capability to support multiple messages with the same name.
- If you choose the Java-to-WSDL approach and your service exposes overloaded methods, be sure to check how any vendor-specific tools you are using represent these overloaded methods in the WSDL description. You need to ensure that the WSDL representation of overloaded methods works in the context of your application.

Let's see how this applies in the weather service scenario. As the provider, you might offer the service to clients, letting them look up weather information by city name or zip code. If you use the Java-to-WSDL approach, you might first define the `WeatherService` interface as shown in Code Example 3.4.

```
public interface WeatherService extends Remote {
    public String getWeather(String city) throws RemoteException;
    public String getWeather(int zip) throws RemoteException;
}
```

Code Example 3.4 `WeatherService` Interface for Java-to-WSDL Approach

After you define the interface, you run the vendor-provided tool to create the WSDL from the interface. Each tool has its own way of representing the `getWeather` overloaded methods in the WSDL, and your WSDL reflects the particular tool you use. For example, if you use the J2EE 1.4 SDK from Sun Microsystems, its `wscompile` tool creates from the `WeatherService` interface the WSDL shown in Code Example 3.5.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherWebService" .......>
    <types/>
    <message name="WeatherService_getWeather">
        <part name="int_1" type="xsd:int"/>
    </message>
    <message name="WeatherService_getWeatherResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <message name="WeatherService_getWeather2">
        <part name="String_1" type="xsd:string"/>
    </message>
    <message name="WeatherService_getWeather2Response">
        <part name="result" type="xsd:string"/>
    </message>
    ...
</definitions>
```

Code Example 3.5 Generated WSDL for `WeatherService` Interface

Notice that the WSDL represents the `getWeather` overloaded methods as two different SOAP messages, naming one `getWeather`, which takes an integer for the zip code as its parameter, and the other `getWeather2`, which takes a string parameter for the city. As a result, a client interested in obtaining weather information using a city name invokes the service by calling `getWeather2`, as shown in Code Example 3.6.

```
...
Context ic = new InitialContext();
WeatherWebService weatherSvc = (WeatherWebService)
       ic.lookup("java:comp/env/service/WeatherService");
WeatherServiceIntf port = (WeatherServiceIntf)
       weatherSvc.getPort(WeatherServiceIntf.class);
String returnValue = port.getWeather2("San Francisco");
...
```

Code Example 3.6 Using Weather Service Interface with Java-to-WSDL Approach

For example, to obtain the weather information for San Francisco, the client called `port.getWeather2("San Francisco")`. Keep in mind that another tool may very likely generate a WSDL whose representation of overloaded methods is different.

- You may want to avoid using overloaded methods in your Java interface altogether if you prefer to have only intuitive method names in the WSDL.

If instead you choose to use the WSDL-to-Java approach, your WSDL description might look as follows. (See Code Example 3.7.)

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherWebService" ...>
    <types/>
    <message name="WeatherService_getWeatherByZip">
        <part name="int_1" type="xsd:int"/>
    </message>
    <message name="WeatherService_getWeatherByZipResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <message name="WeatherService_getWeatherByCity">
        <part name="String_1" type="xsd:string"/>
    </message>
    <message name="WeatherService_getWeatherByCityResponse">
        <part name="result" type="xsd:string"/>
    </message>
    ...
</definitions>
```

Code Example 3.7 WSDL for Weather Service with Overloaded Methods Avoided

Since the messages in a WSDL file must have unique names, you must use different message names to represent methods that you would otherwise overload. These different message names actually convert to different method calls in your interface. Notice that the WSDL includes a method `getWeatherByZip`, which takes an integer parameter, and a method `getWeatherByCity`, which takes a string parameter. Thus, a client wishing to obtain weather information by city name from

a `WeatherService` interface associated with the WSDL in Code Example 3.7 might invoke the service as shown in Code Example 3.8.

```
...
Context ic = new InitialContext();
WeatherWebService weatherSvc = (WeatherWebService)
       ic.lookup("java:comp/env/service/WeatherService");
WeatherServiceIntf port = (WeatherServiceIntf)
       weatherSvc.getPort(WeatherServiceIntf.class);
String returnValue = port.getWeatherByCity("San Francisco");
...
```

Code Example 3.8 Using Weather Service with WSDL-to-Java Approach

3.4.1.5 Handling Exceptions

Just like any Java or J2EE application, a Web service application may encounter an error condition while processing a client request. A Web service application needs to properly catch any exceptions thrown by an error condition and propagate these exceptions. For a Java application running in a single virtual machine, you can propagate exceptions up the call stack until reaching a method with an exception handler that handles the type of exception thrown. To put it another way, for non-Web service J2EE and Java applications, you may continue to throw exceptions up the call stack, passing along the entire stack trace, until reaching a method with an exception handler that handles the type of exception thrown. You can also write exceptions that extend or inherit other exceptions.

However, throwing exceptions in Web service applications has additional constraints that impact the design of the service endpoint. When considering how the service endpoint handles error conditions and notifies clients of errors, you must keep in mind these points:

- Similar to requests and responses, exceptions are also sent back to the client as part of the SOAP messages.
- Your Web service application should support clients running on non-Java platforms that may not have the same, or even similar, error-handling mechanisms as the Java exception-handling mechanism.

A Web service application may encounter two types of error conditions. One type of error might be an irrecoverable system error, such as an error due to a network connection problem. When an error such as this occurs, the JAX-RPC runtime on the client throws the client platform's equivalent of an irrecoverable system exception. For Java clients, this translates to a `RemoteException`.

A Web service application may also encounter a recoverable application error condition. This type of error is called a service-specific exception. The error is particular to the specific service. For example, a weather Web service might indicate an error if it cannot find weather information for a specified city.

To illustrate the Web service exception-handling mechanism, let's examine it in the context of the weather Web service example. When designing the weather service, you want the service to be able to handle a scenario in which the client requests weather information for a nonexistent city. You might design the service to throw a service-specific exception, such as `CityNotFoundException`, to the client that made the request. You might code the service interface so that the `getWeather` method throws this exception. (See Code Example 3.9.)

```
public interface WeatherService extends Remote {
    public String getWeather(String city) throws
                CityNotFoundException, RemoteException;
}
```

Code Example 3.9 Throwing a Service-Specific Exception

Service-specific exceptions like `CityNotFoundException`, which are thrown by the Web service to indicate application-specific error conditions, must be checked exceptions that directly or indirectly extend `java.lang.Exception`. They cannot be unchecked exceptions. Code Example 3.10 shows a typical implementation of a service-specific exception, such as for `CityNotFoundException`.

```
public class CityNotFoundException extends Exception {
    private String message;
    public CityNotFoundException(String message) {
        super(message);
        this.message = message;
    }
    public String getMessage() {
        return message;
```

```
    }
}
```

Code Example 3.10 Implementation of a Service-Specific Exception

Code Example 3.11 shows the service implementation for the same weather service interface. This example illustrates how the service might throw `CityNotFoundException`.

```
public class WeatherServiceImpl implements WeatherService {
    public String getWeather(String city)
                                throws CityNotFoundException {
        if(!validCity(city))
            throw new CityNotFoundException(city + " not found");
        // Get weather info and return it back
    }
}
```

Code Example 3.11 Example of a Service Throwing a Service-Specific Exception

Chapter 5 describes the details of handling exceptions on the client side. (In particular, refer to "Handling Exceptions" on page 230.) On the service side, keep in mind how to include exceptions in the service interface and how to throw them. Generally, you want to do the following:

- ❒ Convert application-specific errors and other Java exceptions into meaningful service-specific exceptions and throw these service-specific exceptions to the clients.

Although they promote interoperability among heterogeneous platforms, Web service standards cannot address every type of exception thrown by different platforms. For example, the standards do not specify how Java exceptions such as `java.io.IOException` and `javax.ejb.EJBException` should be returned to the client. As a consequence, it is important for a Web service—from the service's interoperability point of view—to not expose Java-specific exceptions (such as those just mentioned) in the Web service interface. Instead, throw a service-specific exception. In addition, keep the following points in mind:

- You cannot throw nonserializable exceptions to a client through the Web service endpoint.
- When a service throws `java` or `javax` exceptions, the exception type and its context information are lost to the client that receives the thrown exception. For example, if your service throws a `javax.ejb.FinderException` exception to the client, the client may receive an exception named `FinderException`, but its type information may not be available to the client. Furthermore, the type of the exception to the client may not be the same as the type of the thrown exception. (Depending on the tool used to generate the client-side interfaces, the exception may even belong to some package other than `javax.ejb`.)

As a result, you should avoid directly throwing `java` and `javax` exceptions to clients. Instead, when your service encounters one of these types of exceptions, wrap it within a meaningful service-specific exception and throw this service-specific exception back to the client. For example, suppose your service encounters a `javax.ejb.FinderException` exception while processing a client request. The service should catch the `FinderException` exception, and then, rather than throwing this exception as is back to the client, the service should instead throw a service-specific exception that has more meaning for the client. See Code Example 3.12.

```
...
try {
    // findByPrimaryKey
    // Do processing
    // return results
} catch (javax.ejb.FinderException fe) {
    throw new InvalidKeyException(
        “Unable to find row with given primary key”);
}
```

Code Example 3.12 Converting an Exception into a Service-Specific Exception

❒ Exception inheritances are lost when you throw a service-specific exception.

You should avoid defining service-specific exceptions that inherit or extend other exceptions. For example, if `CityNotFoundException` in Code Example 3.10

extends another exception, such as `RootException`, then when the service throws `CityNotFoundException`, methods and properties inherited from `RootException` are *not* passed to the client.

❒ The exception stack trace is not passed to the client.

The stack trace for an exception is relevant only to the current execution environment and is meaningless on a different system. Hence, when a service throws an exception to the client, the client does not have the stack trace explaining the conditions under which the exception occurred. Thus, you should consider passing additional information in the message for the exception.

Web service standards make it easier for a service to pass error conditions to a client in a platform-independent way. While the following discussion may be of interest, it is not essential that developers know these details about the J2EE platform's error-handling mechanisms for Web services.

As noted previously, error conditions are included within the SOAP messages that a service returns to clients. The SOAP specification defines a message type, called `fault`, that enables error conditions to be passed as part of the SOAP message yet still be differentiated from the request or response portion. Similarly, the WSDL specification defines a set of operations that are possible on an endpoint. These operations include `input` and `output` operations, which represent the request and response respectively, and an operation called `fault`.

A SOAP `fault` defines system-level exceptions, such as `RemoteException`, which are irrecoverable errors. The WSDL `fault` denotes service-specific exceptions, such as `CityNotFoundException`, and these are recoverable application error conditions. Since the WSDL `fault` denotes a recoverable error condition, the platform can pass it as part of the SOAP response message. Thus, the standards provide a way to exchange fault messages and map these messages to operations on the endpoint.

Code Example 3.13 shows the WSDL code for the same weather Web service example. This example illustrates how service-specific exceptions are mapped just like input and output messages are mapped.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
    ...
    <message name="WeatherService_getWeather">
        <part name="String_1" type="xsd:string"/>
```

```
    </message>
    <message name="WeatherService_getWeatherResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <message name="CityNotFoundException">
        <part name="CityNotFoundException"
                      element="tns:CityNotFoundException"/>
    </message>
    <portType name="WeatherService">
        <operation name="getWeather" parameterOrder="String_1">
            <input message="tns:WeatherService_getWeather"/>
            <output message=
                   "tns:WeatherService_getWeatherResponse"/>
            <fault name="CityNotFoundException"
                      message="tns:CityNotFoundException"/>
        </operation>
    </portType>
    ...
</definitions>
```

Code Example 3.13 Mapping a Service-Specific Exception in WSDL

3.4.1.6 Use of Handlers

As discussed in Chapter 2, and as shown in Figure 3.1, JAX-RPC technology enables you to plug in SOAP message handlers, thus allowing processing of SOAP messages that represent requests and responses. Plugging in SOAP message handlers gives you the capability to examine and modify the SOAP requests before they are processed by the Web service and to examine and modify the SOAP responses before they are delivered to the client.

Handlers are particular to a Web service and are associated with the specific port of the service. As a result of this association, the handler's logic applies to all SOAP requests and responses that pass through a service's port. Thus, you use these message handlers when your Web service must perform some SOAP message-specific processing common to all its requests and responses. Because handlers are common to all requests and responses that pass through a Web service endpoint, keep the following guideline in mind:

❒ It is not advisable to put in a handler business logic or processing particular to specific requests and responses.

You cannot store client-specific state in a handler: A handler's logic acts on all requests and responses that pass through an endpoint. However, you may use the handler to store port-specific state, which is state common to all method calls on that service interface. Note also that handlers execute in the context of the component in which they are present.

❒ Do not store client-specific state in a handler.

Also note that handlers work directly on the SOAP message, and this involves XML processing. You can use handlers to pass client-specific state through the message context. (See "Passing Context Information on Web Service Calls" on page 366.)

❒ Use of handlers can result in a significant performance impact for the service as a whole.

Use of handlers could potentially affect the interoperability of your service. See the next section on interoperability. Keep in mind that it takes advanced knowledge of SOAP message manipulation APIs (such as SAAJ) to correctly use handlers. To avoid errors, Web service developers should try to use existing or vendor-supplied handlers. Using handlers makes sense primarily for writing system services such as auditing, logging, and so forth.

3.4.1.7 Interoperability

A major benefit of Web services is interoperability between heterogeneous platforms. To get the maximum benefit, you want to design your Web service to be interoperable with clients on any platform, and, as discussed in Chapter 2, the Web Services Interoperability (WS-I) organization helps in this regard. WS-I promotes a set of generic protocols for the interoperable exchange of messages between Web services. The WS-I Basic Profile promotes interoperability by defining and recommending how a set of core Web services specifications and standards (including SOAP, WSDL, UDDI, and XML) can be used for developing interoperable Web services.

In addition to the WS-I protocols, other groups, such as SOAPBuilders Interoperability group (see `http://java.sun.com/wsinterop/sb/index.html`), provide common testing grounds that make it easier to test the interoperability of various SOAP implementations. This has made it possible for various Web services technology vendors to test the interoperability of implementations of their standards. When you implement your service using technologies that adhere to the WS-I Basic Profile specifications, you are assured that such services are interoperable.

Apart from these standards and testing environments, you as the service developer must design and implement your Web service so that maximum interoperability is possible. For maximum interoperability, you should keep these three points in mind:

1. The two messaging styles and bindings supported by WSDL
2. The WS-I support for attachments
3. The most effective way to use handlers

WSDL supports two types of messaging styles: `rpc` and `document`. The WSDL `style` attribute indicates the messaging style. (See Code Example 3.14.) A `style` attribute set to `rpc` indicates a RPC-oriented operation, where messages contain parameters and return values, or function signatures. When the `style` attribute is set to `document`, it indicates a document-oriented operation, one in which messages contain documents. Each operation style has a different effect on the format of the body of a SOAP message.

Along with operation styles, WSDL supports two types of serialization and deserialization mechanisms: a `literal` and an `encoded` mechanism. The WSDL `use` attribute indicates which mechanism is supported. (See Code Example 3.14.) A `literal` value for the `use` attribute indicates that the data is formatted according to the abstract definitions within the WSDL document. The `encoded` value means data is formatted according to the encodings defined in the URI specified by the `encodingStyle` attribute. Thus, you can choose between an rpc or document style of message passing and each message can use either a literal or encoded data formatting.

❒ Because the WS-I Basic Profile 1.0, to which J2EE1.4 platform conforms, supports only literal bindings, you should avoid encoded bindings.

- Literal bindings cannot represent complex types, such as objects with circular references, in a standard way.

Code Example 3.14 shows a snippet from the WSDL document illustrating how the sample weather service specifies these bindings.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions .......>
<binding name="WeatherServiceBinding" type="tns:WeatherService">
    <operation name="getWeather">
        <input>
            <soap:body use="literal"
                       namespace="urn:WeatherWebService"/>
        </input>
        <output>
            <soap:body use="literal"
                       namespace="urn:WeatherWebService"/>
        </output>
        <soap:operation soapAction=""/></operation>
        <soap:binding style="rpc"
               transport="http://schemas.xmlsoap.org/soap/http" />
        </binding>
        <service .....>
</definitions>
```

Code Example 3.14 Specifying WSDL Bindings

It is important to keep in mind these message styles and bindings, particularly when you design the interface using the WSDL-to-Java approach and when you design the WSDL for your service. When you use the Java-to-WSDL approach, you rely on the vendor-provided tools to generate the WSDL for your Java interfaces, and they can be counted on to create WS-I-compliant WSDL for your service. However, note that some vendors may expect you to specify certain options to ensure the creation of a WS-I-compliant WSDL. For example, the J2EE 1.4 SDK from Sun Microsystems provides a `wscompile` tool, which expects the developer to use the `-f:wsi` flag to create the WS-I-compliant WSDL for the service. It is also a good idea to check the WSDL document itself to ensure that whatever tool you use created the document correctly.

Regarding the second issue, you should note that the WS-I Basic Profile 1.0 (which is the profile supported by the J2EE 1.4 platform) does not address attachments. The section, "Parameter Types for Web Service Operations" on page 72, which discussed Java-MIME type mappings provided by the J2EE platform, advised that an efficient design is to use these mappings to send images and XML documents within a completely Java environment. Because the WS-I Basic Profile, version 1.0 does not address attachments, a Web service that uses these mappings may not be interoperable with clients on a non-Java platform.

❒ Since the WS-I Basic Profile 1.0 specification does not address attachments, a Web service using the Java-MIME mappings provided by the J2EE platform is not guaranteed to be interoperable.

Since most Web services rely on an exchange of business documents, and interoperability is not always guaranteed, it is important that you properly understand the options for handling XML documents. The section, "Exchanging XML Documents" on page 107, explains the various options available to Web services for exchanging XML documents in an interoperable manner. It should also be noted that the next version of the WS-I Basic Profile specification addresses a standard way to send attachments, and later versions of the J2EE platforms will incorporate this.

Last is the issue of handlers. Handlers, which give you access to SOAP messages, at the same time impose major responsibilities on you.

❒ When using handlers, you must be careful not to change a SOAP message to the degree that the message no longer complies with WS-I specifications, thereby endangering the interoperability of your service.

This ends the discussion of considerations for designing a Web service interface. The next sections examine other responsibilities of the interaction layer, such as receiving and delegating requests and formulating responses.

3.4.2 Receiving Requests

The interaction layer, through the endpoint, receives client requests. The platform maps the incoming client requests, which are in the form of SOAP messages, to method calls present in the Web service interface.

- Before delegating these incoming client requests to the Web service business logic, you should perform any required security validation, transformation of parameters, and other required preprocessing of parameters.

As noted in "Parameter Types for Web Service Operations" on page 72 and elsewhere, Web service calls are basically method calls whose parameters are passed as either Java objects, XML documents (`javax.xml.transform.Source` objects), or even SOAP document fragments (`javax.xml.soap.SOAPElement` objects).

- For parameters that are passed as Java objects (such as `String`, `int`, JAX-RPC value types, and so forth), do the application-specific parameter validation and map the incoming objects to domain-specific objects in the interaction layer before delegating the request to the processing layer.

You may have to undertake additional steps to handle XML documents that are passed as parameters. These steps, which are best performed in the interaction layer of your service, are as follows:

1. The service endpoint should validate the incoming XML document against its schema. For details and guidelines on how and when to validate incoming XML documents, along with recommended validation techniques, refer to "Validating XML Documents" on page 139.
2. When the service's processing layer and business logic are designed to deal with XML documents, you should transform the XML document to an internally supported schema, if the schema for the XML document differs from the internal schema, before passing the document to the processing layer.
3. When the processing layer deals with objects but the service interface receives XML documents, then, as part of the interaction layer, map the incoming XML documents to domain objects before delegating the request to the processing layer. For details and guidelines on mapping techniques for incoming XML documents, refer to "Mapping Schemas to the Application Data Model" on page 143.

- It is important that these three steps—validation of incoming parameters or XML documents, translation of XML documents to internal supported sche-

mas, and mapping documents to domain objects—be performed as close to the service endpoint as possible, and certainly in the service interaction layer.

A design such as this helps to catch errors early, and thus avoids unnecessary calls and round-trips to the processing layer. Figure 3.4 shows the recommended way to handle requests and responses in the Web service's interaction layer.

The Web service's interaction layer handles all incoming requests and delegates them to the business logic exposed in the processing layer. When implemented in this manner, the Web service interaction layer has several advantages, since it gives you a common location for the following tasks:

- Managing the handling of requests so that the service endpoint serves as the initial point of contact
- Invoking security services, including authentication and authorization
- Validating and transforming incoming XML documents and mapping XML documents to domain objects

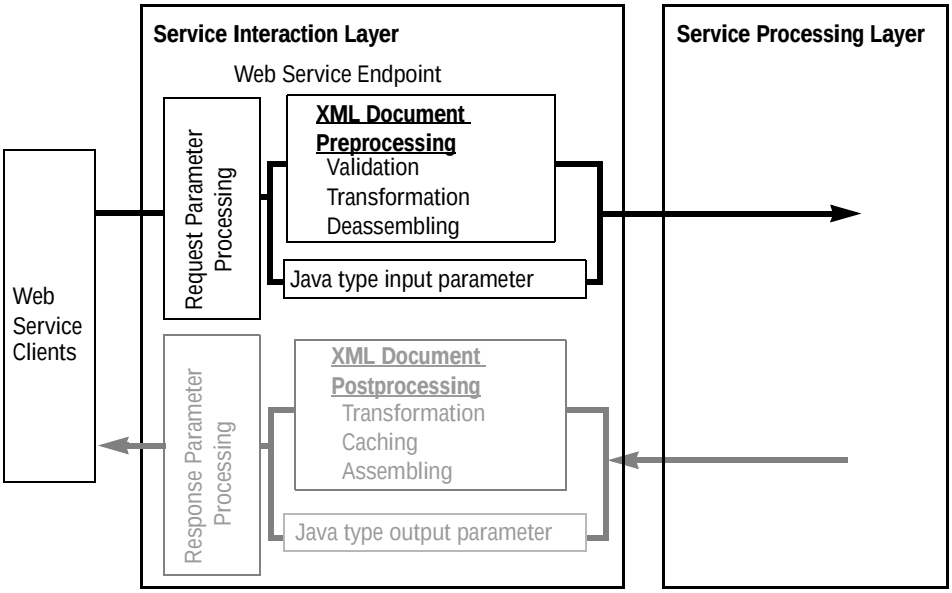

Figure 3.4 Web Service Request Processing

- Delegating to existing business logic
- Handling errors

It is generally advisable to do all common processing—such as security checks, logging, auditing, input validation, and so forth—for requests at the interaction layer as soon as a request is received and before passing it to the processing layer.

3.4.3 Delegating Web Service Requests to Processing Layer

After designing the request preprocessing tasks, the next step is to design how to delegate the request to the processing layer. At this point, consider the kind of processing the request requires, since this helps you decide how to delegate the request to the processing layer. All requests can be categorized into two large categories based on the time it takes to process the request, namely:

- A request that is processed in a short enough time so that a client can afford to block and wait to receive the response before proceeding further. In other words, the client and the service interact in a synchronous manner such that the invoking client blocks until the request is processed completely and the response is received.
- A request that takes a long time to be processed, so much so that it is not a good idea to make the client wait until the processing is completed. In other words, the client and the service interact in an asynchronous manner such that the invoking client need not block and wait until the request is processed completely.

Note: When referring to request processing, we use the terms synchronous and asynchronous from the point of view of when the client's request processing completes fully. Keep in mind that, under the hood, an asynchronous interaction between a client and a service might result in a synchronous invocation over the network, since HTTP is by its nature synchronous. Similarly, SOAP messages sent over HTTP are also synchronous.

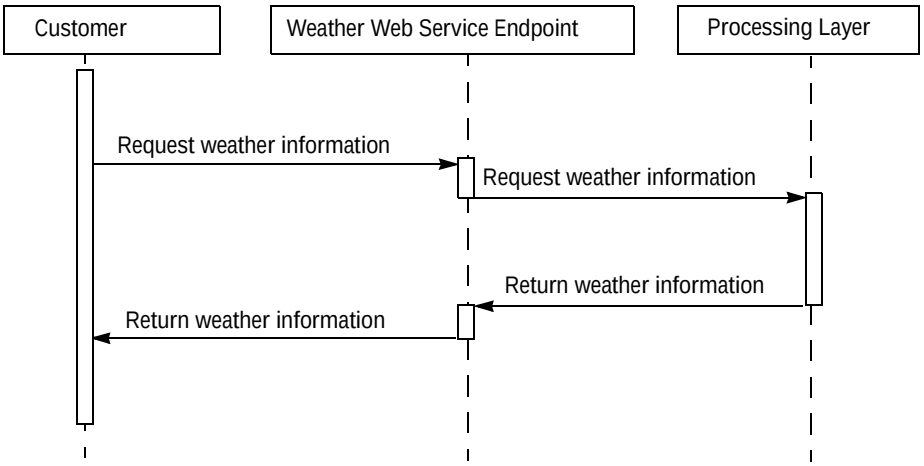

Figure 3.5 Weather Information Service Interaction

The weather information service is a good example of a synchronous interaction between a client and a service. When it receives a client's request, the weather service must look up the required information and send back a response to the client. This look-up and return of the information can be achieved in a relatively short time, during which the client can be expected to block and wait. The client continues its processing only after it obtains a response from the service. (See Figure 3.5.)

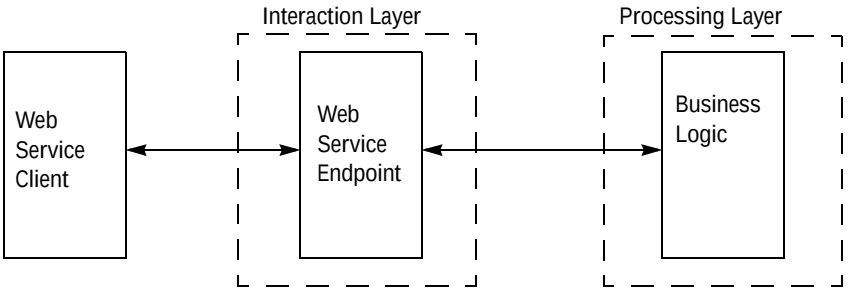

Figure 3.6 Synchronous Interaction Between Client and Service

A Web service such as this can be designed using a service endpoint that receives the client's request and then delegates the request directly to the service's appropriate logic in the processing layer. The service's processing layer processes the request and, when the processing completes, the service endpoint returns the response to the client. (See Figure 3.6.)

Code Example 3.15 shows the weather service interface performing some basic parameter validation checks in the interaction layer. The interface also gets required information and passes that information to the client in a synchronous manner:

```
public class WeatherServiceImpl implements
                    WeatherService, ServiceLifecycle {

   public void init(Object context) throws JAXRPCException {....}

   public String getWeather(String city)
                        throws CityNotFoundException {

      /** Validate parameters **/
      if(!validCity(city))
         throw new CityNotFoundException(....);

      /** Get weather info form processing layer and **/
      / **return results **/
      return (getWeatherInfoFromDataSource(city));
   }

   public void destroy() {....}
}
```

Code Example 3.15 Performing a Synchronous Client Interaction

Now let's examine an asynchronous interaction between a client and a service. When making a request for this type of service, the client cannot afford to wait for the response because of the significant time it takes for the service to process the request completely. Instead, the client may want to continue with some other processing. Later, when it receives the response, the client resumes whatever processing initiated the service request. Typically in these types of ser-

vices, the content of the request parameters initiates and determines the processing workflow—the steps to fulfill the request—for the Web service. Often, fulfilling a request requires multiple workflow steps.

The travel agency service is a good example of an asynchronous interaction between a client and a service. A client requests arrangements for a particular trip by sending the travel service all pertinent information (most likely in an XML document). Based on the document's content, the service performs such steps as verifying the user's account, checking and getting authorization for the credit card, checking accommodations and transportation availability, building an itinerary, purchasing tickets, and so forth. Since the travel service must perform a series of often time-consuming steps in its normal workflow, the client cannot afford to pause and wait for these steps to complete.

Figure 3.7 shows one recommended approach for asynchronously delegating these types of Web service requests to the processing layer. In this architecture, the client sends a request to the service endpoint. The service endpoint validates the incoming request in the interaction layer and then delegates the client's request to the appropriate processing layer of the service. It does so by sending the request as a JMS message to a JMS queue or topic specifically designated for this type of request.

❒ Delegating a request to the processing layer through JMS *before* validating the request should be avoided.

Validation ensures that a request is correct. Delegating the request before validation may result in passing an invalid request to the processing layer, making error tracking and error handling overly complex. After the request is successfully delegated to the processing layer, the service endpoint may return a correlation identifier to the client. This correlation identifier is for the client's future reference and may help the client associate a response that corresponds to its previous request. If the business logic is implemented using enterprise beans, message-driven beans in the EJB tier read the request and initiate processing so that a response can ultimately be formulated.

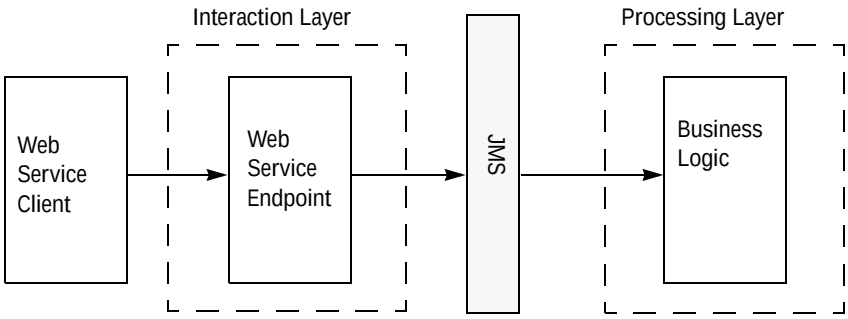

Figure 3.7 Asynchronous Interaction Between Client and Service

Figure 3.8 shows how the travel agency service might implement this interaction, and Code Example 3.16 shows the actual code that might be used.

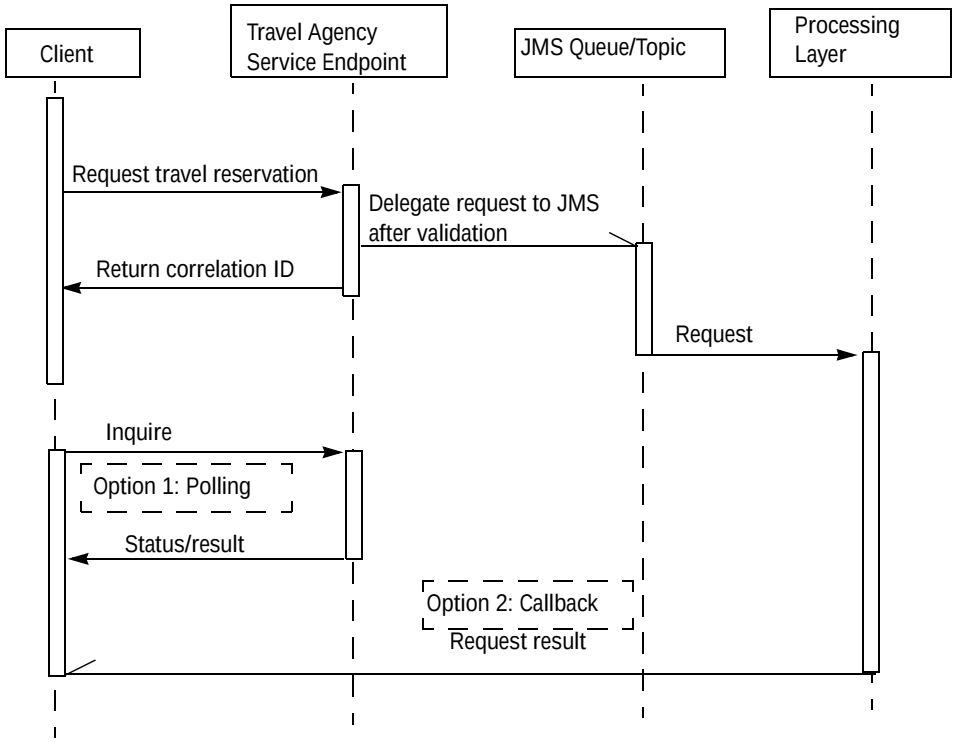

Figure 3.8 Travel Agency Service Interaction

In Figure 3.8, the vertical lines represent the passage of time, from top to bottom. The vertical rectangular boxes indicate when the entity (client or service) is busy processing the request or waiting for the other entity to complete processing. The half arrow type indicates asynchronous communication and the dashed vertical line indicates that the entity is free to work on other things while a request is being processed.

```
public class ReservationRequestRcvr {
    public ReservationRequestRcvr() throws RemoteException {....}

    public String receiveRequest(Source reservationDetails) throws
                        RemoteException, InvalidRequestException{

        /** Validate incoming XML document **/
        String xmlDoc = getDocumentAsString(reservationDetails);
        if(!validDocument(xmlDoc))
            throw new InvalidRequestException(...);

        /** Get a JMS Queue and delegate the incoming request **/
        /** to the queue **/
        QueueConnectionFactory queueFactory =
            serviceLocator.getQueueConnectionFactory(....);
        Queue reservationRequestQueue =
                    serviceLocator.getQueue(...);
        QueueConnection connection =
            queueFactory.createQueueConnection();
        QueueSession session = connection.createQueueSession(false,
                        Session.AUTO_ACKNOWLEDGE);
        QueueSender queueSender = session.createSender(queue);
        TextMessage message = session.createTextMessage();
        message.setText(xmlDoc);
        queueSender.send(message);
        /** Generate and return a correlation identifier **/
        return generateCorrelationID();
    }
}
```

Code Example 3.16 Implementing Travel Agency Service Interaction

One question remains: How does the client get the final result of its request? The service may make the result of the client's request available in one of two ways:

- The client that invoked the service periodically checks the status of the request using the correlation identifier that was provided at the time the request was submitted. This is also known as polling, and it appears as Option 1 in Figure 3.8.
- Or, if the client itself is a Web service peer, the service calls back the client's service with the result. The client may use the correlation identifier to relate the response with the original request (Option 2 in Figure 3.8).

Often this is decided by the nature of the service itself. For example, if the service runs a business process workflow, the workflow requires the service to take appropriate action after processing the request.

3.4.4 Formulating Responses

After you delegate the request to the business logic portion of the application, and the business logic completes its processing, you are ready for the next step: to form the response to the request.

❒ You should perform response generation, which is simply constructing the method call return values and output parameters, on the interaction layer, as close as possible to the service endpoint.

This permits having a common location for response assembly and XML document transformations, particularly if the document you return to the caller must conform to a different schema from the internal schema. Keeping this functionality near the endpoint lets you implement data caching and avoid extra trips to the processing layer. (See Figure 3.9.).

Consider response generation from the weather information service's point-of-view. The weather information service may be used by a variety of client types, from browsers to rich clients to handheld devices. A well-designed weather information service would render its responses in formats suitable for these different client types.

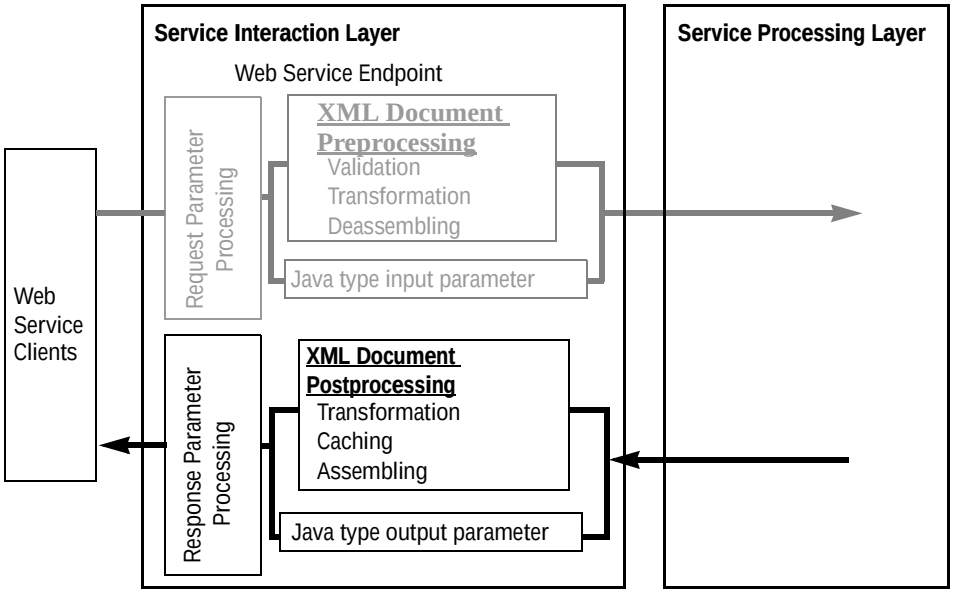

Figure 3.9 Web Service Response Processing

However, it is not good design to have a different implementation of the service's logic for each client type. Rather, it is better to design a common business logic for all client types. Then, in the interaction layer, transform the results per client type for rendering. It is thus important to consider the above guidelines, especially when your service has a common processing logic but potentially has different response rendering needs to fit its varied client types.

3.5 Processing Layer Design

The processing layer is where the business logic is applied to a Web service request. Recall that Web service is an interoperable way to expose new or existing applications. Hence, regardless of the means you use to expose your application's functionality, the business logic design issues are the same. You must still design the business logic by considering such issues as using enterprise beans, exposing a local or a remote EJB interface model, using container-managed or bean-managed persistence, and so forth.

❒ The issues and considerations for designing an application's processing or business logic layer, such as whether to perform this logic in the Web or EJB tier, are the same whether or not you use a Web service.

We do not address these business logic design issues here, since much of this discussion has already been covered in the book *Designing Enterprise Applications with the J2EE™ Platform, Second Edition*, and you can refer to that book for general guidelines and recommendations. You should also refer to the BluePrints Web site at `http://java.sun.com/blueprints` for recommendations on designing an application's business processing logic.

In addition to these general guidelines, there are some specific issues to keep in mind when designing the processing layer of a Web service.

- **Keep the processing layer independent of the interaction layer**. By keeping the layers independent and loosely coupled, the processing layer remains generic and can support different types of clients, such as Web service clients, classic Web clients, and so forth. To achieve loose coupling between the layers, consider using delegate classes that encapsulate the access to business components.
- **Bind XML documents to Java objects in the interaction layer**. There are times when your Web service expects to receive from a client an XML document containing a complete request, but the service's business logic has no need to operate on the document. On these occasions, it is recommended that the interaction layer bind the XML document contents to Java objects before passing the request to the processing layer. Since the processing logic does not have to perform the XML-to-Java conversion, a single processing layer can support XML documents that rely on different schemas. This also makes it easy to support multiple versions of an XML schema.

Keep in mind that your processing logic can operate on the contents of an XML document received from a client. Refer to "Handling XML Documents in a Web Service" on page 105, which highlights issues to consider when you pass XML documents to your business processing logic.

Depending on your application scenario, your processing layer may be required to work with other Web service peers to complete a client's request. If so, your processing layer effectively becomes a client of another Web service. Refer to Chapter 5 for guidelines on Web service clients. In other circumstances, your

processing layer may have to interact with EISs. For these cases, refer to Chapter 6 for guidelines.

3.6 Publishing a Web Service

Up to now, this chapter has covered guidelines for designing and implementing your Web service. Just as important, your Web service needs to be accessible to its intended clients. Recall that some Web services are intended for use by the general public. Other Web services are intended to be used only between trusted business partners (inter-enterprise), and still others are intended for use just within an enterprise (intra-enterprise).

Regardless of whether a service is to be accessible to the public, other enterprises, or even within a single enterprise, you must first make the details about the Web service—its interface, parameters, where the service is located, and so forth—accessible to clients. You do so by making a description of the Web service available to interested parties. As noted in "Web Services Description Language" on page 36, WSDL is the standard language for describing a service. Making this WSDL description available to clients enables them to use the service.

Once the WSDL is ready, you have the option to publish it in a registry. The next section describes when you might want to publish the WSDL in a registry. If you make the WSDL description of your service available in a public registry, then a Java-based client can use the JAXR APIs to look up the description of your service and then use the service. For that matter, a client can use the same JAXR APIs to look up the description of any Web service with an available WSDL description. This section examines registries from the point of view of a service developer.

3.6.1 Publishing a Service in a Registry

Publishing a service in a registry is one method of making the service available to clients. If you decide to publish your service in a registry, you decide on the type of registry to use based on the likely usage scenarios for your service. Registries run the gamut from public registries to corporate registries available only within a single enterprise.

- You may want to register Web services for general public consumption on a well-known public registry.

When you make your service available through a public registry, you essentially open the service's accessibility to the widest possible audience. When a service is registered in a public registry, any client, even one with no prior knowledge of the service, may look up and use the service. Keep in mind that the public registry holds the Web service description, which consists not only of the service's WSDL description but also any XML schemas referenced by the service description. In short, your Web service must publish its public XML schemas and any additional schemas defined in the context of the service. You also *must* publish on the same public registry XML schemas referred to by the Web service description.

- When a Web service is strictly for intra-enterprise use, you may publish a Web service description on a corporate registry within the enterprise.
- You do not need to use a registry if all the customers of your Web services are dedicated partners and there is an agreement among the partners on the use of the services. When this is the case, you can publish your Web service description—the WSDL and referenced XML schemas—at a well-known location with the proper access protections.

3.6.2 Understanding Registry Concepts

When considering whether to publish your service via a registry, it is important to understand some of the concepts, such as repositories and taxonomies, that are associated with registries.

Public registries are not repositories. Rather than containing complete details on services, public registries contain only details about what services are available and how to access these services. For example, a service selling adventure packages cannot register its complete catalog of products. A registry can only store the type of service, its location, and information required to access the service. A client interested in a service must first discover the service from the registry and then bind with the service to obtain the service's complete catalog of products. Once it obtains the service's catalog, the client can ascertain whether the particular service meets its needs. If not, the client must go back to the registry and repeat the discovery and binding process—the client looks in the registry for some other service that potentially offers what it wants, binds to that service, obtains and assesses its catalog, and so forth. Since this process, which is not insignificant, may have to be repeated several times, it is easy to see that it is important to register a service under its proper taxonomy.

❒ Register a service under the proper taxonomy.

It is important to register your service under the proper taxonomies. When you want to publish your service on a registry, either a public or corporate registry, you must do so against a taxonomy that correctly classifies or categorizes your Web service. It is important to decide on the proper taxonomy, as this affects the ease with which clients can find and use your service. Several well-defined industry standard taxonomies exist today, such as those defined by organizations such as the North American Industry Classification System (NAICS).

Using existing, well-known taxonomies gives clients of your Web service a standard base from which to search for your service, making it easy for clients to find your service. For example, suppose your travel business provides South Sea island-related adventure packages as well as alpine or mountaineering adventures. Rather than create your own taxonomy to categorize your service, clients can more easily find your service if you publish your service description using two different standard taxonomies: one taxonomy for island adventures and another for alpine and mountaineering adventures.

You can publish your Web service in more than one registry. To further help clients find your service, it is also a good idea to publish in as many applicable categories as possible. For example, a travel business selling adventure packages might register using a product category taxonomy as well as a geographical taxonomy. This gives clients a chance to use optimal strategies for locating a service. For example, if multiple instances of a service exist for a particular product, the client might further refine its selection by considering geographical location and choosing a service close to its own location. Using the travel business service as an example, such a service might register under the taxonomies for types of adventure packages (island and mountaineering), as well as under the taxonomies for the locales in which the adventure packages are provided (Mount Kilimanjaro or Tahiti), thus making it as easy as possible for a prospective client to locate its services.

3.6.3 Registry Implementation Scenarios

Once you decide to publish your service and establish the taxonomies that best identify your service, you are ready to implement your decisions. Before doing so, you may find it helpful to examine some of the registry implementation scenarios that you may encounter.

When a registry is used, we have seen that the service provider publishes the Web service description on a registry and clients discover and bind to the Web service to use its services. In general, a client must perform three steps to use a Web service:

1. The client must determine how to access the service's methods, such as determining the service method parameters, return values, and so forth. This is referred to as discovering the service definition interface.
2. The client must locate the actual Web service; that is, find the service's address. This is referred to as discovering the service implementation.
3. The client must be bound to the service's specific location, and this may occur on one of three occasions:
   - When the client is developed (called static binding)
   - When the client is deployed (also called static binding)
   - During runtime (called dynamic binding)

These three steps may produce three scenarios. The particular scenario depends on when the binding occurs and whether the client is implemented solely for a specific service or is a generic client. The following paragraphs describe these scenarios. (See Table 3.2 for a summary.) They also note important points you should consider when designing and implementing a Web service. (Chapter 5 considers these scenarios from the point of view of a client.)

- Scenario 1: The Web service has an agreement with its partners and publishes its WSDL description and referenced XML schemas at a well-known, specified location. It expects its client developers to know this location. When this is the case, the client is implemented with the service's interface in mind. When it is built, the client is already designed to look up the service interface directly rather than using a registry to find the service.

- Scenario 2: Similar to scenario 1, the Web service publishes its WSDL description and XML schemas at a well-known location, and it expects its partners to either know this location or be able to discover it easily. Or, when the partner is built, it can use a tool to dynamically discover and then include either the service's specific implementation or the service's interface definition, along with its specific implementation. In this case, binding is static because the partner is

built when the service interface definition and implementation are already known to it, even though this information was found dynamically.

- Scenario 3: The service implements an interface at a well-known location, or it expects its clients to use tools to find the interface at build time. Since the Web service's clients are generic clients—they are not clients designed solely to use this Web service—you must design the service so that it can be registered in a registry. Such generic clients dynamically find a service's specific implementation at runtime using registries. Choose the type of registry for the service—either public, corporate, or private—depending on the types of its clients—either general public or intra-enterprise—its security constraints, and so forth.

Table 3.2 Discovery-Binding Scenarios for Clients

Scenarios	Discover Service Interface Definition	Discover Service Implementation	Binding to Specific Location
1	None	None	Static
2	None or dynamic at build time	Dynamic at build time	Static
3	None or dynamic at build time	Dynamic at runtime	Dynamic at build time

3.7 Handling XML Documents in a Web Service

Up to now, this chapter addressed issues applicable to all Web service implementations. There are additional considerations when a Web service implementation expects to receive an XML document containing all the information from a client, and which the service uses to start a business process to handle the request. There are several reasons why it is appropriate to exchange documents:

- Documents, especially business documents, may be very large, and as such, they are often sent as a batch of related information. They may be compressed independently from the SOAP message.
- Documents may be legally binding business documents. At a minimum, their original form needs to be conserved through the exchange and, more than likely, they may need to be archived and kept as evidence in case of disagreement.

For these documents, the complete infoset of the original document should be preserved, including comments and external entity references (as well as the referred entities).

- Some application processing requires the complete document infoset, including comments and external entity references. As with the legally binding documents, it is necessary to preserve the complete infoset, including comments and external entity references, of the original document.
- When sent as attachments, it is possible to handle documents that may conform to schemas expressed in languages not supported by the Web service endpoint or that are prohibited from being present within a SOAP message infoset (such as the Document Type Declaration `<!DOCTYPE>` for a DTD-based schema).

For example, consider the travel agency Web service, which typically receives a client request as an XML document containing all information needed to arrange a particular trip. The information in the document includes details about the customer's account, credit card status, desired travel destinations, preferred airlines, class of travel, dates, and so forth. The Web service uses the documents contents to perform such steps as verifying the customer's account, obtaining authorization for the credit card, checking accommodations and transportation availability, building an itinerary, and purchasing tickets.

In essence, the service, which receives the request with the XML document, starts a business process to perform a series of steps to complete the request. The contents of the XML document are used throughout the business process. Handling this type of scenario effectively requires some considerations in addition to the general ones for all Web services.

- ❒ Good design expects XML documents to be received as `javax.xml.transform.Source` objects. See "Exchanging XML Documents" on page 107, which discusses exchanging XML documents as parameters. Keep in mind the effect on interoperability (see "Interoperability" on page 86).
- ❒ It is good design to do the validation and any required transformation of the XML documents as close to the endpoint as possible. Validation and transformation should be done before applying any processing logic to the document content. See Figure 3.4 and the discussion on receiving requests in "Receiving Requests" on page 89.

- It is important to consider the processing time for a request and whether the client waits for the response. When a service expects an XML document as input and starts a lengthy business process based on the document contents, then clients typically do not want to wait for the response. Good design when processing time may be extensive is to delegate a request to a JMS queue or topic and return a correlation identifier for the client's future reference. (Recall Figure 3.7 on page 96 and its discussion.)

The following sections discuss other considerations.

3.7.1 Exchanging XML Documents

As noted earlier, there are times when you may have to exchange XML documents as part of your Web service and such documents are received as parameters of a method call. The J2EE platform provides three ways to exchange XML documents.

The first option is to use the Java-MIME mappings provided by the J2EE platform. See Table 3.1 on page 75. With this option, the Web service endpoint receives documents as `javax.xml.transform.Source` objects. (See Code Example 3.3 on page 75.) Along with the document, the service endpoint can also expect to receive other JAX-RPC arguments containing metadata, processing requirements, security information, and so forth. When an XML document is passed as a `Source` object, the container automatically handles the document as an attachment—effectively, the container implementation handles the document-passing details for you. This frees you from the intricacies of sending and retrieving documents as part of the endpoint's request/response handling.

- Passing XML documents as `Source` objects is the most effective option in a completely Java-based environment (one in which all Web service clients are based on Java). However, sending documents as `Source` objects may not be interoperable with non-Java clients. (As already noted in the section "Interoperability" on page 86, standard ways to exchange attachments are currently being formulated. Future versions of the J2EE platform will incorporate these standards once they are final.)

The second option is to design your service endpoint such that it receives documents as `String` types. Code Example 3.17 shows the WSDL description for a service that receives documents as String types, illustrating how the WSDL maps the XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
    <types/>
    <message name="PurchaseOrderService_submitPurchaseOrder">
        <part name="PurchaseOrderXMLDoc" type="xsd:string"/>
    </message>
    <message
            name="PurchaseOrderService_submitPurchaseOrderResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <portType name="PurchaseOrderService">
        <operation name="submitPurchaseOrder"
                          parameterOrder="PurchaseOrderXMLDoc">
            <input
            message="tns:PurchaseOrderService_submitPurchaseOrder"/>
            <output message=
            "tns:PurchaseOrderService_submitPurchaseOrderResponse"/>
        </operation>
    </portType>
    ...
</definitions>
```

Code Example 3.17 Mapping XML Document to `xsd:string`

Code Example 3.18 shows the equivalent Java interface for the WSDL shown in Code Example 3.17.

```
public interface PurchaseOrderService extends Remote {
    public String submitPurchaseOrder(String poDocument)
            throws RemoteException, InvalidOrderException;
}
```

Code Example 3.18 Receiving an XML Document as a `String` object

If you are developing your service using the Java-to-WSDL approach, and the service must exchange XML documents and be interoperable with clients on any platform, then passing documents as `String` objects may be your only option.

- There may be a performance drawback to sending an XML document as a `String` object: As the document size grows, the `String` equivalent size of the document grows as well. As a result, the payload size of the message you send also grows. In addition, the XML document loses its original format since sending a document as a `String` object sends it in a canonical format.

The third option is to exchange the XML document as a SOAP document fragment. With this option, you map the XML document to `xsd:anyType` in the service's WSDL file.

- It is recommended that Web services exchange XML documents as SOAP document fragments because passing XML documents in this manner is both portable across J2EE implementations and interoperable with all platforms.
- To pass SOAP document fragments, you must implement your service using the WSDL-to-Java approach.

For example, the travel agency service receives an XML document representing a purchase order that contains all details about the customer's preferred travel plans. To implement this service, you define the WSDL for the service and, in the WSDL, you map the XML document type as `xsd:anyType`. See Code Example 3.19.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
    <types/>
    <message name="PurchaseOrderService_submitPurchaseOrder">
        <part name="PurchaseOrderXMLDoc" type="xsd:anyType"/>
    </message>
    <message
           name="PurchaseOrderService_submitPurchaseOrderResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <portType name="PurchaseOrderService">
        <operation name="submitPurchaseOrder"
                        parameterOrder="PurchaseOrderXMLDoc">
            <input
            message="tns:PurchaseOrderService_submitPurchaseOrder"/>
            <output message=
```

```
            "tns:PurchaseOrderService_submitPurchaseOrderResponse"/>
        </operation>
    </portType>
    ...
</definitions>
```

Code Example 3.19 Mapping XML document to `xsd:anyType`

A WSDL mapping of the XML document type to `xsd:anyType` requires the platform to map the document parameter as a `javax.xml.soap.SOAPElement` object. For example, Code Example 3.20 shows the Java interface generated for the WSDL description in Code Example 3.19.

```
public interface PurchaseOrderService extends Remote {
    public String submitPurchaseOrder(SOAPElement
                        purchaseOrderXMLDoc) throws RemoteException;
}
```

Code Example 3.20 Java Interface for WSDL in Code Example 3.19

In this example, the `SOAPElement` parameter in `submitPurchaseOrder` represents the SOAP document fragment sent by the client. For the travel agency service, this is the purchase order. The service can parse the received SOAP document fragment using the `javax.xml.soap.SOAPElement` API. Or, the service can use JAXB to map the document fragment to a Java `Object` or transform it to another schema. A client of this Web service builds the purchase order document using the client platform-specific API for building SOAP document fragments—on the Java platform, this is the `javax.xml.soap.SOAPElement` API—and sends the document as one of the Web service's call parameters.

When using the WSDL-to-Java approach, you can directly map the document to be exchanged to its appropriate schema in the WSDL. The corresponding generated Java interface represents the document as its equivalent Java `Object`. As a result, the service endpoint never sees the document that is exchanged in its original document form. It also means that the endpoint is tightly coupled to the document's schema: Any change in the document's schema requires a corresponding change to the endpoint. If you do not want such tight coupling, consider using `xsd:anyType` to map the document.

3.7.2 Separating Document Manipulation from Processing Logic

When your service's business logic operates on the contents of an incoming XML document, the business processing logic must at a minimum read the document, if not modify the document. By separating the document manipulation logic from the processing logic, a developer can switch between various document manipulation mechanisms without affecting the processing logic. In addition, there is a clear division between developer skills.

❒ It is a good practice to separate the XML document manipulation logic from the business logic.

The "Abstracting XML Processing from Application Logic" section on page 155 provides more information on how to accomplish this separation and its merits.

3.7.3 Fragmenting XML Documents

When your service's business logic operates on the contents of an incoming XML document, it is a good idea to break XML documents into logical fragments when appropriate. When the processing logic receives an XML document that contains all information for processing a request, the XML document usually has well-defined segments for different entities, and each segment contains the details about a specific entity.

❒ Rather than pass the entire document to different components handling various stages of the business process, it's best if the processing logic breaks the document into fragments and passes only the required fragments to other components or services that implement portions of the business process logic.

See "Fragmenting Incoming XML Documents" on page 153 for more details on fragmentation.

3.7.4 Using XML

XML, while it has many benefits, also has performance disadvantages. You should weigh the trade-offs of passing XML documents through the business logic processing stages. The pros and cons of passing XML documents take on greater significance when the business logic implementation spans multiple containers. Refer to

Chapter 5, specifically the section entitled "Use XML Judiciously" on page 194, which provides guidelines on this issue. Following these guidelines may help minimize the performance overhead that comes with passing XML documents through workflow stages.

Also, when deciding on an approach, keep in mind the costs involved for using XML and weigh them along with the recommendations on parsing, validation, and binding documents to Java objects. See Chapter 4 for a discussion of these topics.

3.7.5 Using JAXM and SAAJ Technologies

The J2EE platform provides an array of technologies—including mandatory technologies such as JAX-RPC and SAAJ and optional technologies such as Java™ API for XML Messaging (JAXM)—that enable message and document exchanges with SOAP. Each of these J2EE technologies offers a different level of support for SOAP-based messaging and communication. (See Chapter 2 for the discussion on JAX-RPC and SAAJ.)

An obvious question that arises is: Why not use JAXM or SAAJ technologies in scenarios where you have to pass XML documents? If you recall:

- SAAJ lets developers deal directly with SOAP messages, and is best suited for point-to-point messaging environments. SAAJ is better for developers who want more control over the SOAP messages being exchanged and for developers using handlers.
- JAXM defines an infrastructure for guaranteed delivery of messages. It provides a way of sending and receiving XML documents and guaranteeing their receipt, and is designed for use cases that involve storing and forwarding XML documents and messages.

SAAJ is considered more useful for advanced developers who thoroughly know the technology and who must deal directly with SOAP messages.

Using JAXM for scenarios that require passing XML documents may be a good choice. Note, though, that JAXM is optional in the J2EE 1.4 platform. As a result, a service developed with JAXM may not be portable. When you control both end points of a Web service, it may make more sense to consider using JAXM.

3.8 Deploying and Packaging a Service Endpoint

Up to now, we have examined Web services on the J2EE platform in terms of design, development, and implementation. Once you complete the Web services implementation, you must write its deployment descriptors, package the service with all its components, and deploy the service.

❒ Developers should, if at all possible, use tools or IDEs to develop a Web service. These Web service development tools and IDEs automatically create the proper deployment descriptors for the service and correctly handle the packaging of the service—steps necessary for a service to operate properly. Furthermore, tools and IDEs hide these details from the developer.

Although you can expect your development tool to perform these tasks for you, it is good to have a conceptual understanding of the J2EE 1.4 platform deployment descriptor and packaging structure, since they determine how a service is deployed on a J2EE server and the service's availability to clients. This section, which provides a conceptual overview of the deployment and packaging details, is not essential reading. Nonetheless, you may find it worthwhile to see how these details contribute to portable, interoperable Web services.

3.8.1 Service Information in the Deployment Descriptors

To successfully deploy a service, the developer provides the following information.

- Deployment-related details of the service implementation, including the Web service interface, the classes that implement the Web service interface, and so forth.
- Details about the Web services to be deployed, such as the ports and mappings
- Details on the WSDL port-to-port component relationship

More specifically, the deployment descriptor contains information about a service's port and associated WSDL. Recall from "Web Service Technologies Integrated in J2EE Platform" on page 49:

- A port component (also called a port) gives a view of the service to clients such that the client need not worry about how the service has been implemented.
- Each port has an associated WSDL.
- Each port has an associated service endpoint (and its implementation). The endpoint services all requests that pass through the location defined in the WSDL port address.

To begin, the service implementation declares its deployment details in the appropriate module-specific deployment descriptors. For example, a service implementation that uses a JAX-RPC service endpoint declares its details in the `WEB-INF/web.xml` file using the `servlet-class` element. (See Code Example 3.21.)

```
<web-app ...>
    ...
    <servlet>
        <description>Endpoint for Some Web Service</description>
        <display-name>SomeWebService</display-name>
        <servlet-name>SomeService</servlet-name>
        <servlet-class>com.a.b.c.SomeServiceImpl</servlet-class>
        <load-on-startup>0</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>SomeService</servlet-name>
        <url-pattern>/webservice/SomeService</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Code Example 3.21 `web.xml` File for a JAX-RPC Service Endpoint

Note that when you have a service that functions purely as a Web service using JAX-RPC service endpoints, some specifications in the `web.xml` file, such as `<error-page>` and `<welcome-file-list>`, have no effect.

A service implementation that uses an EJB service endpoint declares its deployment details in the file `META-INF/ejb-jar.xml` using the `session` element. (See Code Example 3.22.)

```
<ejb-jar ...>
    <display-name>Some Enterprise Bean</display-name>
    <enterprise-beans>
        <session>
            <ejb-name>SomeBean</ejb-name>
            <service-endpoint>com.a.b.c.SomeIntf</service-endpoint>
            <ejb-class>com.a.b.c.SomeServiceEJB</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
    ...
</ejb-jar>
```

Code Example 3.22 `ejb-jar.xml` File for an EJB Service Endpoint

Next, the details of the port are specified. The Web service deployment descriptor, called `webservices.xml`, defines and declares the structural details for the port of a Web service. This file contains the following information:

- A logical name for the port that is also unique among all port components (`port-component-name` element)
- The service endpoint interface for the port (`service-endpoint-interface` element)
- The name of the class that implements the service interface (`service-impl-bean` element)
- The WSDL file for the service (`wsdl-file` element)
- A `QName` for the port (`wsdl-port` element)
- A correlation between WSDL definitions and actual Java interfaces and definitions using the mapping file (`jaxrpc-mapping-file` element)
- Optional details on any handlers

The reference to the service implementation bean, specified using the `service-impl-bean` element in `webservices.xml`, is either a `servlet-link` or an `ejb-link` depending on whether the endpoint is a JAX-RPC or EJB service end-

point. This link element associates the Web service port to the actual endpoint implementation defined in either the `web.xml` or `ejb-jar.xml` file.

The JAX-RPC mapping file, which is specified using the `jaxrpc-mapping-file` element in `webservices.xml`, keeps details on the relationships and mappings between WSDL definitions and corresponding Java interfaces and definitions. The information contained in this file, along with information in the WSDL, is used to create stubs and ties for deployed services.

Thus, the Web services deployment descriptor, `webservices.xml`, links the WSDL port information to a unique port component and from there to the actual implementation classes and Java-to-WSDL mappings. Code Example 3.23 is an example of the Web services deployment descriptor for our sample weather Web service, which uses a JAX-RPC service endpoint.

```
<webservices ...>
    <description>Web Service Descriptor for weather service
    </description>
    <webservice-description>
        <webservice-description-name>
            WeatherWebService
        </webservice-description-name>
        <wsdl-file>
            WEB-INF/wsdl/WeatherWebService.wsdl
        </wsdl-file>
        <jaxrpc-mapping-file>
            WEB-INF/WeatherWebServiceMapping.xml
        </jaxrpc-mapping-file>
        <port-component>
            <description>port component description</description>
            <port-component-name>
                WeatherServicePort
            </port-component-name>
            <wsdl-port xmlns:weatherns="urn:WeatherWebService">
                weatherns:WeatherServicePort
            </wsdl-port>
            <service-endpoint-interface>
                endpoint.WeatherService
            </service-endpoint-interface>
            <service-impl-bean>
                <servlet-link>WeatherService</servlet-link>
```

```
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

Code Example 3.23 Weather Web Service Deployment Descriptor

3.8.2 Package Structure

Once the service implementation and deployment descriptors are completed, the following files should be packaged into the appropriate J2EE module:

- The WSDL file
- The service endpoint interface, including its implementation and dependent classes
- The JAX-RPC mapping file, which specifies the package name containing the generated runtime classes and defines the namespace URI for the service. See Code Example 5.21 on page 242.
- The Web service deployment descriptor

The type of endpoint used for the service implementation determines the type of the J2EE module to use.

❒ The appropriate J2EE module for a service with a JAX-RPC service endpoint is a WAR file. A service using an EJB service endpoint must be packaged in an EJB-JAR file.

The package structure is as follows:

- WSDL files are located relative to the root of the module.
- The service interface, the service implementation classes, and the dependent classes are packaged just like any other J2EE component.
- The JAX-RPC mapping file is located relative to the root of the module (typically in the same place as the module's deployment descriptor).

- The Web service deployment descriptor location depends on the type of service endpoint, as follows:
  - For an EJB service endpoint, the Web service deployment descriptor is packaged in an EJB-JAR in the META-INF directory as `META-INF/webservice.xml`
  - For a JAX-RPC service endpoint, the deployment descriptor is packaged in a WAR file in the WEB-INF directory as `WEB-INF/webservices.xml`.

See Figure 3.10, which shows a typical package structure for a Web service using an EJB endpoint. Figure 3.11 shows the typical structure for a Web service using a JAX-RPC endpoint.

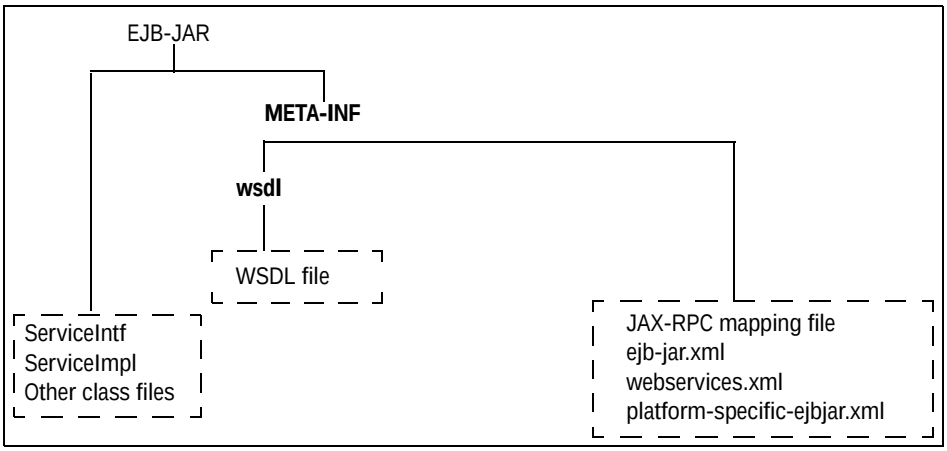

Figure 3.10 Package Structure for EJB Endpoint

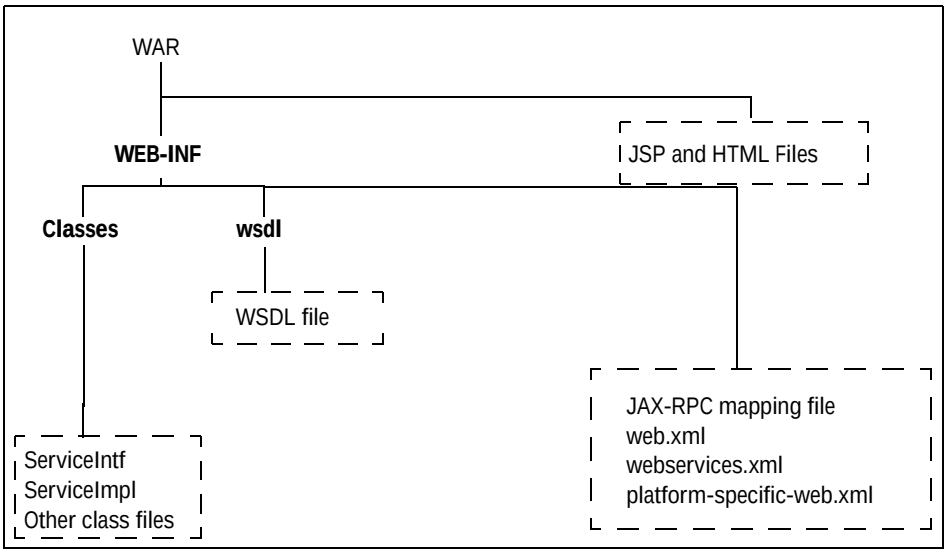

Figure 3.11 Package Structure for JAX-RPC Service Endpoint

3.9 Conclusion

This chapter began with a description of Web service fundamentals. It described the underlying flow of a typical Web service on the Java platform, showing how the various components making up clients and services pass requests and responses among themselves. The chapter also described some example scenarios, which it used to illustrate various concepts. Once the groundwork was set, the chapter discussed the key design decisions that a Web service developer needs to make, principally the design of a service as an interaction and a processing layer. It traced how to go about making design decisions and recommending good design choices for specific scenarios.

The next chapter focuses on developing Web service clients.