CHAPTER 4

Client Design

WEB services take the Web client model to the next level. Developers can write far more powerful clients whose interaction with Web services provide a rich client experience. In this environment, client developers need not have control over the server portion of an application, yet they can still write powerful, rich client applications. This chapter focuses on using the Java platform to design and develop Web services-based clients.

Clients can take advantage of Web services to obtain a wide range of functions or services. To a client, a Web service is a black box: the client does not have to know how the service is implemented or even who provides it. The client only cares about the functionality—the service—provided by the Web service. Examples of Web services include order tracking services, information lookup services, and credit card validation services. Various clients running on different types of platforms can all access these Web services.

One of the principal reasons for implementing Web services is to achieve interoperability. Clients can access Web services regardless of the platform or operating system upon which the service or the client is implemented. Not only is the service's platform of no concern to the client, the client's implementation language is completely independent of the service. In a very real sense, a Web service is a black box to a client.

Web service clients can take many forms, from full-blown J2EE applications, to rich client applications, even to light-weight application clients, such as wireless devices. In short, there are many different types of clients that can talk to Web services. Although developers can write clients in virtually any contemporary programming language, the Java programming language is often the preferred language, as we demonstrate here. Regardless of the language used, the same principles apply to designing a client. In addition to interoperability, Web service clients can use standardized approaches to access services through firewalls. Such access extends the capabilities of clients. Clients accessing Web services also remain more loosely coupled to the service.

We start by looking at the different means for communicating with Web services and applications, and then examine some typical scenarios where using Web services-based clients make sense. From there, the chapter discusses the different types of clients that use Web services, particularly highlighting the different design considerations for the principal Java clients: J2EE, J2SE, and J2ME clients. Then, the chapter addresses the steps for developing these client applications, from locating a service through handling errors. It especially focuses on the communication approaches and technologies that client applications can employ to access a service. Included in this discussion are recommendations and guidelines to help developers choose the optimal approach. The chapter concludes with guidelines for packaging client applications.

4.1 Web Service Communication Styles

Web services are only one of several ways for a client to access an application service. For example, Java applications may access application services using RMI/IIOP, JMS, XML over HTTP, or Web services. There are advantages and disadvantages with each of these communication technologies, and the developer must weigh these considerations when deciding on the client application design.

Interoperability is the primary advantage for using Web services as a means of communication. Web services give clients the ability to interoperate with almost any type of system and application, regardless of the platform on which the system or application runs. In addition, the client can use a variety of technologies for this communication. Furthermore, different client types—such as hand-held devices, desktop browsers, or rich GUI clients—running on different platforms and written in different languages may be able to access the same set of services. Some applications may be designed such that their functionality is only be accessible via a Web service.

The transport protocol used by Web services enable clients to operate with systems through firewalls. The service’s WSDL document enables clients and services that use very different technologies to map and convert their respective data objects. For services and clients that are based on JAX-RPC, the JAX-RPC runtime handles this mapping transparently.

When using Web services for communication, two scenarios should be considered.

1. The client developer needs to access a Web service, and must decide on the best type of client with which to access the service and process the results.
2. The client developer is also the Web service endpoint developer. In this situation, the developer should design the service interface, keeping in mind the interoperability needs of other potential types of clients who may need access to the service.

Let’s look at the different communication approaches that are available to a J2EE client to access a service, including JAX-RPC-generated classes, RMI/IIOP, Java Message Service, and XML document exchange over HTTP. Clients can easily use the JAX-RPC-generated stub classes to access a Web service. Although not as fast from a performance perspective as other technologies (such as RMI/IIOP), JAX-RPC gives clients greater flexibility and supports more types of clients.

J2EE application clients may also use RMI/IIOP to make remote calls over the network on application business logic. RMI/IIOP is often used for clients operating in intranet environments, where there is a greater degree of control over the client’s deployment and the J2EE server. While these controlled environments provide a client container that handles the communication security, passing through firewalls is problematic.

RMI/IIOP provides clients with secure access to the application business logic while at the same time taking care of the details of the client and server communication and marshalling and demarshalling parameters. However, RMI/IIOP does not provide an easy way to notify clients of updates to data, since it cannot easily contact client applications. As a result, clients concerned about the currency of their data, or who need to request data, must frequently poll the server.

Java Message Service (JMS) is another means for J2EE clients to communicate with server applications. JMS provides a means for asynchronous communication. Applications using JMS are better suited to a setting that is behind a firewall, since messaging systems generally do not work well on the Internet. (Often, messaging systems are not even exposed on the Internet.) Not only must developers have some knowledge of how to work with messaging systems, such as how to set up and use topics or queues, but the messaging system mechanisms must already be in place. Developers may also have to convert messages from non-Java formats to their application's object model.

Another valid means of communication is XML document exchange over HTTP. Communication using the HTTP protocol requires only a simple infrastructure to send and receive messages. However, the client application must be able to parse the XML documents representing the messages. (Parsing involves mapping the XML data to the client application's object model.) When using this means of communication, the developer at a minimum needs to write code to send and receive the documents over HTTP as well as to parse the document data. If such communication must also be secure, developers would have to include code that uses Secure Socket Layer (SSL), making the development task more difficult. However, this means of communication may be sufficient, particularly in a closed environment or when clients are applets. Care should be taken if using this approach, since it is not standard.

❐ Whenever possible, developers should use standard approaches for client-server communication, such as SOAP and Web services.

4.2 Scenarios for Web Services-Based Client Applications

Developers typically write clients to access pre-existing Web services—that is, public or private Web services provided by outside entities. At times, the same developers may simultaneously develop both the service and its clients. Regardless of this, client developers should rely on the WSDL service description for their knowledge of the service details. In addition, developers may use a variety of technologies and APIs for developing these clients. They may develop clients using J2EE technologies, often called J2EE clients, or they may use the standard Java (J2SE) technologies, or even non-Java technologies.

Before delving into the details of client development, let's examine several Web service client scenarios. Although different, each scenario is based on accessing the same Web service. The scenarios are as follows:

- J2EE component—In this scenario, a J2EE component accesses an order tracking Web service. The J2EE component receives results from the service, and it formats these results so that it can be read or displayed by a front-end browser.
- J2SE client—A J2SE client may access the same order tracking service as the J2EE component. However, the J2SE client provides a more detailed view of the data (purchase order objects) returned from the service.
- J2ME client—A J2ME client, such as a client application running on a mobile device or PDA, gives a user the freedom to access the same order tracking service from places other than his or her office. In addition, the user can work offline with the order details returned from the service.
- Non-Java client—A non-Java client accesses the same Web service using SOAP 1.1 over HTTP.

Figure 4.1 shows how these different types of clients might access the same purchase order tracking Web service interface. All clients, regardless of their type, rely on the Web service's WSDL document, which uses a standard format to describe the service's location and its operations. Clients need to know the information in the WSDL document to understand how to communicate with a particular service.

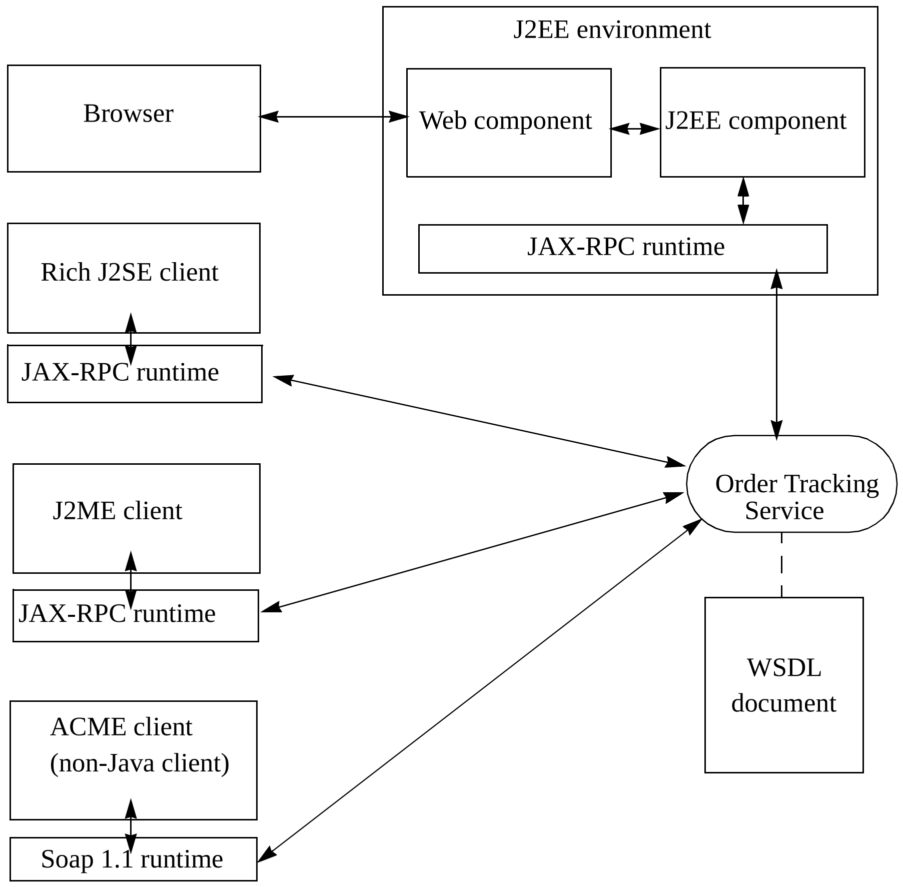

Figure 4.1 Web Service Clients in J2EE and Non-J2EE Environments

It is important to note that none of the clients communicate directly with a Web service. Each client type relies on a runtime—either a JAX-RPC or SOAP runtime—through which it accesses the service. From the developer's perspective, the client's use of a runtime is kept nearly transparent. However, good design dictates that a developer still modularize the Web service access code, plus consider issues related to remote calls and handling remote exceptions. These points are addressed later. (See "Locating and Accessing a Service" on page 143 and "Handling Exceptions" on page 155.)

In a J2EE environment, client applications may be browser based. Browser-based applications rely on J2EE components, which act as clients of the service.

These clients, referred to as J2EE clients, are J2EE components that access a Web component (such as a servlet or JSP) contained in the J2EE environment, and the JAX-RPC runtime in turn handles the communication with the Web service. The J2EE environment shields the developer from the communication details. J2EE clients run in either an EJB container or a Web container, and these containers manage the client environment.

Standalone clients—which may be J2SE clients, J2ME clients, or clients written in a language other than Java—communicate to the Web service through the JAX-RPC runtime or the SOAP 1.1 runtime. Standalone clients are outside the J2EE environment. Because they don't have a J2EE EJB container or Web container to manage the environment, standalone clients require more work from the developer.

Although each type of client works well, developers must deal with an increasing level of complexity if they work directly with the JAX-RPC or SOAP 1.1 runtimes. The advantage of the J2EE environment is that it shields developers from some of the complexity associated with developing Web services, such as the look up of the service and the life-cycle management of objects used to access a service. In the J2EE environment, a developer uses JNDI to look up a service in much the same way as he or she might use other Java APIs, such as JDBC or a JMS queue.

Given that many types of clients can access Web services, how do you determine which type of client is best for your application? To make this determination, you must consider how end users and the application expect to use the service. You should also weigh the pros and cons of each client type, and consider that each type of client has its own unique characteristics that may make it appropriate (or not appropriate) for a specific application task. Thus, you should consider the characteristics of the different client types as well as the communication mechanisms available to each client type when deciding which client type to use to access a Web service. The following general guidelines should help with this decision:

❐ **J2EE clients**—Browser-based clients implemented as J2EE clients have good access to Web services. As a further advantage, J2EE clients may also access Web services that exist on a different J2EE platform or even on a non-Java platform, provided that platform supports the SOAP 1.1 protocol and WSDL.

J2EE clients may also access Web services from within a work flow architecture, and they may aggregate Web services.

- ❐ **J2SE clients**—Generally, J2SE clients are best when you need to provide a rich interface or when you must manipulate large sets of data. J2SE clients may also work in a disconnected mode of operation.
- ❐ **J2ME clients**—J2ME clients are best for applications that require remote and immediate access to a Web service. J2ME clients may be restricted to a limited set of interface components. Like J2SE clients, J2ME clients may also work in a disconnected mode of operation.

Now that we have looked at some sample Web service scenarios, let's examine in more detail the different types of clients that use Web services.

4.3 Designing J2EE Clients

As already noted, the J2EE 1.4 platform provides an environment that supports client application access to Web services. A J2EE environment is an environment onto which clients are deployed and in which deployment descriptors declaratively define the Web services that these clients access. Because they are defined declaratively, the deployer may change these Web services at deployment. In addition, the J2EE platform handles the underlying work for creating and initializing the Web services, plus the client application can use various J2EE components to access the Web service.

The J2EE platform provides many other services in addition to supporting Web services. Developers can obtain a richer set of functionality by using these other services, such as Web and EJB components, in addition to Web services for their client applications.

For example, consider a client application that accesses an order tracking Web service via a Web component. (See Figure 4.1.) The client application presents a browser page to the end user, who enters an order identifier to a form displayed in its browser. The client application passes this order identifier to an order tracking servlet, which accesses the order tracking Web service and retrieves the order information. The Web component converts the retrieved information to an HTML page, and returns the formatted page to the client application for presentation to the end user. The WSDL document for the order tracking service specifies the details of the order identifier parameter and the retrieved information.

Figure 4.2 shows how J2EE components that reside in the Web tier can also access the same order tracking Web service. Such components include Web application frameworks and visualization frameworks, such as portlets and Java Server Faces components. A client application can freely use these J2EE components, but, if making numerous synchronous calls, there may be network-related latency issues.

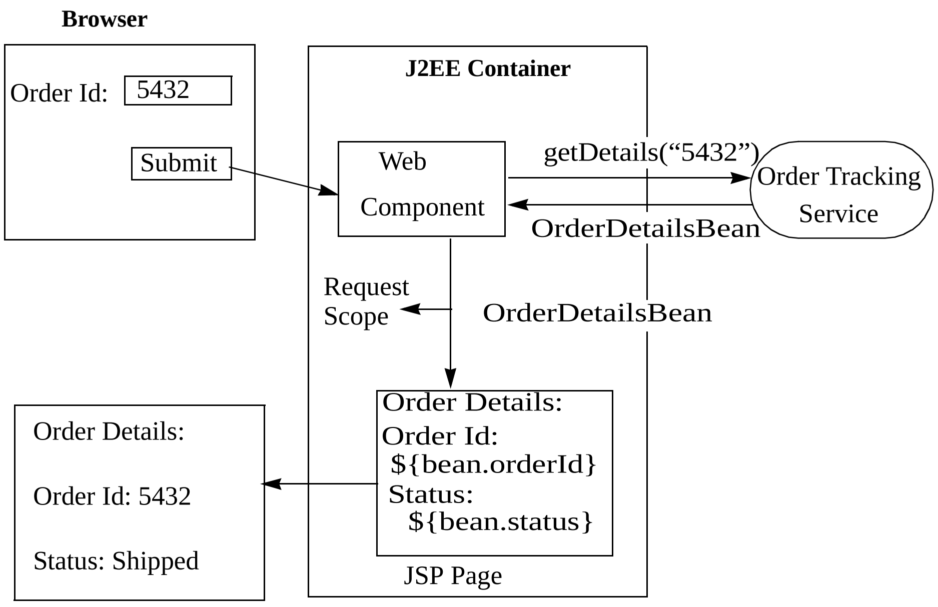

Figure 4.2 Web Tier Component Calling a Web Service

In this example, a Web component calls the order tracking service and, when it receives a response from the service, it puts the results in a Java Beans component (`OrderDetailsBean`) that is within the request scope. The Web component uses a JSP to generate an HTML response, which the container returns to the browser that made the original request. The order tracking service could also return the Java Beans component directly to a framework for further processing. If the service does this, then the JAX-RPC runtime ensures that the returned object is a Java Beans component (if the service endpoint interface return parameter is set properly).

It is also possible to write J2EE clients using EJB components. (See Figure 4.3.) These EJB components may themselves be Web service endpoints as well as clients of other Web services. Often, EJB components are used in a work flow to provide Web services with the additional support provided by an EBJ container—that is, transactional support, declarative security, threading, and life-cycle management. Although Web services themselves are not transactional at this time, they can attain some transactional capabilities when using EJB components.

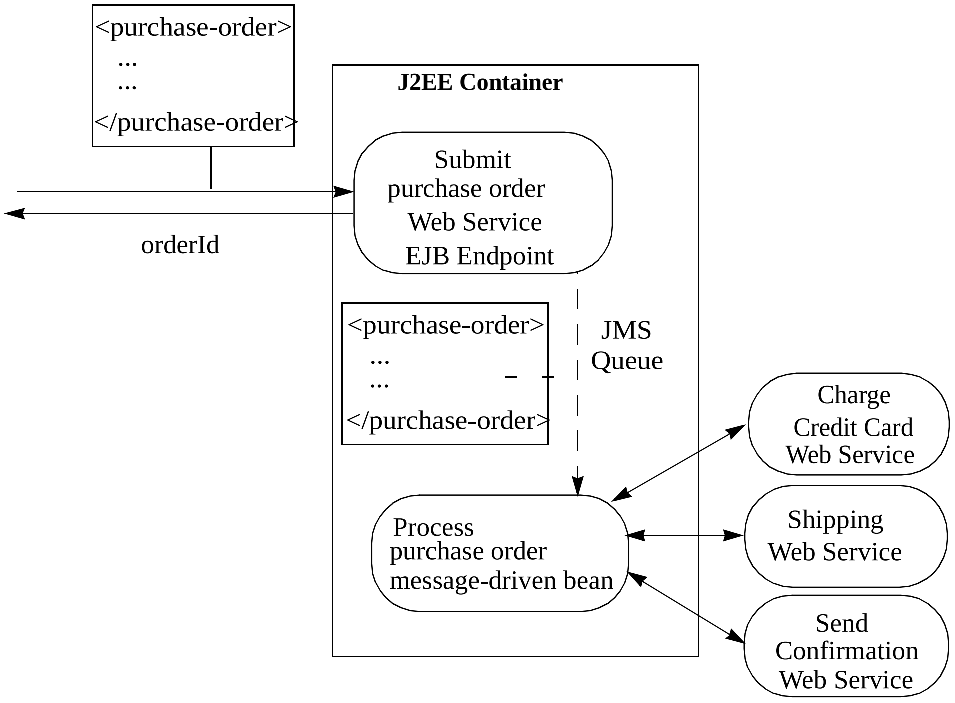
<purchase-order>
...
...
</purchase-order>
J2EE Container
Submit purchase order Web Service EJB Endpoint
orderId
<purchase-order>
...
...
</purchase-order>
JMS Queue
Process purchase order message-driven bean
Charge Credit Card Web Service
Shipping Web Service
Send Confirmation Web Service

Figure 4.3 EJB Components and Web Services in a Workflow

Figure 4.3 demonstrates a Web work flow scenario: a Web service endpoint used in combination with a message-driven bean component to provide a work flow operation that runs asynchronously once the initial service starts. The Web service endpoint synchronously returns an `orderId` to the calling application and then puts the purchase order in a JMS message queue. The message-driven bean listens for messages on the JMS queue. When one arrives, the bean retrieves the message and initiates the purchase order processing work flow. The purchase order is processed asynchronously while the Web service receives other purchase orders. In this example, the work flow consists of three additional stages performed by separate Web services: a credit card charging service, a shipping service, and a service that sends an order confirmation. The message-driven bean aggregates the three work flow stages. Other systems within an organization may provide these services and they may be shared by many applications.

The important points to keep in mind with J2EE clients are:

❒ A client developer should not circumvent the J2EE platform's management of a service. A client should not create or destroy a Web service port.

❒ A client developer should not access a Web service directly, thus circumventing the platform's processing of a service call. This may be a security risk.

❒ A client developer should not assume that the same port instance for the service is used for all calls to the service. Port instances are stateless, and the J2EE platform is not required to return a previously used port instance to a client.

4.4 Designing J2SE Clients

It is possible for a service to include client applications that do not run in a J2EE container. For example, a service for an order entry system might contain a rich interface that allows a user to physically enter and validate data prior to submitting the data to the service. Or, it might have a GUI client to track orders, where the GUI enhances the user's interaction with the data.

Unlike the J2EE environment, developers of non-J2EE clients are responsible for much of the underlying work to look up a service and to create and maintain instances of classes that access the service. Since they cannot rely on a container, these developers must create and manage their own services and ensure the availability of all runtime environments needed to access the Web services.

J2SE clients, often considered rich application clients, have the capability to do more processing and state management than other clients. This type of application client can provide a rich GUI application development environment that includes document editing and graphical manipulation. However, there are some considerations which should be kept in mind when using J2SE clients:

- Long-running applications—Using J2SE is particularly good for Web service clients that run for extended periods of time. Because these applications run for long periods, developers must consider that both the client and the Web service need to maintain the state of at least some of the data. It is possible to embed conversational state within each method invocation, if required by the service's use case. However, Web service endpoints should be considered stateless. Furthermore, Web service clients generally cannot receive communication asynchronously from a server. It may be necessary for the client to poll the service to ensure that data is in sync, and such polling operations should be part of the client design.

- Using a graphic interface for complex data—J2SE clients can provide users with a graphical view of data. Such a graphical interface might permit a user to navigate and modify offline a large set of data returned by a service. The client can later update the Web service with any changes to the data. The JDBC 3.0 API provides `RowSet` functionality to exchange and update complex data. When used, the service provider must ensure that data submitted by the client is in sync with the server's copy.
- Requiring only intermittent network access—J2SE client can use Web services without needing to maintain continuous network access, relieving some of the burden on a network. Care must be taken to ensure data consistency between the service and the client.
- Requiring complex computations on the client—J2SE clients are well-suited for performing complex mathematical calculations, as well as operations that update data, and then submitting the results to a service. For example, these clients are adept at image manipulation, and they can relieve the server of this burden. Thus, J2SE clients can provide a better environment than a service for a user's interaction with data sets.

Figure 4.4 illustrates a J2SE application containing a rich GUI used to update catalog information. The user can manipulate attributes as well as graphical data using the application's GUI screens. When the user finishes, the application submits new or updated catalog data to the catalog service.

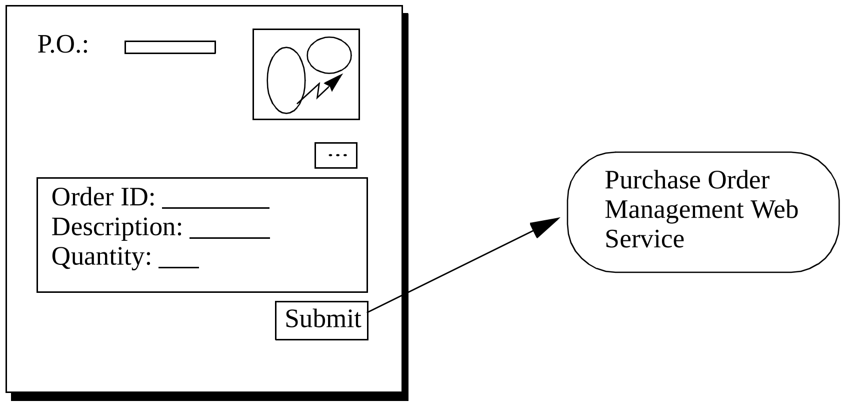

Figure 4.4 Rich J2SE Client Access Catalog Service

J2SE applications may be deployed using Java Web Start technology. Java Web Start simplifies deployment in large enterprise environments by ensuring that clients have available the proper versions of the Java Runtime Environment and all necessary libraries.

- ❒ Whenever possible use Java Web Start to provide a standardized means of deployment for J2SE clients.
- ❒ When designing a Web service client, try to keep Web service access code separate from GUI code.
- ❒ Since J2SE clients may operate in either a connected or disconnected mode, when developing these clients keep in mind issues related to maintenance of state and client and service synchronization.

4.5 J2ME Clients

Java 2 Platform, Micro Edition (J2ME) clients have the ability to interact remotely with a Web service. However, since J2ME clients have limited GUI capabilities compared to J2SE clients, consider using J2ME clients when mobility and remote access are requirements. Also consider using this type of client when immediacy of access is important.

For example, Figure 4.5 shows a J2ME client running on a cell phone and accessing an order tracking service. The client does not need an elaborate GUI or set of widgets to interact with the Web service.

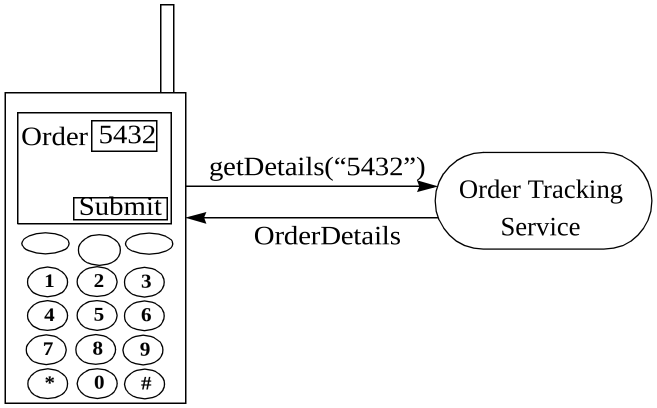

Figure 4.5 J2ME Client Accessing Web Service

J2ME clients may access Web services using a subset of the JAX-RPC API. When accessing Web services, J2ME clients should consider network connection and bandwidth issues, offline versus connected operation, and GUI and processing limitations.

Networks supporting J2ME devices may not always provide consistent connectivity. Applications using such networks must consider connection failure, or sporadic connectivity, and be designed so that recovery is possible.

Additionally, the bandwidths for many networks supporting J2ME devices limit the rate at which data can be exchanged. Applications need to be designed to limit their data exchange rate to that allowed by the network, and to consider the cost that these limitations imply. Care must also be taken to deal with network latency to keep the user experience acceptable. Since JAX-RPC does not specify a level of service for message delivery, the client application must take this into account and provide support for message delivery failures.

Keep in mind that J2ME network providers may charge for network usage by the kilobyte. J2ME-targeted applications may be expensive for the user unless care is taken to limit the data transferred.

Applications for J2ME devices may work in an offline, or disconnected, mode as well as an online, or connected, mode. When working in an offline mode, applications should collect data and batch it into requests to the Web service, as well as obtain data from the service in batches. Consideration should be given to the amount of data that is passed between the Web service and the client.

Applications for J2ME devices have a standard, uniform set of GUI widgets available for manipulating data, such as the liquid crystal display UI that is part of MIDP 1.0 and MIDP 2.0. Plus, each device type may have its own set of widgets. Such widgets have different abilities for validating data prior to accessing a service.

J2ME devices, while capable of small computations and data validation, have limited processing capabilities and memory constraints. Client applications need to consider the types of data exchanged and any pre- and post-processing required. For example, J2ME devices have limited support for XML document processing and are not required to perform XML document validation.

In summary, keep the following considerations in mind when designing J2ME clients.

- ❐ **Connectivity and bandwidth limitations**—J2ME clients may have limited bandwidth, intermittent disconnects, and fees for connection usage.
- ❐ **Processing power**—The processing capabilities of a Web service client should be considered in its design.
- ❐ **State maintenance**—J2ME clients may operate in a connected or disconnected mode, requiring the maintenance of state and synchronization between client and service.

4.6 Developing Client Applications to Use a Web Service

All client applications follow certain steps to use a Web service. In brief, they must first look up or locate the service, make a call to the service, and process any returned data. The choice of communication approach determines much of the details for performing these steps.

An application can communicate with a service using stubs, dynamic proxies, or the dynamic invocation interface (DII) `Call` interface. (The next section explains these approaches.) With these approaches, a developer relies on some form of client-side proxy class (stub, dynamic proxy, or a DII interface) representing a Web service to access the service's functionality. The client developer decides which representation to use based on the WSDL availability, the service's endpoint address, and the use cases for the service.

After deciding on a communication approach, developers need to do the following when designing and developing client applications:

1. **Assess the nature and availability of the service and the service description.**

   Typically, the first step is to determine the location of the Web service's WSDL document and choose a communication model. If you are using the stub or dynamic proxy approach, you must first locate and gain access to the full WSDL document representing the Web service, since you develop the client application from the stubs and supporting files generated from the WSDL. If you have access to only a partial WSDL file, then use the DII approach, since this approach lets you locate the WSDL file at runtime from a registry. See "Communication Modes for Accessing a Service" on page 135 for more information.

2. **Create a client-side Web service delegate.**

   - If applicable, generate the necessary stubs and implementation classes—Generated classes are not portable across implementations. If portability is important, limit your use of vendor-specific method calls in the stub classes.
   - Locate the service—Depending on the communication mode, there are different ways to locate a service. See "Locating and Accessing a Service" on page 143.
   - Configure the stub or Call objects—See "Stub and Call Configuration" on page 148.

3. **Invoke the service.**

   - When using stubs and dynamic proxies, invocations are synchronous. Use DII if you choose to have a one-way call.
   - Handle parameters, return values, and exceptions

4. **Present the view**.

   There are different ways clients can handle the presentation to the user of the Web service response. In some cases, a service sends its response as an XML document, and this document must be transformed or mapped to a suitable format for presentation. Web tier clients might pass the service response to a Web component such as JSP and let the component apply a transformation to the XML document or otherwise handle the returned data. EJB tier clients may themselves apply XSLT transformations to the returned content to create the target content, which they then pass to a Web component.

The following sections examine some of the considerations for designing and implementing client applications.

4.6.1 Communication Modes for Accessing a Service

To best determine the communication mode for your client application, you need to know the answers to the following two questions. Is the Web service's full WSDL document known? Is the endpoint address for the Web service known?

As indicated, there are three principal modes for a client application's communication with a Web service: stub, dynamic proxy, and dynamic invocation interface (DII). When a WSDL document is not provided, a developer, when developing a client application, may use a Web service locator tool to look up the service endpoint address and conforming WSDL document in a registry. Such locator tools are typically included with the development environment. A client that uses stubs or dynamic proxies may use JAXR technology to look up at runtime the endpoint URL address that conforms to a specific WSDL document. The developer can set or change at runtime the client application's reference to the endpoint address.

It is also possible for a client developer to locate the service's WSDL file from a registry. In this case, the developer locates the WSDL from a registry and uses the DII approach at runtime to access the service.

The three modes all require the JAX-RPC runtime between the Web service and the clients at runtime. The JAX-RPC runtime may be provided by the J2EE platform or it may be deployed separately. Clients using either the stub or dynamic proxy models to access a service require the prior definition of a service endpoint interface. Note that when we refer to the service endpoint interface, which is the interface between the stub and dynamic proxy clients and the JAX-RPC API and stub class, we are referring to the interface that represents the client's view of a Web service.

For the stub model, a JAX-RPC runtime tool generates during development static stub classes that enable the service and the client to communicate. The stub class, which sits between the client and the client representation of the service endpoint interface, is responsible for converting a request from a client to a SOAP message and sending it to the service interface. The stub class also converts responses from the service endpoint, which it receives as SOAP messages, to a format understandable by the client. In a sense, a stub is a local object that acts as a proxy for the service endpoint.

The dynamic proxy model provides the same functionality as the stub class, but does so in a more dynamic fashion. Stubs and dynamic proxies both provide the developer access to the `javax.xml.rpc.Stub` interface, which represents a service endpoint. With both models, it is easy for a developer to program against the service endpoint interface, particularly because the JAX-RPC runtime does much of the communication work behind the scenes. The dynamic proxy model differs from the stub model principally because the dynamic proxy model does not require code generation.

DII is a call interface that supports a programmatic creation and invocation of the JAX-RPC request. Using DII, a client can call a service or a remote procedure on a service without knowing at compile time the exact service name or the procedure's signature. A DII client can discover this information at runtime, making use of a service broker that can dynamically look up the service and its remote procedures.

If using the DII model, clients can dynamically access a service at runtime. The stub approach relies on a tool that uses the WSDL file to create the service endpoint interface, plus generate stub and other necessary classes. These generated classes eliminate the need for the developer to use the JAX-RPC APIs (although the JAX-RPC runtime is used behind the scenes). By contrast, the dynamic proxy and DII approaches both require the developer to use the JAX-RPC APIs.

4.6.1.1 Using Stub Communication

J2EE clients should use generated stubs to access services, especially for services that are fairly static. Stub communication is easy and allows for fast programmatic Java access to a Web service. This is the recommended approach for accessing services and their WSDL files when they are unlikely to change over time.

Figure 4.6 shows how a client application might access a Web service using a stub generated by a tool prior to the client's deployment and compilation. The WSDL file for the service served as input to the tool. The client-side service endpoint interface, which implements the service endpoint interface on the client-side of the communication channel, provides the client view of the Web service.

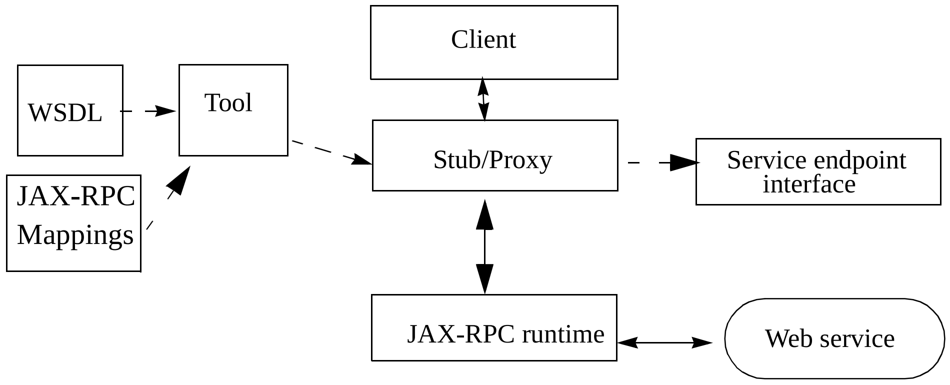

Figure 4.6 Stub Communication Model

The stub class, which is generated by the tool, implements the client-side service endpoint interface. The stub class acts as a proxy to the Web service for the client; that is, the stub is the client's view of the Web service. The tool that generates this stub class also generates all necessary helper classes for accessing the service. These helper classes define the parameters for calls to the service and the service's return values, and they ensure that the parameters and return values are all of the proper types expected by the JAX-RPC runtime. To generate these stub and helper classes, the tool relies on the WSDL document as well as a JAX-RPC mapping file. The mapping file supplies information regarding the Java-to-XML bindings, such as the correct package name for generated classes.

Applications in J2ME environments can use only stubs to access Web services, and stubs are portable for J2ME devices. Dynamic proxies and DII are not supported by the JAX-RPC profile for J2ME environments.

Keep in mind that, although stubs are not portable across J2EE implementations at this time, it is expected that the next version of JAX-RPC will address this portability issue.

❒ The stubs approach, especially in a Java environment, is often the easiest to use, because the developer can work with generated class files representing the service method call parameters and return values. However, there are greater dependencies between the client to the Web service, leading to problems if the service changes frequently. It also requires that stub classes be generated before the compiling the application.

4.6.1.2 Using Dynamic Proxy Communication

A dynamic proxy is another approach clients can use to communicate to Web services. The dynamic proxy approach is similar in many ways to the stub approach previously described. However, unlike stubs, the client developer using a dynamic proxy needs only the client-side interface that matches the service endpoint interface. That is, clients using dynamic proxies program to an interface that ensures the client application development is portable across other JAX-RPC runtime implementations. Developers using dynamic proxies must create Java classes to serve as value types—these classes have an empty constructor and set methods for each field, similar to JavaBeans classes.

The dynamic proxy is based on the service endpoint interface (the client view of the Web service), its WSDL document, and a JAX-RPC mapping file (similar to the stubs model). (See Figure 4.7.) Instead of requiring at compilation a stub class implementing the service endpoint interface, the equivalent dynamic proxy is generated at runtime. Client applications can access this proxy using the `javax.xml.rpc.Service` method `getPort`.

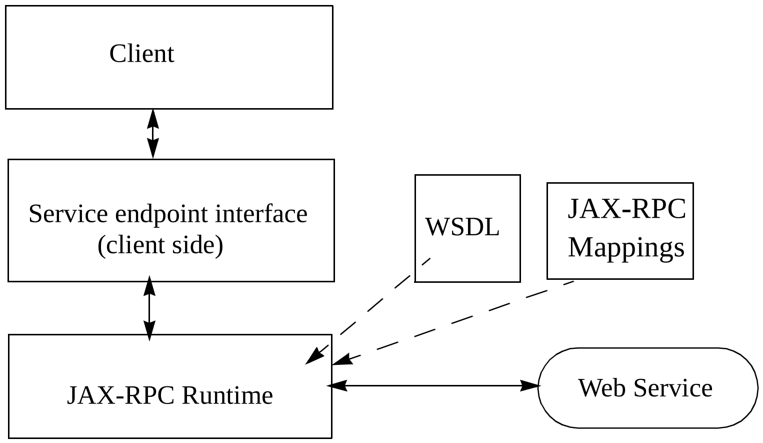

Figure 4.7 Accessing a Service Using a Dynamic Proxy

Note in Figure 4.7 that a client accesses a Web service via the client-side representation of the service endpoint interface, while the JAX-RPC runtime handles the work of communicating with the respective service. A developer programs against an interface, which provides a flexible way to access a service in a JAX-RPC runtime-neutral fashion.

❒ Consider using the dynamic proxy approach if portability is important to your application, since this approach uses the service's endpoint interface to communicate with a service at runtime. Although not as efficient as using the stubs approach, dynamic proxy communication is probably the most portable mode across JAX-RPC implementations.

❒ Keep in mind that classes that access dynamic proxy-generated stubs are implementation specific.

❒ Consider using this approach when client applications access services that change only occasionally.

❒ Because of how they access a service at runtime, dynamic proxies may have additional cost overhead when calls are actually made.

4.6.1.3 Using DII Call Interface

A client application may also dynamically access a Web service by locating the service at runtime from a registry. The client does not know about the service when it is compiled; instead, the client discovers the service's name from a JAXR registry at runtime. Along with the name, the client discovers the required parameters and return values for making a call to the service. Using a dynamic invocation interface, the client locates the service and calls it at runtime.

Generally, the DII approach is more difficult for a developer to use. A developer must work with a more complex interface than with the stub or dynamic proxy approach. Not only does this interface require more work on the part of the developer, it is more prone to class cast exceptions. In addition, access may be slower with the DII approach. A developer may choose to use the DII approach when a complete WSDL document is not available or provided, particularly when the WSDL document does not specify ports. The DII approach is more suitable when used within a framework, since from within a framework, client applications can generically and dynamically access services with no changes to core application code.

Figure 4.8 shows how a client uses the JAXR API to look up the endpoint WSDL for a service in a registry. The WSDL for the service has previously published information to the registry about itself. The client uses the information from the registry to construct a `javax.xml.rpc.Call`, which it uses to access the Web service via the JAX-RPC runtime.

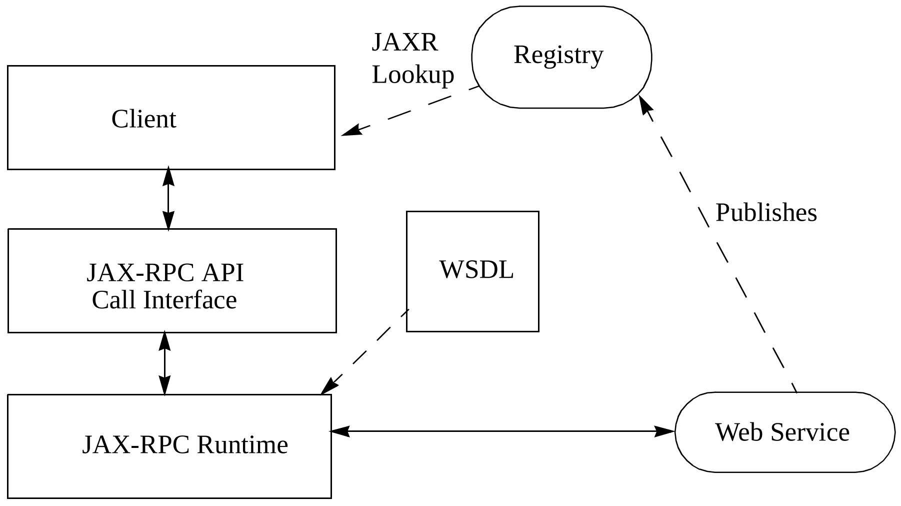

Figure 4.8 DII Call Interface

Using the DII `Call` interface allows a client application to define at runtime. the service name and the operations it intends to call on the service, thus giving the client the benefit of loosely coupling its code with that of the service. The client is less affected by changes to the service, whether those changes involve access to the service or data requirements. In addition, the DII communication model permits Web service access code to be standardized among a set of clients and reused as a component.

❐ The DII approach, although it provides the highest level of abstraction, involves more work than the other approaches and should be used only when use cases require it. You may consider using this approach if a service changes frequently, though generally it is better to use a dynamic proxy if a service changes only occasionally.

4.6.1.4 Summary of Communication Model Guidelines

Table 4.1 summarizes the communication models for Web service clients.

Table 4.1 Client Communication Modes

	Stub	Dynamic Proxy	DII
Portable stubs across JAX-RPC implementations.	No. Future JAX-RPC specifications will address the issue of portable stubs.	Yes	Yes
Portable Client Code across JAX-RPC implementations.	Yes in the J2EE platform when an application uses a neutral means for accessing the stubs. (For an example, see Code Example 4.2.) Since stubs are bound to a specific JAX-RPC runtime, reliance on JAX-RPC-specific access to a stub may not behave the same on all platforms. Stub code needs to be generated for an application.	Yes	Yes
Requires generation of code using a tool	Yes	No. A tool may be used to generate value objects required by a service endpoint interface (but not serializers and other artifacts).	No

Table 4.1 Client Communication Modes (Continued)

	Stub	Dynamic Proxy	DII
Ability to change the service endpoint URL	Yes, but the WSDL must match that used to generate the stub class and supporting classes.	Yes, but the service endpoint interface and supporting value objects must match.	Yes
Supports service-specific checked exceptions	Yes	Yes	No. All are `java.rmi.Remote` exceptions. Checked exceptions cannot be used when calls are made dynamically.
Supports one way communication mode	No	No	Yes
Supports the ability to dynamically specify value objects at runtime	No. Developer must program against a service endpoint interface.	No. Developer must program against a service endpoint interface.	Yes. However, returns Java objects which the developer needs to cast to application-specific objects as necessary (`ClassCastException` may occur).
Supported in J2ME	Yes	No	No
Supported in J2SE and J2EE	Yes	Yes	Yes
Requires full WSDL	No. A service endpoint interface may generate a stub class along with information concerning the protocol binding.	No. A partial WSDL (one with the service port element undefined) may be used.	No. Calls may be used when partial WSDL or no WSDL is specified.Use of methods other than the `createCall` method on the `Call` interface may result in unexpected behavior in such cases.

Choose the communication mode for your client application after considering the following:

- ❐ J2EE clients should use the stubs approach. The stubs approach is the easiest one for developers to use, especially since it insulates developers from most system considerations. Keep in mind, however, that with the stubs approach you might lose some portability across JAX-RPC implementations. A JAX-RPC tool generates stub classes using a WSDL document as well as a JAX-RPC mapping file, which gives the stubs more detailed information regarding the Java-to-XML bindings.
- ❐ J2EE clients should use the dynamic proxy approach when portability is a requirement. Since this approach does not require a JAX-RPC mapping file, some ambiguity may occur and this may result in unexpected behavior when calling a service.
- ❐ The DII approach is probably the most difficult to use of the three approaches. Only consider using the DII approach if you need the runtime ability to look up operations.
- ❐ All non-J2EE clients should use the stubs approach if at all possible.

4.6.2 Locating and Accessing a Service

Client applications running in J2EE environments use the JNDI `InitialContext.lookup` method to locate a service, whereas a J2SE client can use the `javax.xml.rpc.ServiceFactory` class or an implementation-specific stub to locate a service. Clients then use a stub, proxy, or DII `Call` object to access the service. Recall that a Web service is a collection of communication end points, or ports. Each port references a network address for the service and a reusable binding. (See “Web Services Definition Language” on page 41.)

Code Example 4.1 shows how an application in a J2EE environment might use a stub to access a service. The application locates the service using a JNDI `InitialContext.lookup` call.

```
Context ic = new InitialContext();
OpcOrderTrackingService opcOrderTrackingSvc =
       (OpcOrderTrackingService) ic.lookup(
       "java:comp/env/service/OpcOrderTrackingService");
OrderTrackingIntf port =
          opcOrderTrackingSvc.getOrderTrackingIntfPort();
OrderDetails od = port.getOrderDetails(orderId);
```

Code Example 4.1 Accessing a Service with a Stub in a J2EE Environment

Because it depends on the generated stub classes, the client code in Code Example 4.1 is not the recommended approach. Although this example works without problems, JAX-RPC gives you a more neutral way to access a service and obtain the same results. By using the JAX-RPC `javax.xml.rpc.Service` interface method `getPort`, you can access a Web service in the same manner regardless of whether you use stubs or dynamic proxies. The `getPort` method returns either an instance of a generated stub implementation class or a dynamic proxy, and the client can then uses this returned instance to invoke operations on the service endpoint.

The `getPort` method removes the dependencies on generated service-specific implementation classes. When this method is invoked, the JAX-RPC runtime selects a port and protocol binding for communicating with the port, then configures the returned stub that represents the service endpoint interface. Furthermore, since the J2EE platform allows the deployment descriptor to specify multiple ports for a service, the container, based on its configuration, can choose the best available protocol binding and port for the service call. (See Code Example 4.2.)

```
Context ic = new InitialContext();
Service service = (Service)ic.lookup(
          “java:comp/env/service/OpcPurchaseOrderService”);
PurchaseOrderIntf port = (PurchaseOrderIntf)service.getPort(
          PurchaseOrderIntf.class);
```

Code Example 4.2 Looking up a Port Using a Stub or Dynamic Proxy

Code Example 4.2 illustrates how a J2EE client might use the `Service` interface `getPort` method. Rather than cast the JNDI reference to the service implementation class, the code casts the JNDI reference to a `javax.xml.rpc.Service` interface. Using the `Service` interface in this manner reduces the dependency on generated classes. The client developer, by invoking the `getPort` method, uses the client-side representation of the service endpoint interface to look up the port.

After obtaining the port, the client may make any calls desired by the application on the port.

❐ To reduce the dependency on generated classes, use the `java.xml.rpc.Service` interface and the `getPort` method as a proxy for the service implementation class.

An application in a non-J2EE environment uses a stub to make a Web services call in different manner. The client application accesses a stub for a service using the method `getOrderTrackingIntfPort` on the generated implementation class, `OpcOrderTrackingService_Impl`, which is specific to each JAX-RPC runtime. J2SE or J2ME clients use these generated `_Impl` files because they do not have access to the JNDI services available to clients in a J2EE environment. See Code Example 4.3.

```
Stub stub = (Stub)(new OpcOrderTrackingService_Impl().
   getOrderTrackingIntfPort());
OrderTrackingIntf port = (OrderTrackingIntf)stub;
```

Code Example 4.3 Accessing a Service with a Stub in J2SE and J2ME Environments

A J2SE or J2ME client can also access a service using the `javax.xml.rpc.ServiceFactory` class to instantiate a stub object. Code Example 4.4 shows how a J2SE client might use a factory to locate the same order tracking service.

```
ServiceFactory factory =  ServiceFactory.newInstance();
Service service = factory.createService(new QName(
   "urn:OpcOrderTrackingService", "OpcOrderTrackingService"));
```

Code Example 4.4 Looking up a Service in J2SE and J2ME Environments

Similarly, Code Example 4.5 shows how a J2SE application might use a dynamic proxy instead of a stub to access a service.

```
...
ServiceFactory sf = ServiceFactory.newInstance();
```

```
String wsdlURI =
       "http://localhost:8001/webservice/OtEndpointEJB?WSDL";
URL wsdlURL = new URL(wsdlURI);
Service ots = sf.createService(wsdlURL,
       new QName("urn:OpcOrderTrackingService",
       "OpcOrderTrackingService"));
OrderTrackingIntf port = (
       OrderTrackingIntf)ots.getPort(new QName(
       "urn:OpcOrderTrackingService", "OrderTrackingIntfPort"),
       OrderTrackingIntf.class);
...
```

Code Example 4.5 J2SE Client Dynamic Proxy Service Lookup

Code Example 4.5 illustrates how a J2SE client might program to interfaces that are *portable* across JAX-RPC runtimes. It shows how a J2SE client uses a `ServiceFactory` to look up and obtain access to the service, represented as a `Service` object. Because the `Service` object is generic, the client uses the qualified name, or `QName`, of the service to obtain the service's port. (The WSDL document defines the `QName` for the service.) The client needs to pass as arguments the `QName` for the target service port and the client-side representation of the service endpoint interface.

When using a stub or dynamic proxy, a client application in a J2EE environment obtains the service using the JNDI `InitialContext.lookup` method. Then, the application casts the returned port to a `javax.xml.rpc.Stub` so that it can use the `_setProperty` method to set the service's endpoint address, `Stub.ENDPOINT_ADDRESS_PROPERTY`, to a service different than the one specified in the WSDL document. Code Example 4.5 shows how a J2SE application might use a dynamic proxy instead of a stub to access a service.

By contrast, Code Example 4.6 shows how a J2EE client might use a dynamic proxy to look up and access a service. These two examples show how much simpler it is for J2EE clients to look up and access a service than it is for J2SE clients, since a JNDI lookup from the `IntialContext` of an existing service is much simpler than configuring the parameters for `ServiceFactory`. The J2EE client just invokes a `getPort` call on the client-side representation of the service endpoint interface.

```
Context ic = new InitialContext();
Service ots =
       (Service) ic.lookup(
       "java:comp/env/service/OpcOrderTrackingService");
OrderTrackingIntf port = (OrderTrackingIntf)ots.getPort(
    OrderTrackingIntf.class);
```

Code Example 4.6 Using a Dynamic Proxy in a J2EE Environment

A J2SE client using the DII approach might implement the code shown in Code Example 4.7 to look up the same service at runtime. DII communication supports two invocation modes: synchronous and one way, also called fire and forget. Both invocation modes are configured with the `javax.xml.rpc.Call` object. Note that DII is the only communication model that supports one way invocation. Code Example 4.7 illustrates using the DII approach for locating and accessing a service. It shows how the `Call` interface used by DII is configured with the property values required to access the order tracking Web service. The values set for these properties may have been obtained from a registry. Keep in mind that using DII is complex and often requires more work on the part of the client developer.

```
Service service = //get service
QName port = new QName("urn:OpcOrderTrackingService",
       "OrderTrackingIntfPort");
Call call = service.createCall(port);
call.setTargetEndpointAddress(
       "http://localhost:8000/webservice/OtEndpointEJB");
call.setProperty(Call.SOAPACTION_USE_PROPERTY,
       new Boolean(true));
call.setProperty(Call.SOAPACTION_URI_PROPERTY,"");
call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);
QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
call.setReturnType(QNAME_TYPE_STRING);
call.setOperationName(
new QName(BODY_NAMESPACE_VALUE "getOrderDetails"));
call.addParameter("String_1", QNAME_TYPE_STRING,
       ParameterMode.IN);
```

```
    String[] params = {orderId};
    OrderDetails = (OrderDetails)call.invoke(params);
```

Code Example 4.7 J2SE Client Using DII to Access a Web Service

4.6.3 Stub and Call Configuration

Developers may want to configure an instance of a stub or a `Call` interface prior to invoking a service or may prefer to allow the configuration to take place dynamically at runtime. Often, the developer configures the stub or `Call` interface prior to invoking the service when the service requires basic authentication. For example, a J2SE client application might set a user name and password in the stub or `Call` just before invoking the service; the service requires these two fields so that it can authenticate the client. In other cases, the developer may want flexibility in specifying the endpoint address to use for a particular service, depending on network availability and so forth. The developer might configure this endpoint address dynamically at runtime.

Stubs may be configured statically or dynamically. A stub's static configuration is set from the WSDL file description at the time the stub is generated. Instead of using this static configuration, a client may use methods defined by the `javax.xml.rpc.Stub` interface to dynamically configure stub properties at runtime. Two methods are of particular interest: `_setProperty` to configure stub properties and `_getProperty` to obtain stub property information. Clients can use these methods to obtain or configure such properties as the service's endpoint address, user name, and password.

It is advisable to cast vendor-specific stub implementations into a `javax.xml.rpc.Stub` object for configuration. This ensures that configuration is done in a portable manner and that the application may be run on other JAX-RPC implementations with minimal changes, if any.

Code Example 4.8 shows how a J2EE client might use `Stub` interface methods to look up and set the endpoint address of a Web service.

```
Service opcPurchaseOrderSvc =
        (Service) ic.lookup(AdventureKeys.PO_SERVICE);
PurchaseOrderIntf port =
        opcPurchaseOrderSvc.getPort(PurchaseOrderIntf.class);
```

```
((Stub)port)._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
   "http://localhost:8000/webservice/PoEndpointEJB");
```

Code Example 4.8 Setting Properties on a Stub

In a J2EE environment, the J2EE container reserves the right to use two security-related properties for its own purposes. As a result, J2EE clients using any of the three communication approaches (stub, proxy, or DII) should not configure these two security properties:

```
javax.xml.rpc.security.auth.username
javax.xml.rpc.security.auth.password
```

However, J2SE developers do need to set these two security properties if they invoke a Web service that requires basic authentication. When using the DII approach, the client application may set the properties on the `Call` object. See Code Example 4.7, which illustrates setting these properties.

❒ J2EE client developers should avoid setting properties other than the `javax.xml.rpc.endpoint.address` property.

❒ Avoid setting nonstandard properties if it is important to achieve portability among JAX-RPC runtimes. Nonstandard properties are those whose property names are *not* preceded by `javax.xml.rpc`.

❒ Do not assume that session support exists, since the JAX-RPC specification does not require session support for compatibility with existing SOAP runtimes. When using a Web service endpoint developed by others, developers should avoid having their client applications set the `javax.xml.rpc.session.maintain` property on these interfaces. (That is, if you did not write the stub or `Call` interface yourself, do not have the client set the `javax.xml.rpc.session.maintain` property on them.) This property pertains to a service's ability to support sessions. Unless you have control over the development of both the client and the endpoint, such as when both are developed within the same organization, you cannot be sure that the Web service endpoints support sessions, and you may get into trouble if you set this property incorrectly.

4.6.4 WSDL to Java Type Mapping

When working with Web services, there may be differences between the SOAP-defined types (defined in the WSDL document) used by the service and the Java-defined types used by the client application. To handle these different types, a client of a Web service cannot use the normal approach and import remote classes. Instead, the client must map the WSDL types to Java types to obtain the parameter and return types used by the service. Once the types are mapped, the client has the correct Java types to use in its code.

Generally, the JAX-RPC runtime handles the mapping of parameters, exceptions, and return values to JAX-RPC types. When a client invokes a service, the JAX-RPC runtime maps parameter values to their corresponding SOAP representations and sends an HTTP request containing a SOAP message to the service. When the service responds to the request, the JAX-RPC runtime receives this SOAP response and maps the return values to Java objects or standard types. If an exception occurs, then the runtime maps the `WSDL:fault` to a Java exception, or to a `javax.rmi.RemoteException` if a `soap:fault` is encountered. (This is discussed further in “Handling Exceptions” on page 155).

The JAX-RPC runtime supports the following standard value types: `String`, `BigInteger`, `Calender`, `Date`, `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and arrays of these types. Services can return mime types as images mapped to the `java.awt.Image` class and XML text as `javax.xml.transform.Source` objects. (Note that because the WSDL Basic Profile 1.0 does not support `javax.xml.transform.Source` objects as mime types, this usage should be avoided until supported by a future version.) A service may also return complex types, and these are mapped to Java `Object` representations. For more information, see “Interfaces and Parameters” on page 71.

When stubs are used, the JAX-RPC WSDL-to-Java mapping tool maps parameter, exception, and return value types into the generated classes using information contained in the developer-provided WSDL document. Complex types defined within a WSDL document are represented by individual Java classes, as are exceptions. A WSDL-to-Java mapping tool included with the JAX-RPC runtime also generates classes to serialize and deserialize these values to XML.

The JAX-RPC runtime uses these generated classes to serialize parameter values into a SOAP message and deserialize return values and exceptions.

❒ Use WSDL-to-Java tools to generate support classes, even if using the dynamic proxy or DII approach.

Whenever possible, developers should try to use a tool to do the WSDL to Java mapping, since a tool correctly handles the WSDL format and mapping semantics. Note that the Java objects generated by these mapping tools contain empty constructors and get and set methods for elements. You should use the empty constructor to create an instance of the object and set any field or element values using the corresponding set methods. JAX-RPC does not guarantee that it will correctly to map values created as part of the constructor to the corresponding fields.

Although not advisable, it is possible for a developer to work without the benefit of a mapping tool, if none are available. However, without such mapping tools the scope of the developer's work greatly expands. For example, just to compile the client code, the developer must understand the WSDL for a service and generate by hand Java classes that match the parameter and return types defined in the WSDL document or, in the case of a dynamic proxy, the client-side representation of the service endpoint interface. These classes must be set up properly so that the JAX-RPC runtime can match SOAP message types to the corresponding Java objects.

4.6.5 Processing Return Values

J2EE clients usually use Web components to display returned data. For example, a client accessing an order tracking service might display tracking information in a Web page using an HTML browser. The J2EE component may take the values returned from the Web service and handle them just like any other Java object. For example, a Web component in a J2EE client application might query the status of an order from an order tracking service, which returns these values within a JavaBeans-like object. The client component places the returned object in the request scope and uses a JSP to display its contents. (See Code Example 4.9.)

```
<html>
Order ID: ${bean.orderId}
Status: ${bean.status} <br>
```

```
Name: ${bean.givenName} ${bean.familyName} <br>
</html>
```

Code Example 4.9 JSP for Generating an HTML Document

The J2EE platform has a rich set of component technologies for generating Web content, including JavaServer Pages (JSP) technology for generating HTML content and Java Standard Tag Libraries (JSTL). JSTL is a set of tags that assist a client developer in formatting JSPs. For example, JSTL provides additional tags for looping, database access, object access, and XSLT stylesheet transformations. The current version of JSP (2.0), along with future versions, provides an expression language (EL) that allows a developer to access bean properties. Together, developers can use JSTL and JSP technologies to generate HTML documents from data retrieved from a service. For example, a developer might generate the following HTML document for the order details (see Code Example 4.10). This HTML document is returned to the HTML browser client that requested the service.

```
<html>
   Order: 54321<br>
   Status: SHIPPED<br>
   Name: Duke Smith<br>
</html>
```

Code Example 4.10 HTML Document

A different use case might be EJB tier clients (EJB components) using returned data in a workflow. For example, as the workflow progresses, they may provide order tracking updates by formatting the tracking data in HTML and attaching it to an email message sent to the customer. Unless the initial request originated from a Web tier client, EJB components in a workflow situation do not have Web-tier technologies such as JSP available to them. Furthermore, the order tracking service may return results as XML documents, requiring EJB components to apply XSL transformations to these documents. Code Example 4.11 shows an XML document, returned as a `String` or `javax.xml.transform.Source` object, containing the returned data from the service.

```
<orderdetails>
   <id>54321</id>
   <status>SHIPPED</status>
   <shippinginfo>
      <family-name>Smith</family-name>
      <given-name>Duke</given-name>
   </shippinginfo>
</orderdetails>
```

Code Example 4.11 Data Returned as XML Document

An EJB component may use the XSL stylesheet shown in Code Example 4.12 to transform an order details XML document to the same HTML document as in Code Example 4.10. This HTML document may be attached to an email message and sent to a customer. XSL transformations may also be used to transform a document into multiple formats for different types of clients.

```
<xsl:stylesheet version='1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>

 <xsl:output method="html"/>
 <xsl:template match="text()"/>

 <xsl:template match="orderdetails">
   <html>
   Order: <xsl:value-of select="id/text()"/><br/>
   <xsl:apply-templates/>
   </html>
 </xsl:template>

 <xsl:template match="shippinginfo">
  Name: <xsl:value-of select="given-name/text()"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="family-name/text()"/>
  <br/>
 </xsl:template>

 <xsl:template match="status">
```

```
  Status: <xsl:value-of select="text()"/><br/>
 </xsl:template>
</xsl:stylesheet>
```

Code Example 4.12 XSL Stylesheet

Technologies for handling and transforming XML documents are also available to Web components such as servlets or JSPs. Web components may use custom tag components, XSL, or the JAXP APIs to handle XML documents. (For more information on handling XML documents, see Chapter 5.)

- ❐ Generally, whenever possible client developers should use JSP to generate responses by a service and to present the data as a view to clients (such as browsers that use Web tier technologies).
- ❐ For clients in a non-Web tier environment where JSP technology is not available, developers should use XSLT stylesheets.

Developers can use JSP technology to access content in XML documents and to build an HTML page from the XML document contents. Code Example 4.13 shows how JSP technology makes it easy to parse XML content and build an HTML page from the order detail contents shown in Code Example 4.11.

```
<%@ taglib prefix="x" uri="/WEB-INF/x-rt.tld" %>
<x:parse xml="${orderDetailsXml}" var="od" scope="application"/>
<html>
  Order:<x:out select="$od/orderdetails/id"/><br>
  Status:<x:out select="$od/orderdetails/status"/><br>
  Name:<x:out select="$od/orderdetails/shippinginfo/given-name"/>
       <x:out select="$od/orderdetails/shippinginfo/family-name"/>
</html>
```

Code Example 4.13 JSP Generating a Web Service Response

In this example, the J2EE application first places the order details document received from the service in the request scope using the key `orderDetailsXML`. The next lines are JSP code that use the `x:out` JSTL tag to access the order details

XML content. These lines of code select fields of interest (such as order identifier, status, and name fields) using XPath expressions, and convert the data to HTML for presentation to a browser client. The JSP code relies on JSTL tags to access these portions of the order details document. For example, the JSTL tag `x:out`, which uses XPath expressions, accesses the order identifier, status, and name fields in the order details document. When the JSP processing completes, the result is identical to the HTML page shown in Code Example 4.10.

4.6.6 Handling Exceptions

Two types of exceptions occur for client applications that access Web services: system exceptions and service exceptions, which are thrown by a service endpoint. System exceptions are thrown for such conditions as passing incorrect parameters when invoking a service method, service inaccessibility, network error, or some other error beyond the control of the application. Service exceptions, which are mapped from faults, are thrown when a service-specific error is encountered. The client application must deal with both types of exceptions.

4.6.6.1 System Exceptions

System exceptions generally involve unanticipated errors—such as a network timeout or an unavailable or unreachable server—that occur when invoking a Web service. Although system exceptions are usually out of the control of the client developer, you should still be aware that these errors may occur and you should have a strategy for your client application to deal with these situations. For example, with these types of errors, it may be useful to have the client application prompt the user to retry the operation or redirect the user to an alternative service, if possible. Many times the only solution is to display the exception occurrence to the end user. Sometimes it may be appropriate to translate the system exception to an unchecked exception and devise a global way to handle them. The exact solution is particular to each application and situation.

System exceptions arise from a variety of reasons. If an EJB component client is doing its work on behalf of a Web tier client, the EJB client should throw an exception to the Web tier client. The Web tier client notifies the user, giving the user a chance to retry the action or choose an alternative action.

When communicating using stubs, most system exceptions involve network issues or the configuration of the stub. A configuration error is thrown from the `_getProperty` method as a `javax.xml.rpc.JAXRPCException`, and indicates a property name that is not supported or is invalid, a property setting value that is not allowed, or a type mismatch.

When using a dynamic proxy, the `getPort` method may throw a `javax.xml.rpc.ServiceException` if the WSDL document has insufficient metadata to properly create the proxy. If the resulting stub is not properly configures, then that throws a `javax.xml.rpc.JAXRPCException`.

When using the DII `Call` interface, a client may receive a `RemoteException` if a network failure occurs. A client may also be thrown a `javax.xml.rpc.JAXRPCException` if the `Call` set methods use invalid property names or try to set properties with invalid values.

For example, a client application accessing an order tracking service may want to use an alternative service or endpoint if it receives a `RemoteException` indicating the unavailability of the service. Often, with Web services a `RemoteException` is the result of a SOAP fault. Since a `RemoteException` contains an explanation of the error, the application can use that message to provide its own error message to the user and can prompt the user for an appropriate action to take.

J2EE clients using a servlet container may leverage the servlet error messaging system and map the `RemoteException` to some exception handling instructions in the `web.xml` file. The client can throw its own `javax.servlet.ServletException` and can display a general application error message. It is possible to extend `ServletException` to create a set of application-specific exceptions for different errors. It may also use the file to define a specific error JSP to handle the error, as shown in Code Example 4.14.

```
<error-page>
    <exception-type>java.lang.Runtime</exception-type>
    <location>/order_tracking_system_exception.jsp</location>
</error-page>
```

Code Example 4.14 Using the Servlet Error Messaging System

We present here some strategies for dealing with system exceptions.

Although they do not directly interact with the user, workflow clients implemented as EJB components can benefit from transactional processing, particularly if using container-managed transactions. However, since you cannot assume that the Web service is also transactional, you might have to manually back out some of your changes if other Web services are involved in the processing. Keep in mind that backing out changes can be difficult to accomplish and is prone to problems.

An EJB or other J2EE client that receives an error when accessing a service may retry connecting to the service a set number of times. If none of the retries are successful, then the client can log an error and quit the workflow. Or, the client may forward the unit of work to another process in the workflow and let that component access the same service at a later point or use a different service. With the stub and dynamic proxies communication modes, the alternative service must either have a endpoint interface or be defined by a known WSDL document. EJB component-based clients using DII may locate an alternative service using the JAXR API from a registry.

4.6.6.2 Service Exceptions

Service exceptions occur when a Web service call results in the service returning a fault. A service throws such faults when the data presented to it does not meet the service criteria. For example, the data may be beyond boundary limits, it may duplicate other data, or it may be incomplete. These exceptions are defined in the service's WSDL file as `operation` elements, and they are referred to as `wsdl:fault` elements. These are checked exceptions in client applications. For example, a client accessing the order tracking service may pass it an order identifier that does not match orders kept by the service. The client may receive an `OrderNotFoundException`, since that is error message defined in the WSDL document:

```
<fault name="OrderNotFoundException"
          message="tns:OrderNotFoundException"/>
```

This exception mapping mechanism may not be used with DII, since this communication mode returns all exceptions as `java.rmi.RemoteException`. Note that there is no guarantee that the JAX-RPC runtime system will throw a `ServiceException` for a specific `wsdl:fault` error. The runtime may instead throw a `javax.xml.rpc.soap.SOAPFaultException` runtime exception.

Use the JAX-RPC tools to map faults to Java objects. (See "WSDL to Java Type Mapping" on page 150.) These tools generate the necessary parameter mappings for the exception classes and generate the necessary classes for the mapping. Generated exceptions classes extend `java.lang.Exception`. The client application developer is responsible for catching these checked exceptions in a `try`/`catch` block. The developer should also try to provide the appropriate application functionality to recover from such exceptions. For example, an order tracking client might include the code shown in Code Example 4.15 to handle cases where a matching order is not found. The order tracking service threw an `OrderNotFoundException`, and the client presented the user with a GUI dialog indicating that the order was not found.

```
try {
   OrderDetails od = stub.getOrderDetails(orderId);
} catch (OrderNotFoundException onx) {
   JOptionPane.showMessageDialog(gui,
      "Order Not found with Order ID " + orderId, "Error",
      JOptionPane.ERROR_MESSAGE);
}
```

Code Example 4.15 J2SE Client Displaying an Error

A J2EE Web component client may handle the exception using the facilities provided by the J2EE environment. The client may wrap the application exception and throw an unchecked exception, such as a `javax.servlet.ServletException`, or it may map the exception directly to an error page in the Web deployment descriptor. In Code Example 4.16, the client maps the `OrderNotFoundException` directly to a JSP. Clients should always provide human-readable messages and give the user the ability to return to an entry page.

```
<error-page>
   <exception-type>com.sun.blueprints.adventure.
         OrderNotFoundException</exception-type>
   <location>/order_not_found_exception.jsp</location>
</error-page>
```

Code Example 4.16 Using the Servlet Error Mechanism

Notice that using the servlet error mechanism in this way tightly binds the Web application client to the service. If the service changes the exceptions it throws, or any of its exception parameters, the client is directly affected.

❐ Client developers should isolate service exceptions as much as possible by wrapping them in their own application-specific exceptions to keep the client application from being too closely tied to a service. This is especially important when the service is outside the control of the client developer or if the service changes frequently. A client may require refactoring when a service changes because the stubs and supporting Java object representations of the exceptions were generated statically.

Client developers may also generalize exception handling and instead handle all exceptions in a single point in the application. Keeping exception handling to one place in an application reduces the need to include code for handling exceptions throughout the application.

```
try {
    OrderDetails od = stub.getOrderDetails(orderId);
} catch (OrderNotFoundException onx) {
    RuntimeException re= new RuntimeException(onx);
}
```

Code Example 4.17 Generalized Exception Processing

In Code Example 4.17, the exception thrown as a result of the Web service call is set as the cause of the runtime exception. This technique, known as chaining exceptions, produces a better stack trace and helps to more clearly display the root cause of the exception. Chaining exceptions in conjunction with the servlet error mechanism shown in Code Example 4.16 may provide a generalized means of displaying service exceptions.

Boundary checking can help prevent service exceptions. For example, a client of an order tracking service might first check that an order identifier is a five digit integer, or that its value falls within a certain range. While the JAX-RPC API ensures that types are correct, the application developer is responsible for checking boundary limitations.

Clients in a J2EE environment that present HTML content can use JavaScript to validate boundaries before sending requests to a service. Clients using an HTML interface should ensure that types for entered values are correct, since HTML parameters are all treated as `String` objects.

❒ Do boundary checking and other validations on the client side so that remote calls and round-trips to the server are kept to a minimum.

4.7 General Considerations

There are some general considerations that Web service client developers might want to keep in mind. In particular, developers need to understand how to manage conversational state. Also included in this section are some guidelines for improving or enhancing the user experience, and a short discussion of server-side design considerations. Last, there is an explanation of how to package client applications.

4.7.1 Managing Conversational State

Clients should view Web services as stateless, and should assume that a service endpoint retains no knowledge of previous client interactions. However, some use cases may require a client to make a sequence of calls to a service to accomplish a given operation.

It is best if the client application maintains state when circumstances require this. The client may manipulate this retained state when not online. There are ways for a client to identify itself across multiple calls to a service and manage its state to keep it in sync with state on the server.

4.7.1.1 Coordinating State With a Service Endpoint

There are certain situations where the client may want to have its conversational state managed by the service endpoint. A client application, for example, may have only minimal processing capabilities and insufficient memory resources to adequately store its conversational state during a session interaction with a service. For a service endpoint to maintain conversational state with its clients, the endpoint must be designed with this in mind.

Often, such endpoints are designed to use a unique, nonreplicable token to identify communication from a specific client, much like browsers use the cookie mechanism. The service endpoint and the client application pass the token between them with each call during their conversation. Code Example 4.18 shows the definition of an order management service endpoint's methods for retrieving and updating a purchase order. Each method includes a token parameter, `clientToken`, that identifies the specific client. The client application creates a unique token and passes it to the service endpoint when invoking these methods. When it receives the method invocation, the service endpoint identifies the client and persists its state. In this example, notice that the endpoint can identify the purchase order because the same client token is passed when retrieving and updating the purchase order.

```
public interface OrderManagementSEI extends Remote {
    public void updatePurchaseOrder(PurchaseOrder po,
            String clientToken) throws RemoteException;
    public PurchaseOrder getPurchaseOrder(String id,
            String clientToken) throws RemoteException;
}
```

Code Example 4.18 Service Endpoint Interface Methods with Unique Tokens

When an EJB component is the basis for a service endpoint, the EJB component can persist conversational state to a data store during the session. The service endpoint may still be designed to use tokens, and such tokens may represent the primary key of an entity bean. In this case, the endpoint designer must be sure to clean up stale session data from the persistent store. This can be done using a time stamp on the entity bean holding the conversational state and a timer bean to track elapsed time. An endpoint that does not properly clean up stale session data might eventually persist a large amount of data.

A client developer whose application interacts with an endpoint that stores conversational state needs to have detailed knowledge of the endpoint's token requirements and the state maintained by the endpoint. In addition, the developer needs detailed knowledge of the timeout, if one exists. This coordination between the client and endpoint makes them tightly coupled. As such, developers should use this approach only when a tightly-coupled situation is acceptable or required, or when the client and service endpoint responsibilities are clearly documented.

4.7.1.2 Synchronizing Conversational State Among Clients

There are times when multiple Web service clients share conversational state among themselves. Such shared state may be read only or it may be state that is updated. Read-only state does not require synchronization—since clients are retrieving data only, they may poll the service at intervals and obtain current data. However, it is possible for other Web clients to concurrently update shared data. Data consistency is an issue when one or more clients may be manipulating the same data at the same time.

An example use case that illustrates sharing modifiable state among Web service clients might be purchase order management client application. The application accesses a set of purchase orders that require manual approval. After the user manually processes these orders, the application uploads them to the service. It is conceivable that two such client applications may each download some of the same purchase orders, and some of these orders may be changed independent of each other. After the first client application uploads its changed data to the service, there is data consistency problem if the second application tries to upload its modified data.

To prevent such inconsistencies, the client developer should keep certain considerations in mind. The developer can detect data inconsistency using techniques such as time stamp checking or checksum analysis before uploading modified data, and this may require cooperation from the service endpoint. Data locking is another technique to avoid simultaneous changes to the same data. Data locking is not recommended when data changes frequently, especially when the same set of data is updated often by a large number of clients. To implement this, a service endpoint could throw an application-specific exception indicating that data has been checked out by another client and is unavailable. The client application can be designed to overwrite the service endpoint copy or it can ask the user to re-enter the data. Both client and service endpoint developers need to ensure that their respective applications handle such inconsistencies properly.

4.7.2 Enhancing User Experience

The dynamics of the user interface play a large role in determining the quality of a user's experience. J2SE clients have the advantage of drawing on a rich set of APIs, in particular the Swing APIs, to make the user experience the highest quality possible. These APIs give J2SE clients the ability to query a Web service in the background, by invoking the Web service in a different thread, and then updating the user interface when the information is received. The client is able to continue its interaction with the user until the service returns the information. With other types of clients, the user is often left with what appears to be a frozen screen or a nonfunctional or locked application, since the client application blocks during the call to the Web service. In short, the user does not know if the application is still alive.

A J2SE client can use the `SwingUtilities.invokeLater` method to make the call to the service in a separate thread, and thus to achieve a better user experience. (See Code Example 4.19.) This method takes a single argument of an object implementing the `Runnable` interface. When the `invokeLater` method is invoked, the J2SE platform creates another thread to run in background mode by invoking the `run` method on the `Runnable` class.

```
private void trackOrder() {
    setStatus("Tracking Order " + getOrderId());
    GetOrderDetails gd = new GetOrderDetails(this);
    SwingUtilities.invokeLater(gd);
}

class GetOrderDetails implements Runnable {
    private OTClientGUI gui;
    private String orderId;
    boolean done = false;

    GetOrderDetails(OTClientGUI gui) {
        this.gui = gui;
        this.orderId = gui.getOrderId();
    }

public void run() {
    if (!done) {
        try {
            gui. setStatus("Looking for Order " + orderId);
            OrderDetails od = WSProcessor.getOrderDetails(orderId);
            if (od != null) {
                gui.setDetails(od);
            }
        } catch (OrderNotFoundException ex) {
            gui.clearDetails();
            gui.setStatus("");
            JOptionPane.showMessageDialog(gui,
                    "Order Not found with Order ID " + orderId,
                    "Error",
                    JOptionPane.ERROR_MESSAGE);
            } catch (Exception ex) {
                // do nothing for now
            }
            gui.setStatus("Completed lookup for order ID " + orderId);
            SwingUtilities.invokeLater(this);
            done = true;
            }
        }
    }
```

Code Example 4.19 J2SE Client Using `SwingUtils.invokeLater`

This example illustrates how a J2SE client can use other J2SE platform APIs to enhance the user experience. In this example, the user has selected an option to track an order currently being handled by the Web service. That option calls the `invokeLater` method, which in turn invokes a class implementing the `Runnable` interface. This class, using a call back mechanism, updates the user with messages indicating the status of the Web service call and details pertaining to the order being tracked. If the order is not found, the code handles displaying to the user a suitable error message indicating the exception.

(More information is available on the `SwingUtilities` API, along with using threads and GUIS with J2SE clients. You can find this information at `http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html`.)

4.7.3 Server-Side Design Considerations for Clients

It should be clear by now that client developers and service endpoint providers do not operate in a vacuum. Providing pre-packaged bindings and server-side documentation gives client developers not only standardized access to a service but also a better understanding of the service endpoint, and thus helps them to develop better clients.

Providing bindings as a pre-packaged, self-contained Web service library with all necessary files to access a service gives client developers easy access to the Web service. An organization that uses pre-packaged bindings can modularize access to its Web services. Client developers also do not have to generate the service endpoint interface and dependent classes to access the service.

More extensive HTML documentation of a service covering parameters required by the service, return values, and exceptions helps developers better interact with the service. This information should go beyond what is included in the WSDL document.

4.7.4 Packaging

To access a service, a standalone client requires a runtime environment. For J2SE clients, the runtime must be packaged with the application. J2EE clients rely on the JAX-RPC runtime.

J2ME clients do not need to package the JAX-RPC runtime with the applications. Although stubs do need to be packaged with an application, the stubs are portable across JAX-RPC runtimes. The portability of stubs is critical because J2ME clients cannot generate or compile stub implementation code, and thus must rely on more dynamic provisioning.

This section discusses packaging issues for the different types of clients.

4.7.4.1 J2EE Clients

Web service clients running in a J2EE environment require some basic artifacts, as follows:

- Service reference—A `service-ref` element in the deployment descriptor
- Mapping file—A JAX-RPC mapping file
- WSDL document
- Service endpoint interface—A stub or dynamic proxy
- Generated classes

The `service-ref` element, part of the general J2EE 1.4 schema, contains information about a service. Web, EJB, and J2EE application client module deployment descriptors use this element to locate the JAX-RPC mapping files as well as the service's WSDL file. The service reference element maps a service to a JNDI resource name and also specifies the service endpoint interface for those clients using stubs and dynamic proxies. (Clients using DII do not need to specify the service endpoint interface.) It also specifies the WSDL file for the service (its location is given relative to the root of the package) and the qualified name for the service in the WSDL file. If a WSDL file is required, the element specifies a JAX-RPC mapping file. The mapping file's location is also relative to the package root. Code Example 4.20 is an example of a service reference:

```
<service-ref>
   <description>OPC OT Service Client</description>
   <service-ref-name>service/OpcOrderTrackingService
   </service-ref-name>
   <service-interface>
      com.sun.j2ee.blueprints.adventure.web.actions.
         OpcOrderTrackingService
   </service-interface>
   <wsdl-file>WEB-INF/wsdl/OpcOrderTrackingService.wsdl
   </wsdl-file>
   <jaxrpc-mapping-file>WEB-INF/opc-ot-jaxrpc-mapping.xml
   </jaxrpc-mapping-file>
   <service-qname
      xmlns:servicens="urn:OpcOrderTrackingService">
         servicens:OpcOrderTrackingService
   </service-qname>
</service-ref>
```

Code Example 4.20 web.xml Fragment for Web Service Reference

The JAX-RPC mapping file specifies the package name containing the generated runtime classes and defines the namespace URI for the service. (See Code Example 4.21.)

```
<java-wsdl-mapping xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee ht
   tp://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd"
version="1.1">
<package-mapping>
   <package-type>com.sun.j2ee.blueprints.adventure.web.actions
</package-type>
<namespaceURI>urn:OpcOrderTrackingService</namespaceURI>
```

```
</package-mapping>
</java-wsdl-mapping>
```

Code Example 4.21 JAX-RPC Mapping File

WSDL files, including partial WSDL files, are packaged within clients. Their location is dependent on the type of module. Since clients using DII do not require a WSDL file, they leave the `wsdl-file` element portion of the `service-ref` element undefined and they must not specify the `jaxrpc-mapping-file` element.

For a web application archive (WAR) file, the WSDL file is in the `WEB-INF/wsdl` directory. (See Figure 4.9.) For an EJB endpoint as well as a J2EE application client, the WSDL file is in the directory `MET-INF/wsdl`. (See Figure 4.10.) Both directories are relative to the root directory of the application module.

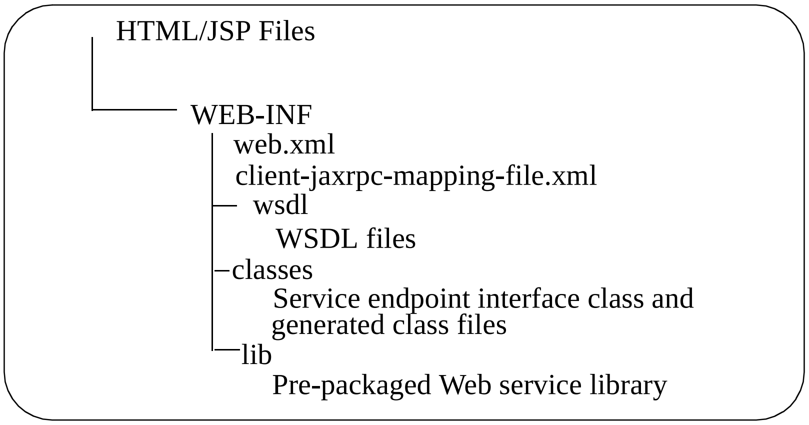
HTML/JSP Files
WEB-INF
web.xml
client-jaxrpc-mapping-file.xml
wsdl
WSDL files
classes
Service endpoint interface class and generated class files
lib
Pre-packaged Web service library

Figure 4.9 Web Application Module Packaging

For Web tier clients, a `service-ref` element in the `web.xml` file contains the location of the JAX-RPC mapping file, `client-jaxrpc-mapping-file.xml`. The service endpoint interface (if provided) is either a class file in the `WEB-INF/classes` directory or it is packaged in a JAR file in the `WEB-INF/lib` directory. Generated classes are located in the same directory.

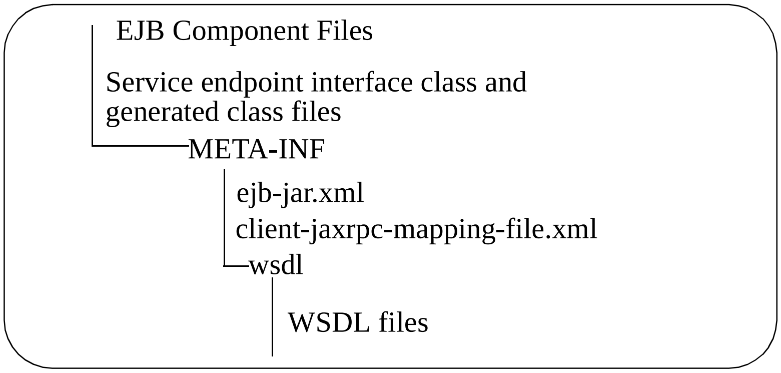
EJB Component Files
Service endpoint interface class and generated class files
META-INF
ejb-jar.xml
client-jaxrpc-mapping-file.xml
wsdl
WSDL files

Figure 4.10 EJB Module Packaging

For EJB tier client components, the `service-ref` element is defined in the deployment descriptor of the `ejb-jar.xml` file and the `client-jaxrpc-mapping-file.xml` mapping file. The WSDL files are in a `META-INF/wsdl` directory. The service endpoint interface as well as generated class files are stored in the module's root directory.

Note that you should not package generated stubs and supporting classes with the tool that generated the stubs, and do not do these tasks manually. The client developer should ensure that the `resource-ref` definition in the client deployment descriptor is correct and that the JAX-RPC mapping file is packaged in the correct module.

4.7.4.2 J2SE Clients

J2SE clients using stubs or dynamic proxies should package the service endpoint interface with the application client and they should be referenced by the class path attribute of the package's manifest file. J2SE clients also must provide a JAX-RPC runtime. For example, a J2SE client is packaged along with its supporting classes or with references to these classes. The service endpoint interface as well as the necessary generated files may be provided in a separate JAR file.

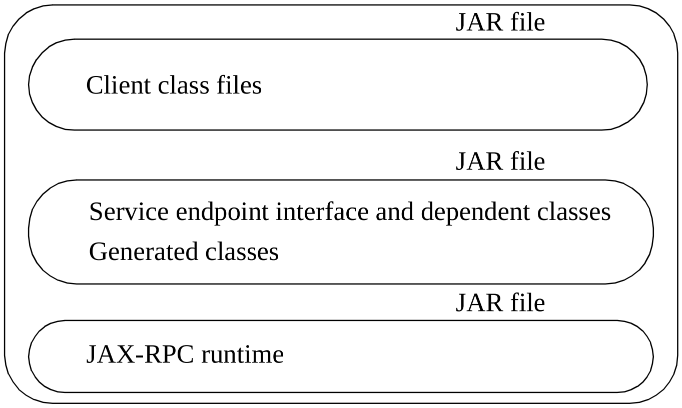

Figure 4.11 Packaging a J2SE Client with Web Service Library

Figure 4.11 shows how to package a J2SE client in a modular manner. The classes specific for Web service access are kept in a separate JAR file referenced via a class path dependency. Packaged this way, a client can swap out the service endpoint access without having to change the core client code. The service access classes may also be shared by different application clients accessing the same service. A developer utilizing a pre-packaged service interface may also be able to develop a Web service client with less knowledge of a service.

4.7.4.3 J2ME Clients

Two optional packages, both of which are extensions of the J2ME platform, enable Web services in the J2ME platform by providing runtime support for XML processing and JAX-RPC communication. (Note that although XML processing capabilities are not provided in the J2ME platform, they are required for JAX-RPC communication).

A J2ME client application developer must package certain resources with a J2ME application. First, J2ME applications are packaged in a MIDlet format. A MIDlet is a Java Archive (JAR) file that contains class files, application resources, and a manifest file (`manifest.mf`), which contains application attributes. A developer may also provide an external Java Application Descriptor (JAD) file for the MIDlet. A JAD file provides the J2ME environment with additional information about the application, such as the location of the application's MIDlet file. A JAD file's attributes mirror those found in the manifest file, but the JAD file takes precedence over the manifest file. Furthermore, a developer or deployer may override application attributes in the JAD file. Figure 4.12 describes the packaging for a Web service MIDlet client application.

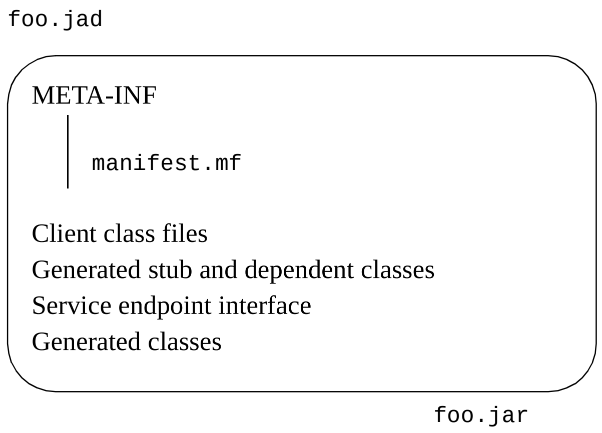

Figure 4.12 Packaging a MIDlet Web Service Client Application

The `foo.jar` MIDlet file contains the client application classes and the respective artifacts generated by the J2ME Web service development tools, as well as a manifest file. A foo.jad file describes the `foo.jar` MIDlet. Similar to the J2EE platform, the J2ME platform with the optional Web service packages provides the resources required for Web service communication.

4.8 Conclusion

This chapter described the different types of clients that may access Web services and the issues faced by developers of these clients. It highlighted the differences between clients running in a J2EE environment from clients running in the J2SE and J2ME environments, and how these differences affect the design of a client application.

The chapter described the three communication modes available to clients: stubs, dynamic proxy, and dynamic invocation interface. These three modes form the basis for a client's access to a Web service. Within the parameters of these communication modes, the chapter discussed the steps for implementing Web service clients. It provided detailed discussions from how to locate and access a service through handling errors thrown by a service.

The chapter also described how to package different types of clients and how to manage conversational state, particularly keeping state synchronized among multiple clients and between the client and the service. It also provided guidelines for improving the overall end user experience.

The next chapter talks about XML in depth, since XML places a large role in Web services.