

CHAPTER 3

The Client Tier

by Ray Ortigas

FROM a developer's point of view, a J2EE application can support many types of clients. J2EE clients can run on laptops, desktops, palmtops, and cell phones. They can connect from within an enterprise's intranet or across the World Wide Web, through a wired network or a wireless network or a combination of both. They can range from something thin, browser-based and largely server-dependent to something rich, programmable, and largely self-sufficient.

From a user's point of view, the client *is* the application. It must be useful, usable, and responsive. Because the user places high expectations on the client, you must choose your client strategy carefully, making sure to consider both technical forces (such as the network) and non-technical forces (such as the nature of the application). This chapter presents guidelines for designing and implementing J2EE clients amidst these competing forces.

This chapter cites examples from the Java Pet Store sample application, an online outlet for selling pets, and the Java Smart Ticket sample application, an e-commerce movie ticket service. The code for these sample applications is available on the Java BluePrints Web site. See "References and Resources" on page 73 for more information.

3.1 Client Considerations

Every application has requirements and expectations that its clients must meet, constrained by the environment in which the client needs to operate.

Your users and their usage patterns largely determine what type of client or interface you need to provide. For example, desktop Web browser clients are

popular for e-mail and e-shopping because they provide a familiar interface. For another example, wireless handheld clients are useful for sales force automation because they provide a convenient way to access enterprise resources from the field in real time. Once you have decided what type of interface you need, you should design your client configuration with network, security, and platform considerations in mind.

3.1.1 Network Considerations

J2EE clients may connect to the enterprise over a wide array of networks. The quality of service on these networks can vary tremendously, from excellent on a company intranet, to modest over a dialup Internet connection, to poor on a wireless network. The connectivity can also vary; intranet clients are always connected, while mobile clients experience intermittent connectivity (and are usually online for short periods of time anyway).

Regardless of the quality of service available, you should always keep in mind that the client *depends* on the network, and the network is imperfect. Although the client *appears* to be a stand-alone entity, it cannot be programmed as such because it is part of a distributed application. Three aspects of the network deserve particular mention:

- Latency is non-zero.
- Bandwidth is finite.
- The network is not always reliable.

A well-designed enterprise application must address these issues, starting with the client. The ideal client connects to the server only when it has to, transmits only as much data as it needs to, and works reasonably well when it cannot reach the server. Later, this chapter elaborates on strategies for achieving those goals.

3.1.2 Security Considerations

Different networks have different security requirements, which constrain how clients connect to an enterprise. For example, when clients connect over the Internet, they usually communicate with servers through a firewall. The presence of a firewall that is not under your control limits the choices of protocols the client can use. Most firewalls are configured to allow Hypertext Transfer Protocol (HTTP) to pass across,

but not Internet Inter-Orb Protocol (IIOP). This aspect of firewalls makes Web-based services, which use HTTP, particularly attractive compared to RMI- or CORBA-based services, which use IIOP.

Security requirements also affect user authentication. When the client and server are in the same security domain, as might be the case on a company intranet, authenticating a user may be as simple as having the user log in only once to obtain access to the entire enterprise, a scheme known as *single sign on*. When the client and server are in different security domains, as would be the case over the Internet, a more elaborate scheme is required for single sign on, such as that proposed by the Liberty Alliance, an industry collaboration spearheaded by Sun Microsystems.

The authentication process itself needs to be confidential and, usually, so does the client-server communication after a user has been authenticated. Both the J2EE platform and the client types discussed in this chapter have well-defined mechanisms for ensuring confidentiality. These mechanisms are discussed in Chapter 9.

3.1.3 Platform Considerations

Every client platform's capabilities influence an application's design. For example, a browser client cannot generate graphs depicting financial projections; it would need a server to render the graphs as images, which it could download from the server. A programmable client, on the other hand, could download financial data from a server and render graphs in its own interface.

Another aspect of the platform to consider is form factor. Desktop computers offer a large screen, a keyboard, and a pointing device such as a mouse or trackball. With such clients, users are willing to view and manipulate large amounts of data. In contrast, cell phones have tiny screens and rely on button-based interactions (usually thumb-operated!). With such clients, users can't (and don't want to) view or manipulate large amounts of data.

Applications serving multiple client platforms pose additional challenges. Developing a client for each platform requires not only more resources for implementation, testing, and maintenance but also specialized knowledge of each platform. It may be easier to develop one client for all platforms (using a browser- or a Java technology-based solution, for example), but designing a truly portable client requires developers to consider the lowest common denominator. Consequently, such a client implementation cannot take advantage of the various capabilities unique to each platform.

3.2 General Design Issues and Guidelines

While the J2EE platform encourages thin-client architectures, J2EE clients are not dumb. A J2EE client may handle many responsibilities, including:

- **Presenting the user interface**—Although a client presents the views to a user, the logic for the views may be programmed on the client or downloaded from a server.
- **Validating user inputs**—Although the EIS and EJB tier must enforce constraints on model data (since they contain the data), a client may also enforce data constraints by validating user inputs.
- **Communicating with the server**—When a user requests functionality that resides on a server, the user's client must present that request to the server using a protocol they both understand.
- **Managing conversational state**—Applications need to track information as a user goes through a workflow or process (effectively conversing with the application). The client may track none, some, or all of this information, known as conversational state.

How you handle these responsibilities on your client can significantly impact your development efforts, your application's performance, and your users' experience. Generally, the more responsibilities you place on the client, the more responsive it will be.

The next two sections consider browser clients and Java clients separately. You do not have to pick one or the other; a J2EE application can accommodate both browser and Java clients. The Java Pet Store sample application, for example, has a Web browser interface for shoppers and a Java application for administrators. Section 4.4.2.2 on page 107 explains how to design the Web tier to support multiple types of clients.

3.3 Design Issues and Guidelines for Browser Clients

Browsers are the thinnest of clients; they display data to their users and rely on servers for application functionality.

From a deployment perspective, browser clients are attractive for a couple of reasons. First, they require minimal updating. When an application changes,

server-side code has to change, but browsers are almost always unaffected. Second, they are ubiquitous. Almost every computer has a Web browser and many mobile devices have a microbrowser.

This section documents the issues behind designing and implementing browser clients.

3.3.1 Presenting the User Interface

Browser clients download documents from a server. These documents contain data as well as instructions for presenting that data. The documents are usually dynamically generated by JSP pages (and less often by Java servlets) and written in a presentational markup language such as Hypertext Markup Language (HTML). A presentational markup language allows a single document to have a reasonable presentation regardless of the browser that presents it. These screenshots in Figure 3.1 show the Java Pet Store sample application running in two different browsers.



Figure 3.1 Java Pet Store Sample Application Shopping Client Rendered by Two Different Browsers

There are other alternatives to HTML, particularly for mobile devices, whose presentation capabilities tend to differ from those of a traditional desktop computer. Examples include Wireless Markup Language (WML), Compact HTML (CHTML), Extensible HTML (XHTML) Basic, and Voice Markup Language (VoiceML).

Browsers have a couple of strengths that make them viable enterprise application clients. First, they offer a familiar environment. Browsers are widely deployed and used, and the interactions they offer are fairly standard. This makes browsers popular, particularly with novice users. Second, browser clients can be easy to implement. The markup languages that browsers use provide high-level

abstractions for how data is presented, leaving the mechanics of presentation and event-handling to the browser.

The trade-off of using a simple markup language, however, is that markup languages allow only limited interactivity. For example, HTML's tags permit presentations and interactions that make sense only for hyperlinked documents. You can enhance HTML documents slightly using technologies such as JavaScript in combination with other standards, such as Cascading Style Sheets (CSS) and the Document Object Model (DOM). However, support for these documents, also known as Dynamic HTML (DHTML) documents, is inconsistent across browsers, so creating a portable DHTML-based client is difficult.

Another, more significant cost of using browser clients is potentially low responsiveness. The client depends on the server for presentation logic, so it must connect to the server whenever its interface changes. Consequently, browser clients make many connections to the server, which is a problem when latency is high. Furthermore, because the responses to a browser intermingle presentation logic with data, they can be large, consuming substantial bandwidth.

3.3.2 Validating User Inputs

Consider an HTML form for completing an order, which includes fields for credit card information. A browser cannot single-handedly validate this information, but it can certainly apply some simple heuristics to determine whether the information is invalid. For example, it can check that the cardholder name is not null, or that the credit card number has the right number of digits. When the browser solves these obvious problems, it can pass the information to the server. The server can deal with more esoteric tasks, such as checking that the credit card number really belongs to the given cardholder or that the cardholder has enough credit.

When using an HTML browser client, you can use the JavaScript scripting language, whose syntax is close to that of the Java programming language. Be aware that JavaScript implementations vary slightly from browser to browser; to accommodate multiple types of browsers, use a subset of JavaScript that you know will work across these browsers. (For more information, see the *ECMA-Script Language Specification*.) It may help to use JSP custom tags that autogenerate simple JavaScript that is known to be portable.

Code Example 3.1 shows how to validate a Web form using JavaScript's DOM hooks to access the form's elements. For example, suppose you have a form for creating an account. When the user submits the form, it can call a JavaScript function to validate the form.

```
<form name="account_form" method="POST"
action="http://acme.sun.com/create_account"
onSubmit="return
checkFamilyName();">
<p>Family name: <input type="text" name="family_name"></p>
<!-- ... -->
<p><input type="submit" value="Send it!" /></p>
</form>
```

Code Example 3.1 HTML Form Calling a JavaScriptValidation Function

Code Example 3.2 shows how the JavaScript validation function might be implemented.

```
<script language="JavaScript">
<!--
function checkFamilyName() {
    var familyName =
        window.document.account_form.family_name.value;
    if (familyName == "") {
        alert("You didn't enter a family name.");
        return false;
    }
    else {
        return true;
    }
}
-->
</script>
```

Code Example 3.2 JavaScript Validation Function Using DOM Hooks

Validating user inputs with a browser does not necessarily improve the responsiveness of the interface. Although the validation code allows the client to instantly report any errors it detects, the client consumes more bandwidth because it must download the code in addition to an HTML form. For a non-trivial form, the amount of validation code downloaded can be significant. To reduce download

time, you can place commonly-used validation functions in a separate source file and use the `SCRIPT` element's `SRC` attribute to reference this file. When a browser sees the `SRC` attribute, it will cache the source file, so that the next time it encounters another page using the same source file, it will not have to download it again.

Also note that implementing browser validation logic will duplicate some server-side validation logic. The EJB and EIS tiers should validate data regardless of what the client does. Client-side validation is an optimization; it improves user experience and decreases load, but you should never rely on the client exclusively to enforce data consistency.

3.3.3 Communicating with the Server

Browser clients connect to a J2EE application over the Web, and hence they use HTTP as the transport protocol.

When using browser interfaces, users generally interact with an application by clicking hyperlinked text or images, and completing and submitting forms. Browser clients translate these gestures into HTTP requests for a Web server, since the server provides most, if not all, of an application's functionality.

User requests to retrieve data from the server normally map to HTTP GET requests. The URLs of the requests sometimes include parameters in a query string that qualify what data should be retrieved. For example, a URL for listing all dogs might be written as follows:

```
http://javapetstore.sun.com/product.screen?category_id=DOGS
```

User requests to update data on the server normally map to HTTP POST requests. Each of these requests includes a MIME envelope of type `application/x-www-form-urlencoded`, containing parameters for the update. For example, a POST request to complete an order might use the URL:

```
http://javapetstore.sun.com/cart.do
```

The body of the request might include the following line:

```
action=add&itemId=EST-27
```

The servlet API provides a simple interface for handling incoming GET and POST requests and for extracting any parameters sent along with the requests.

Section 4.4.2 on page 98 describes strategies for handling requests and translating these requests into events on your application model.

After a server handles a client request, it must send back an HTTP response; the response usually contains an HTML document. A J2EE application should use JSP pages to generate HTML documents; for more information on using JSP pages effectively, see Section 4.2.6.4 on page 86.

Security is another important aspect of client-server communication. Section 9.2.2 on page 284 covers authentication mechanisms and Section 9.4.2 on page 305 covers confidentiality mechanisms.

3.3.4 Managing Conversational State

Because HTTP is a request-response protocol, individual requests are treated independently. Consequently, Web-based enterprise applications need a mechanism for identifying a particular client and the state of any conversation it is having with that client.

The *HTTP State Management Mechanism* specification introduces the notion of a *session* and *session state*. A session is a short-lived sequence of service requests by a single user using a single client to access a server. Session state is the information maintained in the session across requests. For example, a shopping cart uses session state to track selections as a user chooses items from a catalog. Browsers have two mechanisms for caching session state: cookies and URL rewriting.

- A *cookie* is a small chunk of data the server sends for storage on the client. Each time the client sends information to a server, it includes in its request the headers for all the cookies it has received from that server. Unfortunately, cookie support is inconsistent enough to be annoying: some users disable cookies, some firewalls and gateways filter them, and some browsers do not support them. Furthermore, you can store only small amounts of data in a cookie; to be portable across all browsers, you should use four kilobytes at most.
- *URL rewriting* involves encoding session state within a URL, so that when the user makes a request on the URL, the session state is sent back to the server. This technique works almost everywhere, and can be a useful fallback when you cannot use cookies. Unfortunately, pages containing rewritten URLs consume much bandwidth. For each request the server receives, it must rewrite

every URL in its response (the HTML page), thereby increasing the size of the response sent back to the client.

Both cookies and pages containing rewritten URLs are vulnerable to unauthorized access. Browsers usually retain cookies and pages in the local file system, so any sensitive information (passwords, contact information, credit card numbers, etc.) they contain is vulnerable to abuse by anyone else who can access this data. Encrypting the data stored on the client might solve this problem, as long as the data is not intended for display.

Because of the limitations of caching session state on browser clients, these clients should not maintain session state. Rather, servers should manage session state for browsers. Under this arrangement, a server sends a browser client a key that identifies session data (using cookies or URL rewriting), and the browser sends the key back to the server whenever it wants to use the session data. If the browser caches any information beyond a session key, it should be restricted to items like the user's login and preferences for using the site; such items do not need to be manipulated, and they can be easily stored on the client.

3.4 Design Issues and Guidelines for Java Clients

Java clients can be divided into three categories: *applications*, *applets*, and *MIDlets*. They all leverage the Java programming language and a small common set of Java libraries, but they are deployed differently.

3.4.0.1 Application Clients

Application clients execute in the Java 2 Runtime Environment, Standard Edition (JRE). They are very similar to the stand-alone applications that run on traditional desktop computers. As such, they typically depend much less on servers than do browsers.

Application clients are packaged inside JAR files and may be installed explicitly on a client's machine or provisioned on demand using Java Web Start technology. Preparing an application client for Java Web Start deployment involves distributing its JAR with a Java Network Launching Protocol (JNLP) file. When a user running Java Web Start requests the JNLP file (normally by clicking a link in a Web browser), Java Web Start automatically downloads all necessary files. It then caches the files so the user can relaunch the application without having to

download them again (unless they have changed, in which case Java Web Start technology takes care of downloading the appropriate files).

For more information on Java Web Start and JNLP, see the Java Web Start home page listed in “References and Resources” on page 73.

3.4.0.2 Applet Clients

Applet clients are user interface components that typically execute in a Web browser, although they can execute in other applications or devices that support the applet programming model. They are typically more dependent on a server than are application clients, but are less dependent than browser clients.

Like application clients, applet clients are packaged inside JAR files. However, applets are typically executed using Java Plug-in technology. This technology allows applets to be run using Sun’s implementation of the Java 2 Runtime Environment, Standard Edition (instead of, say, a browser’s default JRE).

For more information on packaging applets, consult the *Java Tutorial*. For more information on serving applets from JSP pages using Java Plug-in technology, consult the *J2EE Tutorial*.

3.4.0.3 MIDlet Clients

MIDlet clients are small applications programmed to the Mobile Information Device Profile (MIDP), a set of Java APIs which, together with the Connected Limited Device Configuration (CLDC), provides a complete Java 2 Micro Edition (J2ME) runtime environment for cellular phones, two-way pagers, and palmtops.

A MIDP application is packaged inside a JAR file, which contains the application’s class and resource files. This JAR file may be pre-installed on a mobile device or downloaded onto the device (usually over the air). Accompanying the JAR file is a Java Application Descriptor (JAD) file, which describes the application and any configurable application properties.

For a complete specification of a JAD file’s contents, as well as deploying MIDP applications in general, see the *J2ME Wireless Toolkit User’s Guide*.

3.4.1 Presenting the User Interface

Although a Java client contains an application’s user interface, the presentation logic behind this interface may come from a server, as it would for a browser, or it may be programmed from the ground up on the client. In this section, we discuss the latter case.

Java applet and application clients may use the Java Foundation Classes (JFC)/Swing API, a comprehensive set of GUI components for desktop clients. Java MIDlets, meanwhile, may use the MIDP User Interface API, a GUI toolkit that is geared towards the limited input capabilities of today's mobile information devices. For example, Figure 3.2 shows the Java Smart Ticket sample application using the MIDP UI API and running on a Palm IIIc emulator.



Figure 3.2 Java Smart Ticket Sample Application Client Running on a Palm OS Device

Implementing the user interface for a Java client usually requires more effort to implement than a browser interface, but the benefits are substantial. First, Java client interfaces offer a richer user experience; with programmable GUI components, you can create more natural interfaces for the task at hand. Second, and perhaps more importantly, full programmability makes Java clients much more responsive than browser interfaces.

When a Java client and a browser client request the same data, the Java client consumes less bandwidth. For example, when a browser requests a list of orders and a Java client requests the same list, the response is larger for the browser because it includes presentation logic. The Java client, on the other hand, gets the data and nothing more.

Furthermore, Java clients can be programmed to make fewer connections than a browser to a server. For example, in the Java Pet Store sample application, an administrator may view orders in a table and sort them by date, order identifier, and so on. He or she may also see order data presented in a pie chart or a bar chart, as shown in Figure 3.3.

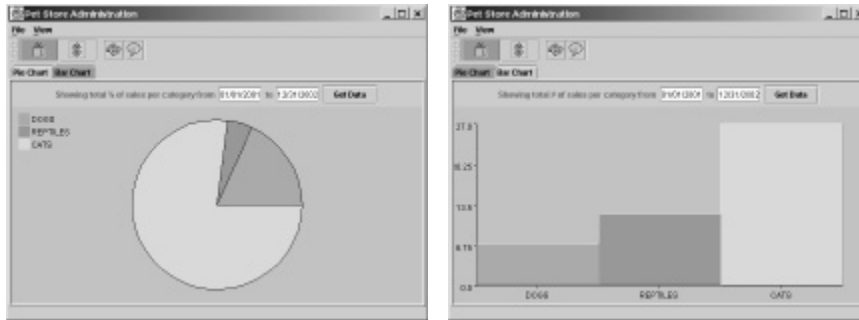


Figure 3.3 Java Pet Store Sample Application Administrator Client Displaying Order Data in Pie and Bar Charts Using the JFC/Swing API

Because the administrator client uses the JFC/Swing API, it can provide all of these views from the same data set; once it retrieves the data, it does not have to reconnect to the server (unless it wants to refresh its data). In contrast, an administrator client implemented using a browser must connect to the server each time the view changes. Even though the data does not change, the browser has to download a new view because the data and the view are intertwined.

For more information on programming JFC/Swing user interfaces, refer to the *JFC Swing Tutorial*. For more information on programming MIDP user interfaces, read *Programming Wireless Devices with the Java 2 Platform, Micro Edition*.

3.4.2 Validating User Inputs

Like presentation logic, input validation logic may also be programmed on Java clients, which have more to gain than browser clients from client-side input validation. Recall that browser clients have to trade off the benefit of fewer connections (from detecting bad inputs before they get to the server) for the cost of using more bandwidth (from downloading validation code from the server). In contrast, Java clients realize a more responsive interface because they do not have to download validation logic from the server.

With Java clients, it is straightforward to write input validation logic. You use the Java programming language, as shown in Code Example 3.3 from the Java Smart Ticket sample application:

```
public void validateAll() throws ApplicationException {
    if (username.size() < 4) {
        /* Complain about username being too short... */
    }
    if (password.size() < 6) {
        /* Complain about password being too short... */
    }
    if (zipCode.size() != 5) {
        /* Complain about ZIP code not having 5 characters... */
    }
    if (creditCard.size() != 12) {
        /* Complain about credit card number not having
        12 digits... */
    }
}
```

Code Example 3.3 Java Smart Ticket Sample Application Code for Validating Inputs in a User Account Form

For more sophisticated input validation on JFC/Swing clients, consider using the `InputVerifier` class provided by the JFC/Swing framework. For more information, see “References and Resources” on page 73.

Of course, the best way to reduce client-side validation requirements is to make it impossible to enter bad data in the first place (especially if you are expecting a value of an enumerated type). For example, using a text field to enter a date

is error-prone because a text field can receive many types of input. Providing a set of drop-downs that contain valid months, days, and years might be an improvement, but a user can still enter invalid input (such as Feb. 30). The best solution is to provide a calendar widget that intelligently constrains what date is chosen, and the only way to implement such a custom component is with a programmable client.

3.4.3 Communicating with the Server

Java clients may connect to a J2EE application as *Web clients* (connecting to the Web tier), *EJB clients* (connecting to the EJB tier), or *EIS clients* (connecting to the EIS tier).

3.4.3.0.1 Web Clients

Like browser clients, Java Web clients connect over HTTP to the Web tier of a J2EE application. This aspect of Web clients is particularly important on the Internet, where HTTP communication is typically the only way a client can reach a server. Many servers are separated from their clients by firewalls, and HTTP is one of the few protocols most firewalls allow through.

Whereas browsers have built-in mechanisms that translate user gestures into HTTP requests and interpret HTTP responses to update the view, Java clients must be programmed to perform these actions. A key consideration when implementing such actions is the format of the messages between client and server.

Unlike browser clients, Java clients may send and receive messages in any format. For example, in the Java Smart Ticket sample application, a user may look at a list of movies. If the user had a browser client, the list would have to be formatted in HTML before downloading it to the client. However, the Java client in this demo downloads a plain binary string representing the list.

A Java client could use another format, such as comma-separated values:

```
1,Big and Badder,2,The Dot,4,Invasion of the Dots
```

Or, the client could use key-value pairs:

```
id=1,title="Big and Badder"  
id=2,title="The Dot"  
id=4,title="Invasion of the Dots"
```

Or, the client could use XML:

```
<movies>
  <movie>
    <id>1</id>
    <title>Big and Badder</title>
  </movie>
  <movie>
    <id>2</id>
    <title>The Dot</title>
  </movie>
  <movie>
    <id>4</id>
    <title>Invasion of the Dots</title>
  </movie>
</movies>
```

Although the possibilities are endless, you can think of message formats as falling into a spectrum, with binary strings on one end and XML documents on the other. To understand the tradeoffs of message formats in general, it helps to consider these two extremes.

Binary messages consume little bandwidth. This aspect of binary messages is especially attractive in low-bandwidth environments (such as wireless and dial-up networks), where every byte counts. Code Example 3.4 illustrates how a Java client might construct a binary request to log into an application.

```
static final int LOGIN_USER = 1;
// ...

HttpConnection c;
DataOutputStream out;
String username, password;
/* Construct the body of the HTTP POST request using out... */

out.write(LOGIN_USER);
out.writeUTF(username);
```

```
out.writeUTF(password);
/* Send the HTTP request... */
```

Code Example 3.4 Java Client Code for Sending a Binary Request

Code Example 3.5 illustrates how a Java servlet might listen for requests from the Java client:

```
public void doPost(HttpServletRequest req,
    HttpServletResponse resp) throws IOException, ServletException {

    /* Interpret the request. */
    DataInputStream in =
        new DataInputStream(req.getInputStream());
    int command = in.readInt();

    resp.setContentType("application/binary");
    DataOutputStream out =
        new DataOutputStream(resp.getOutputStream());
    byte commandByte = in.read();
    switch (command) {
    case LOGIN_USER:
        String username = in.readUTF();
        String password = in.readUTF();
        /* Check username and password against user database... */
    }
}
```

Code Example 3.5 Java Servlet Code for Interpreting a Binary Request

These examples also illustrate a substantial cost of HTTP-based messaging in general; you have to write code for parsing and interpreting messages. Unfortunately, writing such code, especially for multiple programmable clients, can be time-consuming and error-prone.

Java technologies for XML alleviate some of the burdens experienced with binary messaging. These technologies, which include the Java API for XML Processing (JAXP), automate the parsing and aid the construction of XML messages.

Messaging toolkits based on Java technology help interpret messages once they are parsed; these toolkits implement open standards such as the Simple Object Access Protocol (SOAP). The ability to parse and interpret messages automatically reduces development time and helps maintenance and testing.

A side benefit of using XML messages is that alternate clients are easier to support, as XML is a widely-accepted open standard. For example, StarOffice Calc and Macromedia Flash clients could both read order data formatted in XML from the same JSP page and present the data in their respective interfaces. Also, you can use XML to encode messages from a variety of clients. A C++ client, for example, could use a SOAP toolkit to make remote procedure calls (RPC) to a J2EE application.

The most common models for XML processing are DOM and the Simple API for XML (SAX). Unlike DOM, which provides an in-memory, tree-based data structure for random access, SAX offers event-based serial access, which makes processing messages faster. For more information on using XML effectively, see “References and Resources” on page 73.

Like browser clients, Java Web clients carry out secure communication over HTTPS. See Section 9.2.2 on page 284 for more information on Web authentication mechanisms and Section 9.4.2 on page 305 for more information on Web confidentiality mechanisms.

3.4.3.0.2 EJB Clients

When using Web clients, you must write code for translating user gestures into HTTP requests, HTTP requests into application events, event responses into HTTP responses, and HTTP responses into view updates. On the other hand, when using EJB clients, you do not need to write such code because the clients connect directly to the EJB tier using Java Remote Method Invocation (RMI) calls.

Unfortunately, connecting as an EJB client is not always possible. First, only applet and application clients may connect as EJB clients. (At this time, MIDlets cannot connect to the EJB tier because RMI is not a native component of MIDP.) Second, RMI calls are implemented using IIOP, and most firewalls usually block communication using that protocol. So, when a firewall separates a server and its clients, as would be the case over the Internet, using an EJB client is not an option. However, you could use an EJB client within a company intranet, where firewalls generally do not intervene between servers and clients.

When deploying an applet or application EJB client, you should distribute it with a client-side container and install the container on the client machine. This container (usually a class library) allows the client to access middle-tier services

(such as the JMS, JDBC, and JTA APIs) and is provided by the application server vendor. However, the exact behavior for installing EJB clients is not completely specified for the J2EE platform, so the client-side container and deployment mechanisms for EJB clients vary slightly from application server to application server.

Clients should be authenticated to access the EJB tier, and the client container is responsible for providing the appropriate authentication mechanisms. For more information on EJB client authentication, see Section 9.2.2.2 on page 287.

3.4.3.0.3 EIS Clients

Generally, Java clients should not connect directly to a J2EE application's EIS tier. EIS clients require a powerful interface, such as the JDBC API, to manipulate data on a remote resource. When this interface is misused (by a buggy client you have implemented or by a malicious client someone else has hacked or built from scratch), your data can be compromised. Furthermore, non-trivial EIS clients must implement business logic. Because the logic is attached to the client, it is harder to share among multiple types of clients.

In some circumstances, it may be acceptable for clients to access the EIS tier directly, such as for administration or management tasks, where the user interface is small or nonexistent and the task is simple and well understood. For example, a simple Java program could perform maintenance on database tables and be invoked every night through an external mechanism.

3.4.4 Managing Conversational State

Whereas browser clients require a robust server-side mechanism for maintaining session state, Java clients can manage session state on their own, because they can cache and manipulate substantial amounts of state in memory. Consequently, Java clients have the ability to work while disconnected, which is beneficial when latency is high or when each connection consumes significant bandwidth.

To support a disconnected operation, a Java client must retrieve enough usable data for the user before going offline. The initial cost of downloading such data can be high, but you can reduce this cost by constraining what gets downloaded, by filtering on user preferences, or requiring users to enter search queries at the beginning of each session. Many applications for mobile devices already use such strategies; they also apply well to Java clients in general.

For example, you could extend the Java Smart Ticket sample application to allow users to download movie listings onto their phones. To reduce the size of the

listings, you could allow users to filter on simple criteria such as genre (some users may not be in the mood for drama) or ZIP code (some users may only want to go to movie theaters within 10 miles of where they live). Users could then browse the personalized lists on their phones without needing to connect to the server until they want to buy a ticket.

Also note that the movie listings are candidates for persistence on the client, since they are updated infrequently, perhaps once every week. The Java Smart Ticket sample application client uses the MIDP Record Management Store (RMS) API to store data locally. Application clients, meanwhile, can use either local files (assuming they have permission) or the Java Native Launching Protocol and API (JNLP) persistence service. (Applets have very limited local storage because they normally use a browser's cookie store, although they can request permission to use local files as well.)



Figure 3.4 Java Smart Ticket Sample Application Listing Movie Information Downloaded onto the Phone

The example of downloading movie listings illustrates a read-only interaction. The client retrieves data from the server, caches it, and does not modify the cached data. There may be times, however, when a Java client needs to update data it

receives from the server and report its changes to the server. To stay disconnected, the client must queue updates locally on the client and only send the batch when the user connects to the server.

In the Java Smart Ticket sample application, the client allows users to pinpoint the exact seats they want to buy. When the user decides what show he or she wants to see, the client downloads the data for the show's seating plan and displays the plan to the user. The plan indicates which seats are available and which have already been taken, as shown in Figure 3.5.



Figure 3.5 Java Smart Ticket Sample Application Displaying an Editable Seating Plan for a Particular Movie Showing

This example highlights two important issues. First, when Java clients manipulate enterprise data, they need to know about the model and some or all of the business rules surrounding the data model. For example, the client must understand the concept of booked and unbooked seats, and model that concept just like the server does. For another example, the client must also prevent users from trying to select booked seats, enforcing a business rule also implemented on the server. Generally, clients manipulating enterprise data must duplicate logic on the

server, because the server must enforce all business rules regardless of what its clients do.

Second, when Java clients manipulate enterprise data, applications need to implement data synchronization schemes. For example, between the time when the user downloads the seating plan and the time when the user decides what seats he or she wants to buy, another user may buy some or all of those seats. The application needs rules and mechanisms for resolving such a conflict. In this case, the server's data trumps the client's data because whoever buys the tickets first—and hence updates the server first—gets the tickets. The application could continue by asking the second user if he or she wants the seats that the first user did not buy. Or, it could refresh the second user's display with an updated seating plan and have the user pick seats all over again.

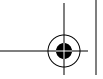
3.5 Summary

The J2EE platform supports a range of client devices and client programming models. Supported devices include desktop systems, laptops, palmtops, cell phones, and various emerging non-traditional devices. The supported programming models include browser clients using HTML and JavaScript, browser plug-in clients such as Flash, office suite clients such as StarOffice, and programmable clients based on Java technologies.

Application developers should make an effort to provide users with the highest possible level of service and functionality supported by each client device. The primary consideration throughout the design of the client should be the network, since the client participates in a networked application. At the same time, there may be other important considerations, such as development and support capabilities, time to market, and other factors that affect the ultimate client solution chosen for a particular application.

3.6 References and Resources

- *The J2EE Tutorial*. S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan. Copyright 2001, Sun Microsystems, Inc.
<<http://java.sun.com/j2ee/tutorial/index.html>>
- *The JFC/Swing Tutorial*. M. Campione, K. Walrath. Copyright 2000, Addison-Wesley. Also available as
<<http://java.sun.com/docs/books/tutorial/uiswing/index.html>>
- *The Java Tutorial, Third Edition: A Short Course on the Basics*. M. Campione, K. Walrath, A. Huml. Copyright 2000, Addison-Wesley. Also available as
<<http://java.sun.com/docs/books/tutorial/index.html>>
- *The Eight Fallacies of Distributed Computing*. P. Deutsch. Copyright 2001, Sun Microsystems, Inc.
<<http://java.sun.com/people/jag/Fallacies.html>>
- *Programming Wireless Devices with the Java 2 Platform, Micro Edition*. R. Riggs, A. Taivalsaari, M. VandenBrink. Copyright 2001, Addison-Wesley.
- *eMobile End-to-End Application Using the Java 2 Platform, Enterprise Edition*. T. Violleau. Copyright 2000, Sun Microsystems, Inc.
<<http://developer.java.sun.com/developer/technicalArticles/javaone00/eMobileApplet.pdf>>
- *Java Technology and XML*. T. Violleau. Copyright 2001, Sun Microsystems, Inc. <<http://developer.java.sun.com/developer/technicalArticles/xml/JavaTechandXML/>>
- *A Note on Distributed Computing*. J. Waldo, G. Wyant, A. Wollrath, S. Kendall. Copyright November 1994, Sun Microsystems, Inc.
<http://research.sun.com/research/techrep/1994/sml_i_tr-94-29.pdf>
- *Cascading Style Sheets Level 2 Specification*. World Wide Web Consortium, May 1998. <<http://www.w3.org/TR/REC-CSS2/>>
- *Document Object Model (DOM) Level 2 Core Specification*. World Wide Web Consortium, November 2000. <<http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/>>
- *ECMAScript Language Specification*. European Computer Manufacturers Association, December 1999. <<ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf>>



- *HTML 4.01 Specification*. World Wide Web Consortium, December 1999.
<<http://www.w3.org/TR/html4/>>
- *Hypertext Transfer Protocol — HTTP/1.1*. The Internet Society, 1999.
<<http://www.ietf.org/rfc/rfc2616.txt>>
- *HTTP State Management Mechanism*. The Internet Society, February 1997.
<<http://www.ietf.org/rfc/rfc2109.txt>>
- Java Web Start Web site <<http://java.sun.com/products/javaweb-start/developers.html>>
- *Input Verification*. Sun Microsystems, 2001.
<<http://java.sun.com/j2se/1.3/docs/guide/swing/InputChanges.html>>
- *Webmonkey*. Lycos, 2001. <<http://webmonkey.com/>>

