CHAPTER 2

The Client Tier

by Ray Ortigas

To an enterprise application user, the client *is* the application. A client makes requests to the server on behalf of the user, and presents the outcomes of those requests to the user. Clients also often reinforce data consistency rules, and may implement some business logic. Therefore, it is important to choose a client configuration that best addresses the requirements of the application and empowers the user with a rich interface.

The J2EE platform supports many types of clients. A J2EE client can run on a laptop or a desktop, or even on a PDA or a cell phone. It can connect over a wired or wireless network, from within an enterprise's intranet, or across the World Wide Web. A J2EE client can range from something very thin and browser-based, to something richer that involves programmable GUI components. Clients can be implemented using a number of languages, such as HTML or Java, or even Visual Basic.

Ultimately, however, the implementation of a J2EE client is constrained by aspects of its environment. These aspects include platform and networking capabilities, which can be hard to control. In this chapter, we present guidelines for designing and implementing J2EE clients in the face of those constraints.

2.1 Client Requirements and Constraints

Every application has requirements and expectations that its clients must meet, constrained by the environment in which the client needs to operate. Platform, networking, and security always play a large role in determining the client configuration. Application designers must also consider non-technical factors such as the availabil-

ity of expertise, the desire to integrate with other legacy application clients, and the nature of the target client base.

2.1.1 Platform Variability

Every client platform has limitations that constrain an application's design. The processing power of a client device is a major constraint. If the client executes in a limited device such as a cell phone or pager, the server should perform as much heavy computation as possible. Conversely, a powerful client platform enables distributed problem-solving. For example, a financial analysis system could offload complex financial projections to a powerful desktop client.

Applications serving clients across multiple 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 any capabilities unique to a particular platform.

Sometimes, supporting multiple platforms with a single, elegant solution is not even possible, because the differences among platforms are simply too great. For example, the requirements of wireless devices are significantly unique, leading to the emergence of the Wireless Application Protocol (WAP) and the Java Mobile Information Device Profile (MIDP).

2.1.2 Network Service

Not only must enterprises be available to clients on many platforms, but also they must communicate with clients over a wider 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 face intermittent connectivity (and are usually online for short periods of time, anyway).

Regardless of the quality of service available, developers need to remember that the network is imperfect. Three problems of distributed computing are worth mentioning here (paraphrasing Deutsch):

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

A well-designed enterprise application must address these problems. Possible solutions include creating a client that works effectively when disconnected, makes fewer connections to the server, and transmits less data per connection. Later in this chapter, we elaborate on strategies for achieving these goals.

2.1.3 Security and Firewalls

Different networks have different security requirements, which constrain how clients connect to an enterprise. For example, when clients connect over the Internet, they must be able to talk to 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. (Unfortunately, it is also mistaken as a *carte blanche* for circumventing firewalls altogether.)

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 simple as having the user log in only once to obtain access to the entire enterprise. (This scheme is known as *single signon*.) When the client and server are in different security domains, as would be the case over the Internet, the security integration cannot be as tight.

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.

2.1.4 User Type

Last, but not least, developers must consider who is using their application. The general public? A captive audience? Business partners? Employees in the field? The

users generally determine the platform, network, and security requirements. For example, Web browser clients are popular for business-to-consumer applications because they provide a familiar interface. In contrast, mobile clients might be more convenient for sales force automation or customer relationship management applications because they facilitate real-time access to enterprise resources.

2.2 Design Issues and Guidelines

The J2EE platform generally encourages *thin client* configurations, as its component model assigns heavyweight functions such as business operations and data access to the server. This is not to say that J2EE clients are dumb; a J2EE application client may handle a range of responsibilities, such as messaging, presentation, input validation, and session management. How you assign these responsibilities to the client can significantly impact your development efforts, the performance of your application, and the experience of your users.

This section discusses the issues behind designing J2EE clients. Before beginning this discussion, it is necessary to group J2EE clients into the following three classes:

- **Enterprise Information Systems clients.** EIS clients connect to the EIS tier, using an interface such as the Java Database Connectivity (JDBC) API. The programming model for EIS clients usually follows the canonical two-tier or three-tier client-server architecture.
- **EJB clients.** EJB clients using Java technology connect to the EJB tier through the Remote Method Invocation (RMI) API. Under the hood, the transport protocol is Internet Inter-ORB Protocol (IIOP). The RMI mechanism shields Java client developers from the details of IIOP, but developers using non-Java technologies can connect to the EJB tier over IIOP.
- **Web clients.** Web clients connect to the Web tier, using HTTP as the transport protocol. The class of Web clients includes not only browsers, but also Java applets, MIDlets, and applications, as well as other enterprise services.

This book details the programming models for Web clients. (It gives a brief treatment of EJB clients using Java technology, and does not discuss EIS clients at all.) Web-based programming models are generally easier to implement and maintain, and the technologies involved are well-understood.

2.2.1 Messaging

The Web-based programming model revolves around HTTP, and in many circumstances, around HyperText Markup Language (HTML) and Extensible Markup Language (XML) as well. This section discusses these standards and why they are important.

HTTP

The rise in popularity of Web services, and the ease of implementing Web clients in general, can be traced to several characteristics of HTTP (and its secure extension, HTTP over Secure Sockets Layer, or HTTPS):

- **It is widely deployed.** Almost every desktop or laptop computer has a browser that can communicate over HTTP. In the wireless space, HTTP networking occurs on native TCP/IP stacks on some devices, and through gateways on others.
- **It is robust and simple.** HTTP is simple and well understood. Good implementations of HTTP servers and clients are widely available. Furthermore, applets, MIDlets, and rich clients all have access to HTTP networking libraries that are native to their Java runtime environments.
- **It goes through firewalls.** Due to the extensive use of HTTP, firewalls are typically set up to pass HTTP through. This makes it the protocol of choice on the Internet where the server and the client are separated by a firewall.
- **It supports any data format.** You can send any data over HTTP; not only HTML, GIF, or JPEG data, but also XML documents and even arbitrarily-structured text or binary messages.

At the same time, the simplicity of HTTP presents a couple of minor disadvantages that J2EE addresses:

- **It is sessionless.** HTTP is a request-response protocol, with no built-in session support. Therefore, individual requests are treated independently. The J2EE Web tier solves this problem by providing a simple, flexible mechanism for sessions with HTTP.
- **It is non-transactional.** HTTP is a networking protocol, and not designed for general purpose distributed computing. As a result, it has no concept of trans-

action or security contexts. However, this is not a problem in practice because J2EE servers provide transactions and security context on top of HTTP.

HTML and Other Presentational Markup

HTML allows a single document to have a reasonable presentation regardless of the agent that displays it. All an HTML-compliant agent has to do is interpret the document's markup, which draws from the set of tags defined in the W3C HTML specifications. As long as the server sends the agent pages containing valid markup, the pages will appear correctly on on the agent.

By design, HTML defines a limited set of formatting directives and user interactions. You can enhance HTML documents using technologies such as JavaScript in combination with other W3C specifications such as Cascading Style Sheets (CSS) and the Document Object Model (DOM); these enhanced documents are said to be authored in Dynamic HTML (DHTML). However, the support for DHTML is inconsistent across browsers; sometimes, developers must design different DHTML versions of a page.

Other alternatives exist for HTML, particularly for mobile devices, which have more limited presentation capabilities than those of a traditional desktop computer. Examples include Wireless Markup Language (WML), Compact HTML (CHTML), Extensible HTML (XHTML) Basic, and Voice Markup Language (VoiceML).

XML

XML is a standard notation for structured data. It is also presentation-independent, making it attractive to use with clients that have richer interface capabilities that exceed the requirements of HTML. For example, the sample application's StarOffice and Excel clients read XML order data from the server and present the data in their respective interfaces. You could easily use the same XML with a Flash client or any other client with XML-processing capabilities.

XML can be used to encode not only data served to clients, but also messages from clients to J2EE applications. To accommodate a Visual Basic client, for instance, you could set up a servlet to listen for Simple Object Access Protocol (SOAP) messages over HTTP. The servlet would fulfill the requests encoded in these messages by calling the appropriate server-side business objects.

2.2.2 Presentation

Although the client presents the interface, the mechanisms behind generating this interface can be distributed among client and server. In a browser-based scenario, the server generates presentation instructions in a marked-up document and the browser needs only interpret these instructions. With richer front-ends, you program the GUI completely on the client.

With respect to presentation, browser-based interfaces offer the following advantages over richer front-ends:

- **A familiar environment.** Browsers are widely deployed and used, and the interactions they offer are fairly standard. Such ubiquity and predictability makes browsers ideal for novice users.
- **Easier implementation.** The markup languages that browsers use provide high-level abstractions for how data is presented, leaving layout and event-handling to the browser. A richer front-end requires you to handle these details yourself.

On the other hand, richer front-ends offer the following benefits compared to browser-based interfaces:

- **Customizable interactions.** By design, HTML is document-oriented; its tags are restricted to interactions that make sense for hyperlinked documents. Richer front-ends let you create interactions that are more natural for the task at hand.
- **Increased responsiveness and lower server usage.** Every time the browser switches views, it usually needs to retrieve markup for the new view from the server. In contrast, richer clients connect to the server only when they need to retrieve or store business data (and caching or synchronization techniques can reduce these connections even further).

You do not necessarily have to adopt an either/or approach. One of the J2EE platform's strengths is its integrated support for multiple types of clients. A flight booking application that lets users select their seats, for example, might expose two interfaces: a simple HTML interface that lets users choose between aisle or window seats, and a richer applet interface that allows users to pinpoint the exact seat they want. The interface a user chooses depends on the amount of interactivity he or she desires.

2.2.3 Input Validation

Consider a transaction where an application needs to verify the credit-card information a user has given. A J2EE client cannot single-handedly validate this information, but it can certainly apply some simple heuristics to *invalidate* the information. For example, a client could check that the cardholder name is not null, or that the credit-card number has the right number of digits. A client can enforce such rules, and only contact the server when all obvious problems are solved.

We recommend validating user input as much as possible on the client before sending it to the server. By doing so, you reduce the number of network roundtrips, which leads to a couple of benefits:

- **Increased responsiveness.** Because validation occurs on the client, any errors the client detects can be reported instantly. Errors the server detects take longer to report, especially when latency is high.
- **Decreased server load.** Because the client filters out obviously bad inputs, it makes fewer connections to the server, allowing the server to focus on less trivial tasks.

Client-side validation also has some costs:

- **Duplicate logic among clients.** If your application has multiple types of clients, then generally you should duplicate any validation occurring on one client type on the other client types. Otherwise, not all client types will enjoy the benefits of client-side validation.
- **Duplicate logic among clients and server.** The EJB and EIS tiers must enforce consistency regardless of what the client does, because data are meaningless to the extent that they are inconsistent. Client-side validation is an optimization. It improves user experience and decreases load, but you must never rely on the client exclusively to enforce data consistency.

The advantages in performance usually outweigh the costs in duplication. Also, you can minimize the impact of duplication on your development work. For example, auto-generating recurring validation code for browser clients with JSP tags makes maintaining such code slightly easier. (For more information on JSP tags, see [REF: the Web Tier Chapter].)

Types of Validation

You cannot validate all types of data on the client. To understand why, consider these three types of validation:

- **Lexical:** Is a single value well-formed? For example, is the value an integer as expected?
- **Syntactic:** Are composite values well-formed? For example, is the date given as `<month>/<day>/<year>`?
- **Semantic:** Is the value meaningful? For example, 1/29/2001 is semantically valid, but 2/29/2001 is not.

You can easily implement client-side lexical and syntactic validation. Client-side semantic validation, however, is not always possible. You can eyeball a date and determine whether it is legal, but you would need a server to validate a credit card number, even if it had the correct number of digits.

Validation vs. Error Prevention

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 which contain valid months, days, and years might be an improvement; a calendar widget might be even better. The potential to implement such custom GUI components depends on the facilities of the client and the expectations of your users.

2.2.4 Session Management

A *session* is a 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. Session state may include both information visible to the user (shopping cart contents, for example) and invisible application control information (such as user preferences). Web clients require a session state mechanism because HTTP is stateless. This section discusses how to maintain state in the client tier.

Generally, an enterprise application should not maintain session state in the client tier. Instead, we recommend maintaining session state using a stateful session bean (see [REF: EJB Tier chapter]). Applications not using enterprise beans should use the HttpSession interface to store session state in the Web tier

(see [REF: Web Tier chapter]). In either case, the application server provides reliable, scalable, and standard access to session state, saving the developer the trouble of implementing a custom solution.

Additional reasons to avoid client-tier session state include:

- **Higher data transmission.** Session state stored on the client must be sent to the server on each request. You can solve this problem by storing just references to data (session IDs, user IDs, etc.) on the client, and retrieving the data as necessary from another tier (at the cost of increased server complexity and resource usage). For more information, see [REF: Web Tier chapter].
- **Vulnerability to unauthorized access.** Sensitive information (passwords, contact information, credit card numbers, etc.) stored on a client device is vulnerable to abuse by anyone else who can access the device (or its local storage). Encrypting the data stored on the client might solve this problem, as long as the data is not intended for display.
- **Complex implementation.** The mechanisms for maintaining client-side session state all have peculiarities that make them difficult to use reliably and consistently. (These issues are expanded upon later in this chapter.)

In contrast with these disadvantages, client-tier session state enables servers to be stateless, providing the following advantages:

- **Reduced server load.** Stateless servers do not require resources to track session state.
- **Easier load-balancing.** Stateless servers are easier to cluster; any server can service any client because session state does not need to be migrated between cluster nodes.
- **Increased reliability.** When a stateless server goes down, it does not lose any session information since it does not maintain such information. So, reconnecting to the same URL should revive the session. Alternately, a client can be redirected to another server on failover to complete the request.

2.3 Implementation Issues and Guidelines

The following sections describe how to implement presentation, messaging, validation, session management, deployment, and security for browser clients, as well as Java applet, application, and mobile clients.

2.3.1 Browser Clients

Browsers are the simplest type of J2EE client. They present documents, authored in some markup language, which specify both the data to be presented, as well as how the data should be presented.

Browsers are attractive to use as enterprise clients because almost every desktop or laptop computer comes with a Web browser, and deploying a Web browser is usually simple. Furthermore, the browser implementation handles many of the details behind session management and security.

2.3.1.1 Messaging and Presentation

In a browser-based configuration, the messages a server sends to the client already contain presentation information. A J2EE application can serve this information in the form of HTML, and generate it dynamically by Java Server Pages and Java servlet technologies. Java servlets can also be used to process messages sent to the server.

HTML

The following example shows what an HTML form might look like in a J2EE application. The following example includes a form whose inputs are sent via HTTP POST to a servlet `com.acme.AccountProcessor` in the Web tier of the J2EE application.

A J2EE application can deliver the form using a static HTML page, or construct it dynamically and serve it using JSP pages or Java servlets. For more information on using JSP pages and Java servlets, see [REF: Web Tier chapter].

CSS

A CSS stylesheet associates presentation styles with HTML or XML tags in a browser. Each time the browser renders a tagged item, it applies the style corresponding to the tag defined in the stylesheet.

Using stylesheets has a few benefits:

- **Increased consistency.** Because stylesheets work by reference, they encourage a consistent look-and-feel.
- **Increased maintainability.** To change the look-and-feel of an application, you modify one stylesheet instead of multiple pages. Furthermore, the pages are not cluttered with duplicate styling statements so they are more readable to developers.
- **Lower network usage.** Because stylesheets help eliminate duplicate style directives, they reduce the size of each page served to the client.

2.3.1.2 Validation

You can embed JavaScript in an HTML page for validating inputs before sending them to the server. JavaScript contains DOM hooks that permit access to the elements of the form used to input data. Note that JavaScript implementations tend to vary from browser to browser. If you application serves multiple types of browsers, you should use a subset of JavaScript that you know will work across these browsers.

2.3.1.3 Session Management

Generally, you should manage as little session state as possible on the client. This section describes the issues behind using two mechanisms for maintaining client-side session state: cookies or URL rewriting. (For more information on cookies and URL rewriting, see *The J2EE Tutorial*.)

Cookies

A *cookie* is a small chunk of data sent by a server for storage on the client. (See RFC-2109 at `http://www.ietf.org/rfc/rfc2109.txt` for the cookie specification.) Each time the client sends information to a server, it includes in its request the headers for all the cookies it has received (and stored).

Cookies are usually persistent, so for low-security sites, user data that must be stored long-term (such as a user ID, historical information, etc.) can be maintained easily with no server interaction. For small- and medium-sized session data, the entire session data (instead of just the session ID) can be kept in the cookie.

Unfortunately, cookies have several disadvantages:

- **Cookies are optional.** Cookies may not be available for many reasons: the user may have disabled them, the browser may not support them, the browser may be behind a firewall that filters cookies, and so on.
- **Multiple sessions are impossible.** Browser instances share cookies, so users cannot have multiple simultaneous sessions.
- **Cookie implementations vary.** Historically, cookie implementations in both browsers and servers have tended to be buggy or to vary in their conformance to standards. For example, size limitations on cookies differ from browser to browser (although a 4 kilobyte maximum is a safe bet).

URL Rewriting

URL rewriting involves encoding every URL on a served page to include client-side session state. When a user clicks a link with a rewritten URL, the browser performs an HTTP GET to the server, sending the client-side state as a query string.

Unfortunately, URL rewriting has the following disadvantages:

- **Higher server processing.** Everytime time a page is served, the server must rewrite each URL on the page to encode the session information.
- **Higher data transmission.** URL rewriting adds more information to the page, so the server must transmit more data.
- **Vulnerability to client crashes.** If a client workstation crashes, all of the URLs (and therefore all the data for that session) are lost.
- **Incompatibility with static content.** Entirely static pages cannot be used with URL rewriting, since every link must be dynamically written with the session state (for the session to be kept alive).

URL rewriting has a couple of minor advantages:

- **Slightly more portability than cookies.** URL rewriting works just about everywhere, especially when cookies are turned off.
- **Support for multiple sessions.** Session information is local to each browser instance, since it is stored in the URLs of each page displayed.

2.3.1.4 Security

Web browser clients carry out secure communication using HTTPS, which is widely supported across browsers and servers. For more information, see [REF: Security chapter].

2.3.2 Java Applet and Application Clients

Java *applet clients* are user interface components that typically execute in a Web browser, although they can execute in a variety of other applications or devices that support the applet programming model. Because they are implemented using Java technology, applets can use the facilities of the Java runtime environment in which they execute. Although these facilities may vary from browser to browser, applets can expect to have access to a standard set of APIs (particularly for constructing GUIs and networking over HTTP).

Java *application clients* are first-tier client programs that execute in their own Java virtual machines. Application clients follow the model for Java technology-based applications: they are invoked at their main method and run until the virtual machine is terminated. Also, application clients are less dependent on the network and the J2EE application server; because they can work when disconnected, they are sometimes termed as Java *standalone clients*.

2.3.2.1 Presentation

For Java applets and application clients, the Java Foundation Classes (JFC)/Swing API provides a comprehensive set of GUI components (tables, trees, and so on) that can be used to provide a more interactive experience than the typical HTML page. Additionally, the JFC/Swing HTML text component can act as an embedded browser, displaying HTML received from the server.

For more information on JFC/Swing, see the Java Tutorial at `http://java.sun.com/docs/books/tutorial/uiswing/index.html`.

2.3.2.2 Messaging

When applet and application clients connect to the EJB tier, they use regular Java RMI calls; transparent to the developer, the RMI mechanism dispatches calls to the correct objects and also marshalls and unmarshalls data sent from point to point.

The programming model for Web clients is slightly more complex, because additional code is required to interpret the messages sent by the client, and the

responses sent back by the server. Applet and application clients send messages to the J2EE server in the form of HTTP POST requests with an XML payload, and receive XML in the HTTP response.

2.3.2.3 Session Management

When applets or applications communicate with servlets or JSP pages over HTTP they need a way to keep a session alive. For this purpose, cookies and URL rewriting work adequately.

When a client uses cookies, the server sets the cookie at the beginning of the session and the client sends it back with each request. When a client uses URL rewriting, the server gives the client a session ID, which the client appends to each request.

2.3.2.4 Deployment

Java applications and applets as Web clients are generally straightforward to deploy. Application clients are deployed using Java Web Start, while applet clients are deployed as components embedded in a Web page. Applets and applications as EJB clients, on the other hand, have additional deployment requirements.

Deploying Application Clients as Web Clients

To deploy application clients as Web clients, use Java Web Start. Preparing an application client for Java Web Start deployment involves packaging the application's classes and libraries into a JAR, and distributing the JAR with an accompanying Java Network Launching Protocol (JNLP) file. When a Java Web Start-enabled browser reads this file, it automatically launches the application with the help of the information in the JNLP file.

For more information on Java Web Start and JNLP, see `http://java.sun.com/products/javawebstart/developers.html`.

Deploying Applet Clients

Applet clients are delivered through applet tags embedded in HTML. The Web browser downloads the code for the applet (packaged inside a JAR file) at request time and executes it in a Java virtual machine on the client platform.

The J2EE server replaces the `<jsp:plugin>` with either an `<object>` or `<embed>` tag, as appropriate for the requesting client. The attributes of the

<jsp:plugin> tag provide configuration data for the presentation of the element as well as what plugin is required, and where it can be obtained.

For more information on deploying applet clients, see the J2EE Tutorial.

Deploying Java EJB Clients

If your applet or application is an EJB client, you must distribute it with a client-side container; the application server vendor provides this container, usually in the form of a library.

The client-side container manages the details of RMI-IIOP communication. It also allows access to a variety of middle-tier services, such as Java Naming and Directory Interface (JNDI) for directory lookups, Java Message Service (JMS) for messaging, and the Java Database Connectivity (JDBC) API for relational database access.

The JAR files for EJB clients include a deployment descriptor similar to other J2EE application components. The deployment descriptor indicates what components and resources the client accesses.

The J2EE platform does not completely specify a mechanism for deploying EJB clients, so it varies from vendor to vendor. For more information, consult your application server's documentation.

2.3.2.5 Security

Like browser clients, Java applet and applications that connect as Web clients use HTTPS for secure communication. For more information, see [REF: Security chapter].

On the client itself, the Java programming model protects users from local security violations. The Java runtime environment puts all code into a *sandbox* by default; the environment considers the sandbox code untrusted and imposes strict limitations on what the code can do. Administrators can ease these limitations by modifying the security policies of the Java runtime environment on the client. Alternatively, a program may request additional permissions from the user at runtime, presenting its credentials in a digitally-signed certificate.

2.3.3 MIDlet Clients

Java *MIDlets* 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 Runtime Environment for cellular phones, two-way pagers, and palmtops.

2.3.3.1 Presentation

MIDP's GUI package `javax.microedition.lcdui` is screen-oriented, and geared towards the limited input capabilities of mobile information devices. The MIDP GUI package contains simpler components than the JFC/Swing GUI package. Consequently, the interactions with these components must be simpler.

2.3.3.2 Messaging

HTTP provides the only bridge between the J2ME MID Profile and the J2EE platform. (At the moment, RMI is not a native component of MIDP, so MIDlets cannot connect to the EJB tier.)

MIDP includes standard support for HTTP 1.1, and includes APIs for generating GET, POST and HEAD requests, basic header manipulation, and stream-basedconsumption and generation of messages. Consequently, you can use the JSP and Java servlet technologies to serve MIDP clients.

Note that while you program against an HTTP networking API, what goes on behind the scenes may or may not occur over TCP/IP. Depending on the carrier-network, messages moving between a MIDP device and a J2EE server may tunnel through a number of different protocols. On the server side, the deployment configuration of the carrier network ensures that such network connections are routed to and received by the J2EE server in the form of HTTP messages.

There are no requirements about the format of messages that flow between a MIDP client and a J2EE application. Recall that both JSP pages and Java servlets let you send any type of data in the body of an HTTP response, not only HTML, or markup for that matter. Similarly, a MIDlet can send any type of data in the body of an HTTP request. So, the format could be something simple like a string of comma-separated values, or something more structured in the form of key-value pairs, or more formally, XML data conforming to a set of DTDs or schemas.

A few well-supported XML parsers exist for MIDP. (At the moment, these are not native components of the profile.) These parsers do come at a cost, however, increasing a MIDlet's size by 15 to 30 kilobytes. Also note that since the parsers are designed for constrained environments, they may offer only SAX for processing XML documents, and may not include XML validation or sophisticated error-reporting mechanisms.

2.3.3.3 Session Management

Like JFC/Swing clients, MIDP devices can use either cookies or URL rewriting to maintain sessions. When using cookies, clients must always retransmit the last cookie sent by the server, which usually contains only a session ID. When using URL rewriting, the server encodes the session ID in the URL and the client reuses this URL for each request.

It can be useful to use URL rewriting as a fallback to cookie support since some wireless gateways do not allow the transmission of cookies.

2.3.3.4 Deployment

A MIDP application is packaged inside a JAR file, which contains the application's class and resource files. You can deploy this JAR file using a couple of methods:

- Pre-installing the JAR on the mobile device.
- Downloading the JAR from the J2EE server once onto the mobile device.

When deploying the application, keep in mind that wireless devices generally have limited memory and work on networks with limited bandwidth. Smaller applications are desirable.

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, see the [J2ME Wireless Toolkit User's Guide].

2.3.3.5 Security

Some, but not all, MIDP-compliant devices support SSL, which can be used for secure communication over HTTPS. Furthermore, because of MIDP's Generic Connection Framework, making an HTTPS connection does not require any special or different code. A MIDP client just needs to open a connection to a server on an URL beginning with `https:`.

Like Java applet and application Web clients, a MIDP client relies on a digital certificate mechanism for server authentication. For more information, see [J2ME Wireless Toolkit User's Guide].

2.4 MVC in the Client Tier

[TBD]