Designing Wireless Clients for Enterprise Applications with Java Technology

A Java BluePrints for Wireless White Paper

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

June 26, 2003 (draft)

Please
Recycle

Designing Wireless Clients for Enterprise Applications with Java Technology

*Please send comments on this **draft** paper to javablueprintswireless-feedback@sun.com.*
For the latest version of this paper, please visit the Java BluePrints for Wireless Web site at http://java.sun.com/blueprints/wireless/.

This paper describes how to use Java technology to design enterprise applications for wireless clients. It begins with a short overview of enterprise and wireless Java technologies and introduces using these technologies together. The majority of the paper describes how to design a successful wireless-enabled enterprise application. It illustrates its design guidelines using the Java Smart Ticket sample application, which is an example mobile commerce application for movie ticketing. This application is available through the Java BluePrints Web site at `http://java.sun.com/blueprints/`. The principles behind the design of the sample application can be applied to other Java technology-based mobile services.

This paper refers to many parts of Java technology. For introductory and supplementary information, refer to the Resources section at the end of this paper.

1 Overview of Enterprise and Wireless Java Technologies

From big servers to small devices, the Java platform is everywhere. Enterprise applications for wireless clients rely heavily on two parts of the Java platform in particu-

lar: the Java 2 Platform, Enterprise Edition (J2EE), and the Java 2 Platform, Micro Edition (J2ME).

1.1 Java 2 Platform, Enterprise Edition (J2EE)

Widely supported across the industry, the Java 2 Platform, Enterprise Edition (J2EE) standard gives developers a rich technology stack on which to run their enterprise applications. This stack provides facilities for:

- Integrating databases and information systems using the JDBC API and the Java Connector Architecture
- Interoperating with other enterprise applications using the Java Web Services Developer Pack and the Java Interface Definition Language (IDL)
- Reliable messaging using the Java Message Service (JMS) API
- Accessing e-mail services using the JavaMail API
- Accessing naming and directory services using the Java Naming and Directory Interface (JNDI) API
- Modularizing reusable business logic using the Enterprise JavaBeans (EJB) component architecture

When writing enterprise services that leverage the J2EE technology stack, developers may choose how mobile device users interact with their services. There is the Web or browser paradigm, where developers use a markup language to develop user interfaces, as dynamic content, for a service. A browser client interprets these user interfaces. For some applications, extending this traditional Web paradigm to mobile devices may be sufficient. Indeed, the same strengths of JSP and Java servlet technologies, tried and tested on the traditional Web for generation of dynamic HTML pages, apply equally to dynamic content generation using formats such as Wireless Markup Language (WML) and Compact HTML (CHTML) for consumption by browsers on mobile devices.

However, the advent of J2ME technology provides a rich and standard environment with which to enhance the user's experience of a wireless service. This environment is described in the following section.

1.2 Java 2 Platform, Micro Edition (J2ME)

The Java 2 Platform, Micro Edition (J2ME) enables developers to create Java applications for consumer and embedded products. These products include mobile devices, which can be programmed using the Mobile Information Device Profile (MIDP), a set of J2ME APIs. MIDP, together with the Connected Limited Device Configuration (CLDC), provides a complete Java runtime environment for cell phones, palmtops, and two-way pagers, and includes libraries for:

- Programming user interfaces on a device using the Limited Connected Device User Interface (LCDUI) API
- Storing data persistently on a device using the Record Management Store (RMS) API
- Networking with a server or another device through the Generic Connection Framework (GCF)

These libraries enable developers creating networked wireless services to reap the following benefits:

- Developers can create highly interactive and flexible user interfaces. The LCDUI API provides high-level components (lists, forms), which offer standard interactions, and lower-level components (canvas) which allow for a more customized look-and-feel interaction.
- Developers can produce MIDP applications that work even when the device is disconnected from the network. These applications may embed UI logic on the client together with the data that drives it, perhaps using the RMS API to store the data persistently.
- Developers can produce networked application clients that connect to wireless services using standard networking protocols. MIDP requires all devices to support HTTP, the same protocol that Web browsers use.

The last point is particularly important, because it means developers need only understand HTTP and the APIs for HTTP networking, even though the operations behind the scenes may be more complex. Depending on the carrier network used, an HTTP request from a MIDP device may tunnel through various protocols, which may be based on TCP/IP or not. Ultimately, the carrier network must ensure that its intended recipient receives the HTTP request intact. Similarly, the

carrier must ensure that HTTP responses generated by the recipient are properly sent to the MIDP device. (See Figure 1.)

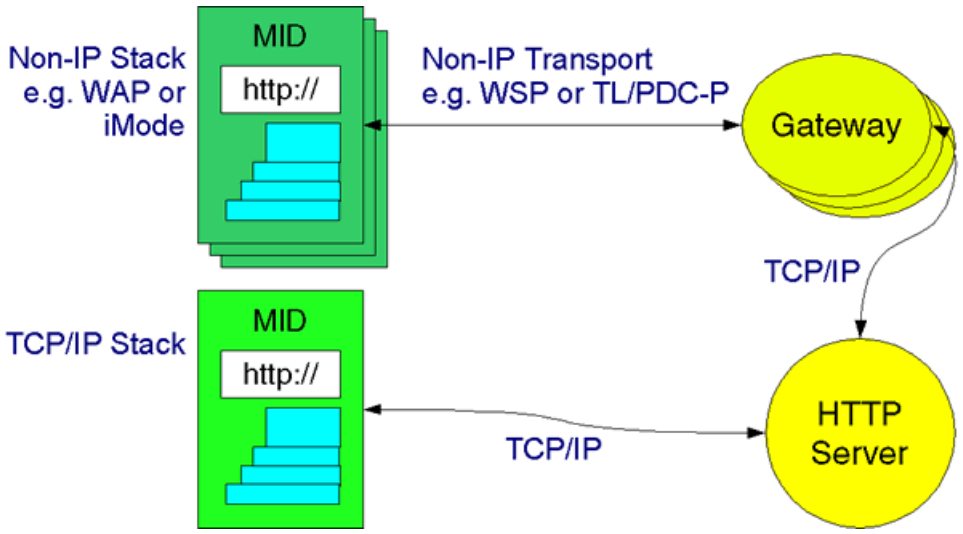

Figure 1 HTTP Network Connections from a MIDP device

Furthermore, HTTP networking bridges the gap between a J2ME mobile device and a J2EE application server. Because they both support HTTP and can easily use this protocol to communicate with each other, developers can implement wireless enterprise applications with Java technology from end to end. The next section examines the anatomy of such applications.

2 The Basic Programming Model

Figure 2 shows the structure of a typical Java wireless enterprise application spanning a J2ME device and a J2EE application server.

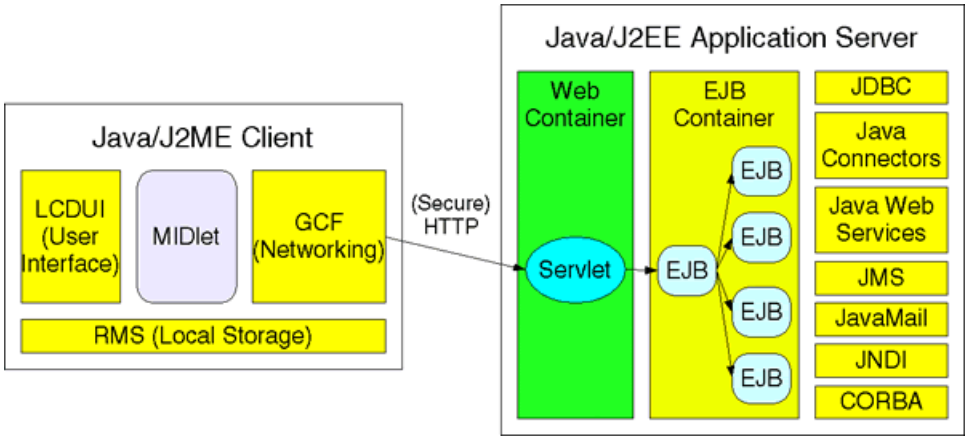

Figure 2 High-Level Architecture of a Java Wireless Enterprise Application

The architecture of an enterprise application serving wireless clients is similar to that of a canonical J2EE application:

- An application client implemented using MIDP, referred to as a *MIDlet*, provides the user interface on the mobile device. This MIDlet communicates with a Java servlet, usually via HTTP, and over a secure channel when necessary.
- The servlet interprets requests from the MIDlet, and in turn, dispatches client requests to EJB components. When the requests are fulfilled, the servlet generates a response for the MIDlet.
- The EJB components, or *enterprise beans*, encapsulate the application's business logic. An EJB container provides standard services such as transactions, security, and resource management, so that developers can concentrate on implementing business logic.
- The servlet and EJB components may use additional APIs to access enterprise information and services. For example, they may use the JDBC API to access a relational database, or the JavaMail API to send an e-mail to a user.

2.1 Supporting Multiple Types of Clients

Developers should note that the J2EE platform emphasizes reusable components through the use of enterprise beans. Applications may leverage these components to

support multiple types of clients with little, if any, impact on the core business logic of the application. Figure 3 shows the architecture of an application with a J2ME client and a non-Java browser client. Although the J2ME and browser clients may access different servlet and JSP components in the Web container, they often indirectly use the same EJB components for business logic and access to persistent data.

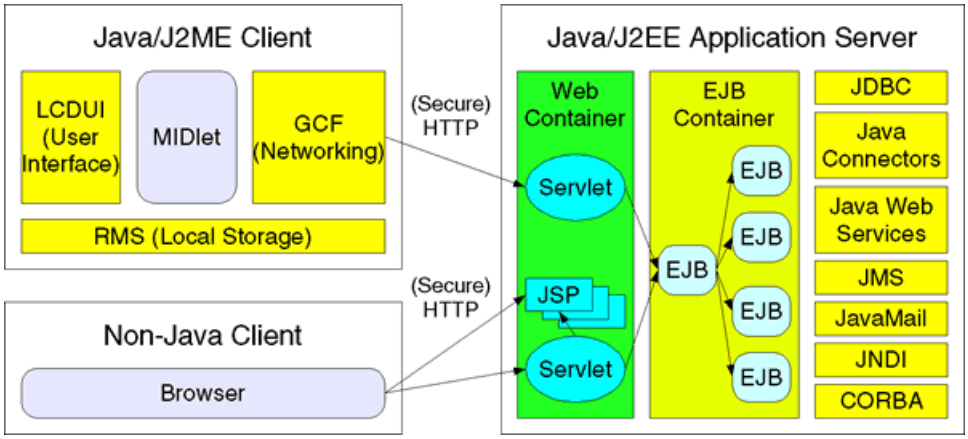

Figure 3 High-Level Architecture of a J2EE Application Supporting a J2ME Client and a Browser Client

For more information on architecting the lower tiers of a J2EE application to support multiple types of clients, see *Designing Enterprise Applications with the J2EE Platform*.

2.2 Java Smart Ticket Sample Application

The Java Smart Ticket sample application shows how to design and implement a wireless enterprise application with Java technology. In this sample movie ticketing application, customers use a mobile device to perform the following functions:

- Create an account through which they sign on to the application
- Browse personalized listings of theaters, movies, and show times
- Reserve seats and purchase movie tickets

Figure 4 shows the basic flow of the application when it is first used, from account creation, to movie selection, and then to seat selection:

Figure 4 Flow of the Java Smart Ticket Sample Application

Figure 5 illustrates the high-level architecture of the application. A MIDlet provides the user interface on the mobile device. The MIDlet communicates with a Java servlet via secure HTTP. The servlet dispatches client requests to various enterprise beans handling account management, movie listings, and ticket purchases. These beans, in turn, access a database via *container-managed persistence* (CMP). (The beans could use alternate data access methods, such as the JDBC API.)

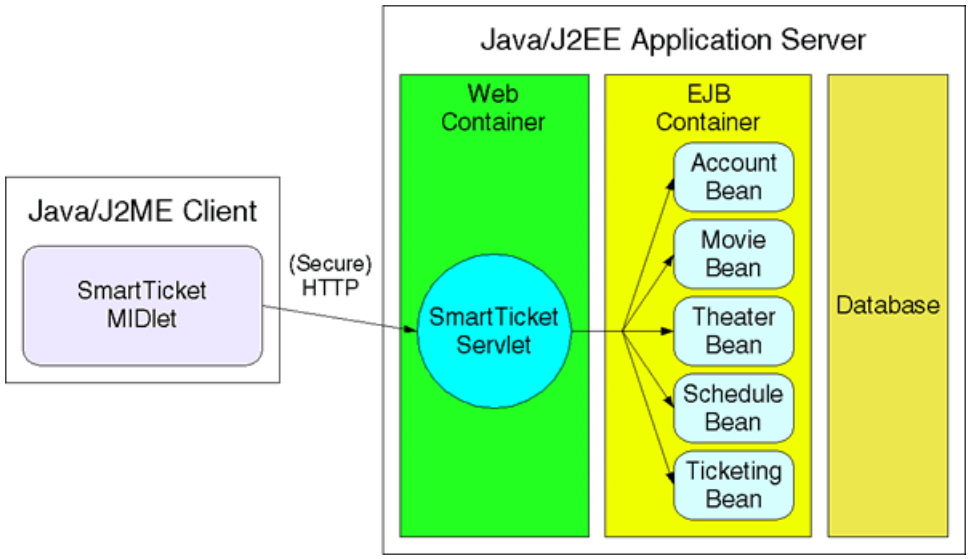

Figure 5 High-level Architecture of the Java Smart Ticket Sample Application

3 Design and Implementation Guidelines

Three aspects to networked wireless applications—client-side architecture, messaging, and presentation—are key areas to understand when designing and implementing such applications. It is important, however, to first understand the constraints wireless application developers face.

3.1 Design Constraints

Mobile device limitations impose many constraints on the design of wireless applications. Applications must offer useful and usable interfaces even though the devices on which they run have limited screen size, input capabilities, processing power, memory, persistent storage, and battery life.

Wireless enterprise applications are especially constrained because of their dependence on the network. The constraints imposed by a mobile network are significantly larger than those of a typical Web browser connected to the Internet. In general, mobile devices can expect to encounter:

- high latency
- limited bandwidth
- intermittent connectivity

A MIDP client can address these constraints by adopting the following objectives:

- Connect to the network only when needed
- Consume only as much data from the network as needed
- Remain useful when disconnected

The guidelines in the following sections should help applications meet these objectives.

3.2 Client-Side Architecture

To take advantage of MIDP's capabilities, the client portion of the application may use the traditional Model-View-Controller (MVC) architectural pattern.

- The *model* contains the data for the application. In the movie ticketing appli-

cation, for example, the model includes collections of movies, theaters, and show times, as well as seating plans for each show.

- The *view* contains the code for presenting data to and gathering data from the user. In the movie ticketing application, the view includes the movie list screen and the interactive seating plan canvas.
- The *controller* governs the logical flow of the user interface. It also controls how user interactions affect the data model and vice-versa. For example, the movie ticketing application's controller asks users to create an account the first time they use the application and keeps a local, persistent copy of the user's login; on subsequent uses, the controller proceeds directly to the main menu, knowing that the user has already created an account.

Although adopting a formal MVC architecture on the client may increase the size of the code, much of it in the controller since it implements application flow, using a MVC architecture offers benefits that may outweigh the costs.

- Application flow is kept isolated from data and presentation—By separating the controller from the view and the model, the flow of the application is isolated in one place. Consequently, developers can look at just the controller to understand the client's perspective. They can also change the flow of the application by modifying the controller, with little, if any, disruption to the rest of the code.
- Data is insulated from presentation—Separating the model from the view insulates the model from changes to the view. In other words, developers can alter the look of the client without having to change the model.
- Presentation is not contingent on how the data is stored—Separating the view from the model insulates the view from the details of how the model works. The view is not affected by how the model manages data: the model may get its data from local storage using the RMS API, from a server using HTTP networking, or from an in-memory cache, or a combination of these sources.

The last benefit is probably the strongest reason to use MVC for mobile clients. To use the network sparingly and remain useful when disconnected, a client must decide when to fetch data from the server or from local storage. Local data strategies may be based on caching or synchronization to improve responsiveness and maintain data coherence. Partitioning these details into a model makes imple-

mentation, testing, and maintenance easier than if those details were spread out across the application.

3.3 Messaging

Although MIDP does not include sophisticated client/server communication mechanisms, such as Java Remote Method Invocation (RMI) or the Java API for XML-based Remote Procedure Calls (JAX-RPC), developers may design a message protocol using a format and transport of their choosing.

For most applications, HTTP is adequate as the basis of a messaging protocol, and HTTP is preferable to other methods of communication (such as those based on sockets or datagrams, for example) for the following reasons:

- All MIDP devices must support HTTP networking. Therefore, applications that rely on HTTP networking are portable across devices. On the other hand, not all MIDP devices support socket-based or datagram-based communication, so applications using those methods are not guaranteed to be portable.
- HTTP is firewall friendly. Most servers are separated from mobile clients by firewalls, and HTTP is one of the few protocols most firewalls allow through.
- Java networking APIs make HTTP programming easier. MIDP includes standard support for HTTP 1.1 and APIs for generating GET, POST, and HEAD requests, basic header manipulation, and stream-based consumption and generation of messages. The Java servlet API, meanwhile, provides an extensive framework for handling HTTP requests and generating HTTP responses.

Figure 6 illustrates a basic HTTP-based messaging scheme between a MIDP client and a Java servlet.

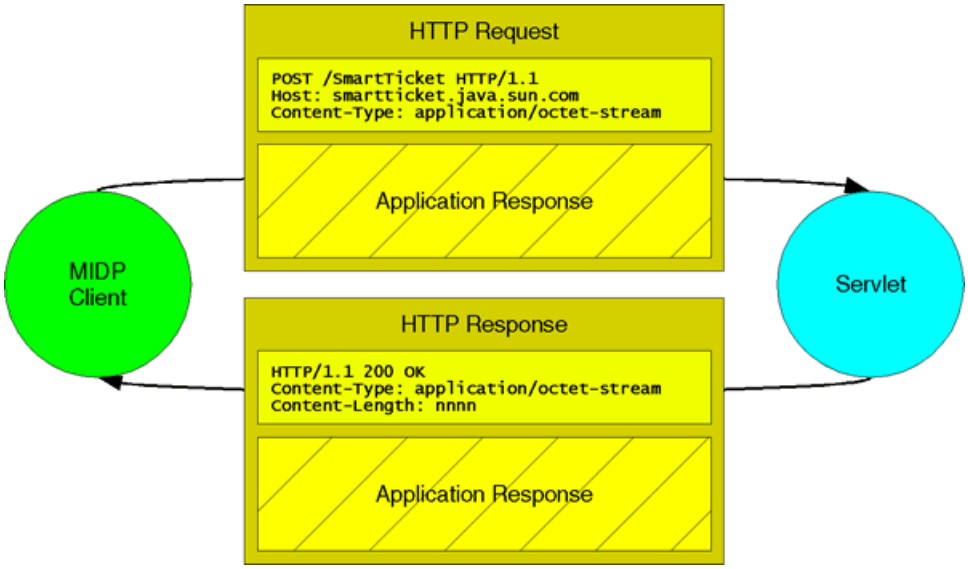

Figure 6 HTTP-Based Messaging Scheme

When a MIDP client communicates with a Java servlet the following events take place:

- The client encodes an *application request* and packages it in an HTTP request, setting the `Content-Type` header to ensure that intervening gateways process the request correctly. For example, `text/plain` may be appropriate as a content type for text requests, while `application/octet-stream` may be used to designate binary requests.
- The servlet receives the HTTP request and decodes the application request. The servlet or some delegate (such as an enterprise bean) performs the work specified by the application request.
- The servlet encodes an *application response* and packages it in an HTTP response, setting the `Content-Type` and `Content-Length` headers to ensure that intervening gateways process the response correctly. For example, `text/plain` may be appropriate as a content type for text responses; `image/png` may be appropriate for PNG image responses; and `application/octet-stream` may be used to designate binary responses in general.
- The client receives the HTTP response and decodes the application response it

contains. The client may instantiate one or more objects and perform some work on these local objects.

The rest of this section focuses on guidelines for HTTP-based messaging. It covers message formats, security, session management, transaction management, and error handling.

3.3.1 Designing a Message Format

The developer decides how to format application requests and responses. The available formatting choices fall between the two extremes of a simple binary message format versus a complex XML message format, as shown in Figure 7.

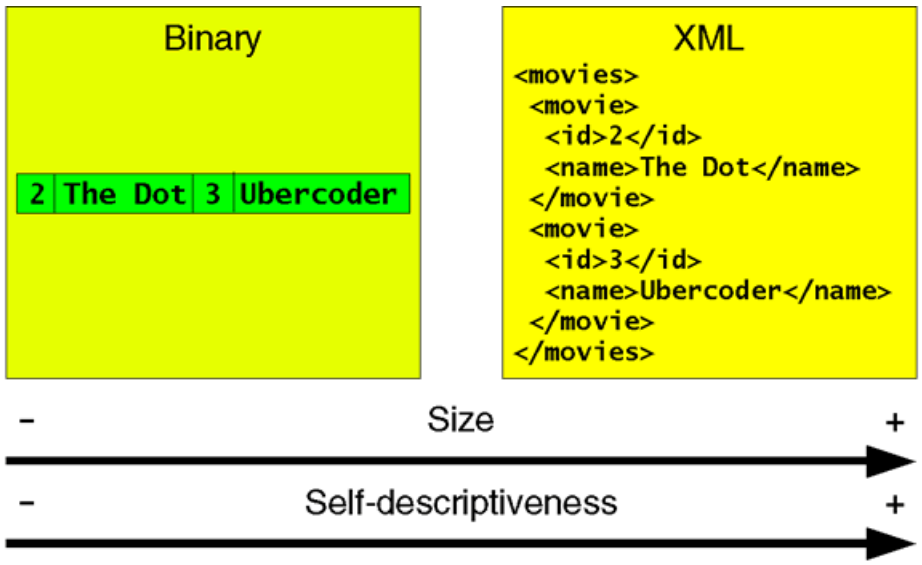

Figure 7 Spectrum of Message Formats

Binary messages, the simplest of message formats, may be read and written using the `DataInputStream` and `DataOutputStream` classes in the `java.io` package. Developers may use different pairs of methods for reading message data, such as `DataInputStream.readInt` and `DataOutputStream.writeInt` for reading and writing integers, `DataInputStream.readUTF` and `DataOutputStream.writeUTF` for reading and writing UTF-8-encoded strings, and so on.

Using binary messages results in efficient exchanges because the message payloads are compact. However, because they are also intended to save space, binary messages sacrifice self-descriptiveness. Consequently, both client and server must know the format of binary messages in advance, which means that client and server are tightly coupled.

Extensible Markup Language (XML) messages are at the complex end of the range of message formats. Whereas the J2EE platform defines many uses for XML (particularly for Web services), the MIDP specification does not require XML support. However, developers are free to add XML support to a MIDP application by incorporating additional libraries.

For XML parsing and general processing, developers may choose from a number of implementations, including those based on two popular processing models, the Document Object Model (DOM) and the Simple API for XML (SAX). (Keep in mind that, given the memory and processing limitations of mobile devices, SAX and similar event-based parsers are considered more appropriate than DOM.) XML-based RPC libraries are also available, including those based on the Simple Access Object Protocol (SOAP) specification.

Developers should take care when adding XML support to an application. In addition to the costs in size and bandwidth, there are non-trivial costs in memory, processing, and storage. Developers are advised to measure these costs when deciding to support XML.

Developers accustomed to targeting browser clients, which consume documents marked up with presentation directives in the form of Hypertext Markup Language (HTML) or Wireless Markup Language (WML), may be unfamiliar with designing a message format or uncomfortable with the freedom to do such design. Java mobile devices, in contrast to browser clients, implement presentation on the device. Because they do not depend on the server for presentation, Java devices need only retrieve data from the server.

Finally, developers may wish to consider two implementation strategies after choosing a message format.

First, to ensure the compatibility of the serialization and deserialization of request parameters and returned values exchanged between the client and the server, developers may find it helpful if classes support their own serialization and deserialization and if these classes are shared between the client and the server. Of course, special care must be taken to ensure that these shared classes only use APIs compatible with both the J2ME MIDP and J2SE/J2EE APIs.

Second, if objects exchanged in messages are to be saved persistently on the client, then developers may find it helpful to reuse for data persistence the serialization and deserialization methods used for messaging.

The movie ticketing application uses a binary data format. Model objects know how to serialize and deserialize themselves, and this serialization/deserialization mechanism is used for both client-server messaging and client-side data persistence. (See Code Example 0.1, Code Example 0.2, and Code Example 0.3.)

```
package com.sun.j2me.blueprints.smartticket.shared.midp.model;

public class Movie {
    // ...

    public void serialize(DataOutputStream dataStream)
            throws ApplicationException {
        try {
            dataStream.writeUTF(primaryKey);
            dataStream.writeUTF(title);
            dataStream.writeUTF(summary);
            // ...
        } catch (IOException ioe) {
            throw new ApplicationException(ioe);
        }
    }

    public static Movie deserialize(DataInputStream dataStream)
            throws ApplicationException {
        try {
            Movie movie = new Movie();
            movie.primaryKey = dataStream.readUTF();
            movie.title = dataStream.readUTF();
            movie.summary = dataStream.readUTF();
            // ...
        } catch (IOException ioe) {
            throw new ApplicationException(ioe);
        }
```

```
    }
}
```

Code Example 0.1 Serialization and Deserialization of a Movie

```
public class HTTPCommunicationHandler /* ... */ {
    public Movie getMovie(String movieKey)
            throws ModelException, ApplicationException {
        HttpConnection connection = null;
        DataOutputStream outputStream = null;
        DataInputStream inputStream = null;

        try {
            connection = openConnection();
            outputStream = openConnectionOutputStream(connection);
            outputStream.writeByte
                  (MessageConstants.OPERATION_GET_MOVIE);
            outputStream.writeUTF(movieKey);
            outputStream.close();
            inputStream = openConnectionInputStream(connection);
            Movie movie = Movie.deserialize(inputStream);
            return movie;
        }
        // ...
    }
}
```

Code Example 0.2 Movie Deserialization Used in Messaging

```
public class RMSAdapter {
    // ...

    public int storeMovie(Movie movie,
                          int recordId) throws ApplicationException {
        try {
            ByteArrayOutputStream stream
                = new ByteArrayOutputStream();
            DataOutputStream dataStream
```

```
                    = new DataOutputStream(stream);
                movie.serialize(dataStream);
                dataStream.flush();
                if (recordId > 0) {
                    remoteDataRecordStore.setRecord(
                        recordId, stream.toByteArray(), 0,
                            stream.size());
                } else {
                    recordId =
                        remoteDataRecordStore.addRecord(
                            stream.toByteArray(), 0, stream.size());
                }
                return recordId;
            }
            // ...
        }
    }
```

Code Example 0.3 Movie Deserialization Used in Data Persistence

3.3.2 Communicating Securely

MIDP clients may rely on some of the same mechanisms used to support secure communication between J2EE applications and Web browser clients. J2EE application servers and many MIDP devices support HTTP over Secure Sockets Layer (SSL). MIDP devices use secure HTTP to authenticate a server and carry out confidential conversations with that server. The Generic Connection Framework in MIDP allows developers to open secure HTTP connections simply by calling `Connector.open` with an URL that begins with `https`. (Note that `Connector.open` throws a `ConnectionNotFoundException` if HTTP over SSL is not available, so developers may catch this exception and handle the absence of secure HTTP appropriately.)

For client authentication, MIDP clients rely on application-managed sign on, possibly based on self-registration. In other words, a MIDP client sends its credentials (such as a login and password combination) to a J2EE application, and the application verifies these credentials, perhaps using a database. This technique is used in the movie ticketing application.

3.3.3 Managing Session State

Clients frequently make multiple related requests to a server in the context of a single *session*. During a session, the client may want the application to track *session state*. Session state, or conversational state, is information related to a particular user's interaction with an application and, while it may span multiple user requests, usually lasts for just that single session. An example of session state is the contents of a shopping cart kept updated as a user browses and requests items from a catalog.

However, because HTTP is a stateless protocol, individual client requests are treated independently, presenting an obstacle to an application tracking a client's session state. To work around this limitation, HTTP clients—and by extension, MIDP clients—may group related requests into a session using either cookies or URL rewriting.

- Cookies—A cookie is a small chunk of data that a server sends for storage on the client. Each time it sends information to a server, the client includes in its request the headers for all the cookies it has received (and stored).
- URL rewriting—This technique involves encoding every URL on a served page to include client-side session state. When the client accesses an URL, the encoded session state is sent back to the server as part of the accessed URL.

URL rewriting is preferable to using cookies; many wireless network gateways filter cookies, so using cookies is not always reliable.

When using URL rewriting or cookies, clients should restrict themselves to embedding a session ID in rewritten URLs or the cookies to keep programming simple and bandwidth costs to a minimum. This session ID is the key to session information on the server. Regardless of which method—cookies or URL rewriting—is used, a J2EE Web container associates a state cache (HTTPSession) with the session, which it identifies with a session ID, and manages it for the lifetime of the session. This cache may be used to hold references to other components, such as enterprise beans, database connections, and so forth.

To illustrate, in the Java Smart Ticket sample application, the server tracks reservations on behalf of the client during a session. When it makes a reservation, the client in its request specifies the particular seats to reserve. When it makes a subsequent request to confirm or cancel the reservation, the client does not have to again specify the particular seats because the server has cached this information and associated it with the client's session ID.

3.3.4 Managing Transactions

The J2EE platform supports transactions in several ways. Developers may manage transactions manually using the Java Transaction API, or rely on a J2EE application server to manage those transactions automatically. Enterprise beans typically perform transactions, but business components in the Web tier may perform transactions as well.

When designing a MIDP client, developers should be aware that transactions cannot span HTTP requests. That is, each HTTP request has its own transaction context. (Note that browser clients are also affected by this restriction.) If a request specifies operations that require a transaction, the operations are treated as an atomic unit; before the response is returned, either all of the operations have been performed, or none of them have been performed.

Consequently, if a MIDP client wants to undo the effect of a request, it technically cannot issue a rollback in a subsequent request, because the subsequent request would have a separate transaction context. Instead, the application must use a *compensating transaction* to undo the request. Such a transaction requires developers to consider additional issues beyond the scope of this document. (For more information on transactions, see *Designing Enterprise Applications with the J2EE Platform*.)

Write operations to the local database that depend on the success of a remote operation should occur only after the success is confirmed. For example, in the sample application, when a user creates an account, the account information is saved on the server and some of the information is also saved on the client, such as the login and password. The client saves account information persistently only after asking the server to create an account and receiving acknowledgement that the account was successfully created. If the client saved the information persistently before contacting the server, and the operation failed, then the information the client saved persistently would be useless.

3.3.5 Error Handling

When a J2EE application server cannot carry out a request on behalf of a MIDP client, the server needs to report this to the client. While it may use the Java exception-handling mechanism to deal with errors internally, the server code cannot use this mechanism to report errors to a MIDP client, because the MIDP client communicates over the network. In other words, a programmer cannot set up a try-catch block in the client code to directly catch exceptions thrown by the server. Instead,

developers must incorporate an error reporting mechanism into their messaging protocol.

One strategy is to devote a fixed portion of each application response to a status code that indicates whether an application request was successful. For example, if using binary messaging, the first two bytes of a message can be devoted to an integer status code. When using HTTP, application developers may also use the HTTP response status code to indicate success or failure at the communication level. For example, a status code of 200 (`OK`) may indicate success, whereas a status code of 500 (`Internal Server Error`) may indicate failure. However, it suffices to embed error reporting in the messages themselves.

3.4 Presentation

The more focused and directed a user's interaction is with an application, the better the user's experience. This is particularly crucial for wireless applications given the limited screen and input capabilities of mobile devices. Furthermore, because they are networked, wireless enterprise applications are susceptible to latency delays, which can create serious usability problems if unaddressed.

Developers may employ several strategies to make networked wireless applications more usable: implement client-side validation, use threads for long operations, provide progress indicators, make operations interruptible, and personalize the application. This section discusses those strategies. For general usability considerations, see the *MIDP Style Guide*.

3.4.1 Implement Client-Side Validation

Validating inputs on the client side is a well-documented method for reducing the costs of calls to the server. Although not all information can be validated on the client, it is possible to minimize the information that must be validated on the server. For example, consider a form for placing an order, where the form includes fields for credit card information. A MIDlet can not validate this information entirely on its own, but it can certainly apply some simple heuristics to determine whether the information is invalid. For example, the MIDlet can check that the cardholder name is not null or that the credit card number has the right number of digits. If the inputs pass these heuristics, the client can pass them on to the server. The server can perform more complex validation such as checking that the credit card number really belongs to the given cardholder or that the cardholder has sufficient credit for the purchase.

Note that the reason to perform client-side input checking is to avoid unnecessary round trips to the server. Client-side input checking should not obviate the need for equivalent checking on the server side. The server should still check for invalid inputs, regardless of what the client does.

MIDlets can be even more proactive by *preventing* invalid inputs. In the Java Smart Ticket sample application, for example, the ZIP code text field is created using a numeric constraint. As a result, a non-numeric ZIP code entry cannot be sent to the server (and subsequently be rejected).

3.4.2 Thread Long Operations

Because networked operations and some operations on local storage may take a long time, developers should run these types of operations in a thread.

Often, such operations are invoked as a result of the user selecting a soft command, which results in a callback to `CommandListener.commandAction`. According to the MIDP specification, this method should return immediately; if it doesn't, the application risks deadlock. So, any `commandAction` implementation that starts a networking or local storage operation should thread that operation so that the callback can return quickly.

Also, threading a long operation allows the client to concurrently throw up a progress indicator, periodically update this indicator, and listen for user requests to cancel the operation, as the next two sections describe.

3.4.3 Provide Progress Indicators

Because networked operations may take a long time, applications should give users feedback on an operation's progress. The Java Smart Ticket sample application, for example, uses a gauge to implement a progress indicator. (See Figure 8.) Generally, use a progress indicator for longer operations, such as downloading a seating plan over the network.

Figure 8 A Progress Indicator

3.4.4 Make Operations Interruptible

Giving users the option to interrupt long operations keeps them in control of the application. In the Java Smart Ticket sample application, progress indicators may optionally incorporate a stop button. The indicator listens for stop button events, and the display immediately switches to the previous screen when the stop button is pressed.

Note that not all operations should be interruptible, especially operations that have multiple dependencies. For example, it may be preferable not to interrupt an operation that creates a user account on the server and also saves a part of the account on the client. These two tasks ideally should occur as a unit or not at all. If the operation were interrupted, inconsistency might be introduced between the client and server data.

3.4.5 Personalize the Application

The concept of *personalization* refers to the ability of a service to adapt to information that it knows about a user. Typically, many aspects of a user, such as his or her address, ZIP code, or favorite color, do not change from session to session. Because such data is stable, an application can use it to personalize a user's experience.

Personalizing a service can be beneficial for these reasons:

- It can reduce input requirements. In the movie ticketing application, it is tedious to re-enter a ZIP code with every use of the service, especially since the user is usually interested in seeing a movie close to home. Thus, the application

remembers the user's preferred ZIP code and uses it to retrieve movie schedules. The user has the option to override this ZIP code, which may be useful when he or she is visiting another city with a different ZIP code or wants to see a movie outside his or her immediate locale.

- It can shorten workflow. In the movie ticketing application, customers enter their account information when they first use the application, and the client keeps a copy of their login information. Customers can choose to automatically sign in to the application without having to go through a login screen on subsequent uses.

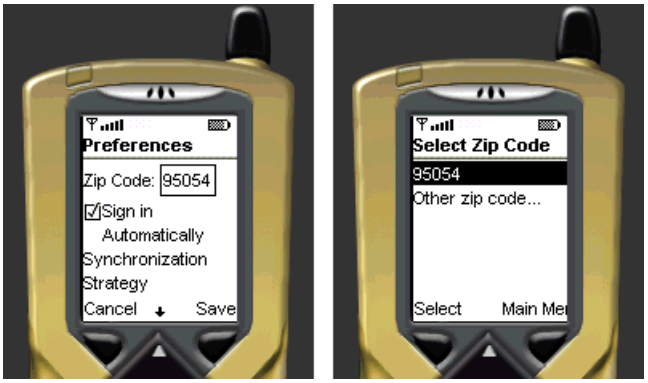

Figure 9 Inputting Personalization Data and Allowing the User to Override Personalization Data

In contrast to session state, which can be characterized as transient, personalization data is by nature persistent. The application developer decides where to persistently store personalization data.

Developers should consider whether personalization data is usable across multiple types of clients. For example, users who access the movie ticketing application from a cell phone might want to access the same application through a Web front end when they are at their workstation. When they do so, they may want their personal data, such as their ZIP code, to be available so they can avoid having to re-enter it through the Web front end. In this instance, it may help to store the personal data on the server.

The decision of where to store personalization data is not always an either/or decision. Personalization data may be distributed and even duplicated across the client and server. When personalization data is duplicated across client and server,

an application may need to incorporate additional facilities to synchronize the data. Developers are advised to consider the costs of incorporating such additional facilities.

4 Summary

This paper has described how to design and develop complete wireless enterprise solutions, like the Java Smart Ticket sample application, that have the power of Java technology from end to end. Such solutions can fully reap the benefits of Java technology: portability, scalability, security, and programming ease.

The high-level architecture of these solutions is simple, yet effective. Enterprise JavaBeans components provide business logic on top of enterprise data. Java servlets use EJB components to access and manipulate data on behalf of MIDP clients, which in turn present the data to the user. This division of responsibilities makes applications easier to implement, test, and maintain.

Most importantly, because EJB components are designed to be reusable, an enterprise application enabled for mobile clients can just as easily serve traditional desktop clients, such as Java applets and Web browsers. The result is a truly accessible enterprise: anytime, anywhere, from any device.

5 Resources

For more information on designing wireless enterprise applications using Java technology, and to download the Java Smart Ticket sample application, visit the Java BluePrints for Wireless Web site at:

```
http://java.sun.com/blueprints/wireless/
```

To get started building wireless enterprise application using Java technology, use the following tools and development kits:

- The J2ME Wireless Toolkit, available at
  `http://java.sun.com/products/j2mewtoolkit/`
- The J2EE Software Development Kit, available at
  `http://java.sun.com/j2ee/download.html#sdk`
- Various Java IDEs, available through

`http://java.sun.com/tools/`
including Sun ONE Studio, available at
`http://wwws.sun.com/software/sundev/jde/`

For introductory information on Java technologies, consult the following resources:

- *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/`>
- J2ME technology Web site <`http://java.sun.com/j2me/`>
- J2EE technology Web site <`http://java.sun.com/j2ee/`>
- *Programming Wireless Devices with the Java 2 Platform, Micro Edition.* R. Riggs, A. Taivalsaari, M. VandenBrink. Copyright 2001, Addison-Wesley.
- *The J2EE Tutorial.* S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan, B. Stearns. Copyright 2001, Addison-Wesley. Also available as <`http://java.sun.com/j2ee/tutorial/`>

The following references cover advanced topics in wireless and enterprise Java application design:

- *MIDP Style Guide.* Copyright 2002, Sun Microsystems, Inc. <`http://java.sun.com/j2me/docs/alt-html/midp-style-guide/style-guideTOC.html`>
- *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition.* I. Singh, B. Stearns, M. Johnson, Enterprise Team. Copyright 2002. Also available as <`http://java.sun.com/blueprints/enterprise/`>

Also consider joining the following interest lists hosted at Sun:

`kvm-interest@java.sun.com`
`j2eeblueprints-interest@java.sun.com`

Details on how to subscribe to these lists are available at:

`http://archives.java.sun.com`

Readers may send comments on this paper to:

`javablueprintswireless-feedback@sun.com`