

---

# Server-Supported Internationalization of Wireless Java Applications

*by Jon Ellis*

*Please send comments on this **draft** paper to [javablueprintswireless-feedback@sun.com](mailto: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/>.*

Mobile devices, more so than desktop computers, are used throughout the world. By internationalizing and localizing its content, a mobile application can provide the best experience for the widest possible audience.

This paper describes how to internationalize and localize Java technology-based wireless enterprise applications. Although the constraints of mobile devices make internationalization challenging, particularly on the client side, application developers can use the server side of the mobile client/server architecture to support the internationalization effort. This paper helps developers apportion handling internationalization work between the client and the server.

This paper illustrates its design guidelines using the Java Smart Ticket sample application, which is available through the Java BluePrints Web site at <http://java.sun.com/blueprints/>. This sample application is an example mobile commerce application using a movie ticketing scenario. Its design principles can be applied to other mobile services based on Java technology.

Note that this paper is one of several Java BluePrints for Wireless papers. It limits itself to the issues of server-supported internationalization of wireless clients and applications and leaves the basics of designing Java wireless enterprise

*Contents © 1999-2003 by Sun Microsystems, Inc. All rights reserved.*

*Please send comments on this **draft** paper to [javablueprintswireless-feedback@sun.com](mailto:javablueprintswireless-feedback@sun.com).*

*For the latest version of this paper, please visit <http://java.sun.com/blueprints/wireless/>.*

*Document last modified: April 17, 2003 7:56 am*

applications, such as communication, session management, security, and so forth, to the other papers.

## 1 Background

The Java 2 Platform, Standard Edition (J2SE) and Java 2 Platform, Enterprise Edition (J2EE) provide a rich framework and programming model for internationalization and localization. Standard classes such as `java.util.ResourceBundle` allow applications to support users in many different languages, and these classes may be used in conjunction with enterprise information systems to deliver internationalized content over the network. (For information on `java.util.ResourceBundle` and other standard classes, see *The Java Tutorial*. For information on designing internationalized enterprise applications, see *Designing Enterprise Applications with the J2EE Platform, Second Edition*.)

Developers internationalizing Java 2 Platform, Micro Edition (J2ME) applications targeted for the Mobile Information Device Profile (MIDP) need to consider these limitations of MIDP devices:

- No API for internationalization—MIDP developers need to provide a framework for doing internationalization because MIDP devices do not have functionality equivalent to `java.util.ResourceBundle`. Fortunately, many techniques described in the documentation of `java.util.ResourceBundle` apply for MIDP devices.
- Limited support for encodings—Mobile devices usually support only a single encoding specific to their primary sales region. For example, devices sold in Japan typically support only Japanese character encodings.
- Limited memory, persistent storage, and bandwidth—A client may offload internationalization and localization work to the server to save memory and storage space on the device, but this is done at the expense of bandwidth to download localized messages. On the other extreme, a client that handles all internationalization and localization issues does so at the expense of exhausting limited resources on the device.

## 2 Design Guidelines

Guidelines exist for developing and delivering internationalized and localized applications. The following steps, discussed in detail in the subsequent sections, are recommended for internationalizing and localizing mobile applications:

- Identifying content to be localized
- Isolating and translating the data to be localized from the application
- Packaging and delivering the localized data

### 2.1 Identifying What Needs to be Localized

Developers should identify the content that needs to be localized, isolate that information from the application code, then decide where to store the information. All content displayed to the user, including labels, icons, and error messages, needs to be localized. The decision about where to store localized content is largely influenced by whether the content is static or dynamic.

Static content is usually kept in the application JAR file and then stored locally on the client. An example of static content is the user interface (UI) component data in the Java Smart Ticket sample application. Because it does not change during an application session, the UI component data can be packaged with the application JAR file. When the client downloads the JAR file, the localized data is stored locally on the client device.

Dynamic content, on the other hand, is kept on the server separate from the application JAR file so that the client can download content for its own particular needs. Dynamic content in the sample application includes movie titles or movie reviews, the retrieval of which depends on criteria specified by the user. The server must return to the client content that is translated appropriately based on the locale that the client requests.

This paper focuses on the internationalization of static content. For information on internationalizing dynamic content, which is generally the domain of the server, see *Designing Enterprise Applications with the J2EE Platform, Second Edition*.

### 2.2 Isolating and Translating Data for Localization

Resource bundles, such as those represented by the `ResourceBundle` API, store localized data so that an application does not need to include the data in its code, but

*Contents © 1999-2003 by Sun Microsystems, Inc. All rights reserved.*

*Please send comments on this **draft** paper to [javablueprintswireless-feedback@sun.com](mailto:javablueprintswireless-feedback@sun.com).*

*For the latest version of this paper, please visit <http://java.sun.com/blueprints/wireless/>.*

*Document last modified: April 17, 2003 7:56 am*

instead can access the data from the bundle at runtime. By keeping localized data bundled separate from the application code, an application can be easily deployed in more than one language. Although MIDP devices do not have access to the `ResourceBundle` API, which J2SE application developers use to perform internationalization, MIDP application developers can easily implement a similar mechanism.

J2SE application developers often combine `ResourceBundles` with property files. Each property file contains a list of unique keys and the localized data corresponding to the keys. At runtime, an application specifies a locale and key to access the desired localized message. To improve code readability, a localized message key is a string describing the message.

The Java Smart Ticket sample application implements behaviors similar to the `ResourceBundle` API. The sample application moves localizable message strings from the application code to a separate file and provides integer keys to reference these strings. Using integers instead of strings as keys to reference the localized messages saves space.

```
public class IndexedResourceBundle {
    private String[] resources;
    // ...
    public String getString(int resourceId) {
        return (resourceId >= 0 && resourceId < resources.length)
            ? resources[resourceId] : null;
    }
    // ...
}
```

**Code Example 0.1** `IndexedResourceBundle`, a Class Similar to `ResourceBundle`

To improve readability, the application uses named constants rather than integer literals. The name of the constant serves as an identifier of the localized message.

```
public class UIController {
    // ...
    public String getString(int uiConstant) {
        return resourceBundle.getString(uiConstant);
    }
}
```

```
    // ...
}

public class AccountInfoUI extends Form
    implements CommandListener, ItemStateListener {
    // ...
    public AccountInfoUI(UIController uiController) {
        super(uiController.getString(
            UIMessageCodes.ACCOUNT_INFO));
        saveCommand = new Command(
            uiController.getString(UIMessageCodes.SAVE),
            Command.SCREEN, 1);
        // ...
    }
    // ...
}
```

### Code Example 0.2 Using IndexedResourceBundle

```
public final class UIMessageCodes {
    public static final int MAIN_MENU = 0;
    public static final int ACCOUNT_INFO = 1;
    public static final int MOVIE_RATINGS = 2;
    public static final int MOVIES = 3;
    public static final int PREFERENCES = 4;
    // ...
}
```

### Code Example 0.3 Named Constants Identifying Localized Messages

The constants are matched to the localized messages in a mapping file resembling a property file. This file is packaged in the application JAR file. Developers should ensure that the mapping file uses an encoding supported by most devices. Generally, messages are encoded in ASCII, which is a subset of most commonly-supported encodings. Once they are separated from the application code, messages requiring localization can be translated and placed in files containing the localized messages.

```

0=Main Menu
1=Account Info
2=Movie Ratings
3=Movies
4=Preferences
...

```

#### Code Example 0.4 Message Mapping File

### 2.3 Packaging and Delivering Localized Data

Every internationalized application must initialize with a default locale because the end user cannot specify a locale until the application starts. Consequently, application developers must select the default locale and package in the application JAR file messages corresponding to this locale. However, developers should also make it easy for users to select a different locale and retrieve localized messages from the server.

Localized data can be delivered in one of two ways:

- Include the data for all locales in the application JAR file and deliver localized data on demand
- Include only the default localized data in the application JAR file

Developers choose a delivery method based on the amount of localized UI data that must be delivered and the encoding supported by the device. There are advantages and disadvantages to both methods, and the next sections explain these trade offs.

#### 2.3.1 Delivering All Locales in the JAR File

When a MIDP application supports a set of locales, rather than just a single locale, it may be better to include localized data for all locales in the application JAR file. This permits a user to use a locale other than the default locale without making another (expensive) connection to the server and transferring another set of localized data to the device.

However, this delivery method is not always practical for mobile devices because of their limited storage space. Bundling data for all locales is not recommended when an application has large amounts of localized data to transfer to the

client or supports many locales. This method has drawbacks even if the application does not deliver large amounts of localized data, because future enhancements to the application may likely produce additional localized data and increase the JAR file size.

Bundling all locales in one JAR file is needlessly resource expensive—in terms of download time and memory or storage space—if the mobile device does not support encoding for all locales, as is often the case. There is no need for such a device to download and store a large JAR file with multiple locale encodings. Many cell phones support only the encoding representing the language of the region in which they are sold. For example, a cell phone sold in Pakistan might not support the encoding for Korean. The user of this phone only wants localized data in Urdu and does not want to download or store other locales.

### 2.3.2 Delivering Only the Default Locale in the JAR File

A developer can minimize download time and take less storage space on the device by packaging only the default locale in the application JAR file. The JAR file contains only the localized data needed by the user. If the user wants to change the locale, he or she can easily download additional localized data separately from the server.

However, because this delivery technique results in separate sets of localized data, the application development process is more complicated. To internationalize this application, a developer selects one locale to be the default locale. If the default locale is not the user's desired one, the user must reconnect to the server to download the correct locale. To deploy the application to a diverse set of clients, a developer needs to assemble the JAR file with alternate default locales for different clients. While the build process for the Java Smart Ticket sample application takes this into account, it is an added development consideration.

Because a user may select a new locale after downloading an application, a developer must also keep in mind how the user makes this locale selection so that the application starts with the desired locale. For example, in the Java Smart Ticket sample application, the user chooses a new locale from a menu dynamically downloaded from the server. Because the menu's locale choices are not included in the JAR file, the application developer can add support for additional locales after the application is made available for download. By retrieving the list of locales from the server, the client is assured of having the current list of supported locales.

Because the internationalization issues discussed here relate to presentation, the server-side responsibilities in this design are best handled by the Web tier of the application, thereby shielding the lower tiers from these issues. In the movie ticketing application, message mapping files are packaged as resources for the Web component containing the servlet that processes client requests. The servlet provides the list of locales for which mapping files are available, and dispenses requested mapping files to clients, serializing the mapping files appropriately.

### 3 Conclusion

This paper has described how to develop internationalized wireless Java applications with the support of a Java application server. Although J2ME MIDP devices do not have as rich support for internationalization as J2SE devices, they may nevertheless employ the strategies described in this paper to provide a rewarding user experience.

### 4 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://www.sun.com/software/sundev/jde/>

For introductory information on wireless and enterprise 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](mailto:kvm-interest@java.sun.com)  
[j2eeblueprints-interest@java.sun.com](mailto: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:

*Contents* © 1999-2003 by Sun Microsystems, Inc. All rights reserved.  
Please send comments on this **draft** paper to [javablueprintswireless-feedback@sun.com](mailto:javablueprintswireless-feedback@sun.com).  
For the latest version of this paper, please visit <http://java.sun.com/blueprints/wireless/>.  
Document last modified: April 17, 2003 7:56 am

`javablueprintswireless-feedback@sun.com`