Oracle® Retail POS Suite

Implementation Guide, Volume 5 – Mobile Point-of-Service

Release 14.0

E50544-01

December 2013

ORACLE®

Oracle Retail POS Suite Implementation Guide, Volume 5 - Mobile Point-of-Service, Release 14.0

E50544-01

Primary Author: Bernadette Goodman

Contributing Author: Tony Zgarba, John Yopp, Chuck Pilon

reconfigurations, reassembly or reverse assembly, re-engineering or reverse engineering and recompilations or reverse compilations of the VAR Applications or any derivatives of the VAR Applications. You acknowledge that it shall be a breach of the agreement to utilize the relationship, and/or confidential information of the VAR Applications for purposes of competitive discovery.

The VAR Applications contain trade secrets of Oracle and Oracle's licensors and Customer shall not attempt, cause, or permit the alteration, decompilation, reverse engineering, disassembly or other reduction of the VAR Applications to a human perceivable form. Oracle reserves the right to replace, with functional equivalent software, any of the VAR Applications in future releases of the applicable program.

Contents

3 Mobile POS Server Overview

4 Mobile POS Frameworks

5 Mobile POS Extension Guidelines

6 Implementation Environment

7 Internationalization

A Appendix: API URLs

B Appendix: API Result Formats

C Appendix: Eclipse Screens

List of Examples

List of Figures

Send Us Your Comments

Oracle Retail POS Suite Implementation Guide, Volume 5 - Mobile Point-of-Service, Release 14.0

Oracle welcomes customers' comments and suggestions on the quality and usefulness of this document.

Your feedback is important, and helps us to best meet your needs as a user of our products. For example:

- Are the implementation steps correct and complete?
- Did you understand the context of the procedures?
- Did you find any errors in the information?
- Does the structure of the information help you with your tasks?
- Do you need different information or graphics? If so, where, and in what format?
- Are the examples correct? Do you need more examples?

If you find any errors or have any other suggestions for improvement, then please tell us your name, the name of the company who has licensed our products, the title and part number of the documentation and the chapter, section, and page number (if available).

> **Note:** Before sending us your comments, you might like to check that you have the latest version of the document and if any concerns are already addressed. To do this, access the Online Documentation available on the Oracle Technology Network web site. It contains the most current Documentation Library plus all documents revised or released recently.

Send your comments to us using the electronic mail address: retail-doc_us@oracle.com

Please give your name, address, electronic mail address, and telephone number (optional).

If you need assistance with Oracle software, then please contact your support representative or Oracle Support Services.

If you require training or instruction in using Oracle software, then please contact your Oracle local office and inquire about our Oracle University offerings. A list of Oracle offices is available on our web site at http://www.oracle.com.

Preface

This Implementation Guide describes the requirements and procedures to extend and customize this Oracle Retail Mobile Point-of-Service release.

Audience

This Implementation Guide is intended for Oracle Retail Mobile Point-of-Service application integration and implementation staff.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following documents in the Oracle Retail Mobile Point-of-Service Release 14.0 documentation set:

- *Oracle Retail Mobile Point-of-Service Release Notes*
- *Oracle Retail Point-of-Service Installation Guide*
- *Oracle Retail Point-of-Service User Guide*
- *Oracle Retail POS Suite Configuration Guide*
- *Oracle Retail POS Suite Data Dictionary*
- *Oracle Retail POS Suite Data Model Differences*
- *Oracle Retail POS Suite Data Model ERWIN File*
- *Oracle Retail POS Suite Data Model Mapping File*
- *Oracle Retail POS Suite Entity Relationship Diagrams, Volume 1 - Subject Areas*
- *Oracle Retail POS Suite Entity Relationship Diagrams, Volume 2 - Overviews*

- *Oracle Retail POS Suite Implementation Guide, Volume 1 - Implementation Solutions*
- *Oracle Retail POS Suite Implementation Guide, Volume 2 - Extension Solutions*
- *Oracle Retail POS Suite Implementation Guide, Volume 4 - Point-of-Service External Order*
- *Oracle Retail POS Suite Operations Guide*
- *Oracle Retail POS Suite Security Guide*
- *Oracle Retail POS Suite 14.0/Merchandising Operations Management 14.0 Implementation Guide*

Customer Support

To contact Oracle Customer Support, access My Oracle Support at the following URL:

https://support.oracle.com

When contacting Customer Support, please provide the following:

- Product version and program/module name
- Functional and technical description of the problem (include business impact)
- Detailed step-by-step instructions to re-create
- Exact error message received
- Screen shots of each step you take

Review Patch Documentation

When you install the application for the first time, you install either a base release (for example, 14.0) or a later patch release (for example, 14.0.1). If you are installing the base release or additional patch releases, read the documentation for all releases that have occurred since the base release before you begin installation. Documentation for patch releases can contain critical information related to the base release, as well as information about code changes since the base release.

Improved Process for Oracle Retail Documentation Corrections

To more quickly address critical corrections to Oracle Retail documentation content, Oracle Retail documentation may be republished whenever a critical correction is needed. For critical corrections, the republication of an Oracle Retail document may at times not be attached to a numbered software release; instead, the Oracle Retail document will simply be replaced on the Oracle Technology Network Web site, or, in the case of Data Models, to the applicable My Oracle Support Documentation container where they reside.

This process will prevent delays in making critical corrections available to customers. For the customer, it means that before you begin installation, you must verify that you have the most recent version of the Oracle Retail documentation set. Oracle Retail documentation is available on the Oracle Technology Network at the following URL:

http://www.oracle.com/technetwork/documentation/oracle-retail-100266.html

An updated version of the applicable Oracle Retail document is indicated by Oracle part number, as well as print date (month and year). An updated version uses the

same part number, with a higher-numbered suffix. For example, part number E123456-02 is an updated version of a document with part number E123456-01.

If a more recent version of a document is available, that version supersedes all previous versions.

Oracle Retail Documentation on the Oracle Technology Network

Documentation is packaged with each Oracle Retail product release. Oracle Retail product documentation is also available on the following web site:

http://www.oracle.com/technetwork/documentation/oracle-retail-100266.html

(Data Model documents are not available through Oracle Technology Network. These documents are packaged with released code, or you can obtain them through My Oracle Support.)

Documentation should be available on this web site within a month after a product release.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction

Oracle Retail Mobile Point-of-Service is an extension of Oracle Retail Point-of-Service. It provides the capability to use a mobile device for processing transactions. It provides a subset of the capabilities that are available on a register for entering transactions, tendering transactions, and looking up items.

Contents of this Guide

This implementation guide addresses the following topics:

- Chapter 1, "Introduction": Overview of Mobile Point-of-Service and the skills needed for implementation.
- Chapter 2, "Mobile POS User Interface Customization": Guidelines for customizing the Mobile POS User Interface.
- Chapter 3, "Mobile POS Server Overview": Overview of the Mobile POS Server.
- Chapter 4, "Mobile POS Frameworks": Description of the frameworks used for Mobile Point-of-Service.
- Chapter 5, "Mobile POS Extension Guidelines": Guidelines for extending and customizing Mobile Point-of-Service.
- Chapter 6, "Implementation Environment": Information on how to set up a single-user development environment.
- Chapter 7, "Internationalization": Translations provided for Mobile Point-of-Service.
- Appendix A, "Appendix: API URLs": Overview of the API URLs, including parameter and return types.
- Appendix B, "Appendix: API Result Formats": Messages returned by the APIs.
- Appendix C, "Appendix: Eclipse Screens": Screenshots from Eclipse that illustrate steps to create an implementation environment.

Key Features of Mobile Point-of-Service

Mobile Point-of-Service provides the following features:

- Scan items or enter the item numbers manually. Scan or enter a serial number if required for an item.
- Sell and activate gift cards.
- Look up item availability.

- Apply discounts and price overrides to items. Apply discounts to transactions.
- Tender transactions using a credit card, debit card, or gift card.
- Print and email receipts.
- Suspend a transaction in order to complete it at a register.

If Oracle Retail Mobile Point-of-Service is implemented with Oracle Retail Store Inventory Management, the following Oracle Retail Store Inventory Management functionality is supported:

- Inventory lookup at the current store
- Inventory lookup at buddy stores
- Validation of serial numbers

Skills Needed for Implementation

The implementer needs an understanding of the following applications and technical concepts.

Applications

The implementer should understand the interface requirements of the integrated applications and data sources. The implementer needs this knowledge for the following applications:

- Oracle Retail Back Office
- Oracle Retail Point-of-Service
- Oracle Retail Store Inventory Management

Technical Concepts

The implementer should understand the following technical concepts:

- UNIX system administration, shell scripts, and job scheduling
- Eclipse
- Technical architecture for Mobile Point-of-Service
- Application servers, specifically servlet containers
- Java coding, including REST Java coding concepts
- XML manipulation

For customization of the user interface, the implementer should also understand the following technical concepts:

- JavaScript programming
- HTML programming
- CSS programming
- Xcode development
- Apple Enterprise Development setup and deployment
- Certificate creation and deployment

2

Mobile POS User Interface Customization

This chapter provides information on how to customize the user interface (UI) for the Mobile POS application. Examples of code changes are included.

See the *Oracle Retail Point-of-Service Installation Guide - Volume 1, Oracle Stack* for the following information:

- Setup for the Mobile POS Xcode project.
- Configuration and deployment of the Mobile POS UI distribution certificate.

Resources

Consult the following web sites for additional information.

	Web Site
CSS Gradients	http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/IconsImages/IconsImages.html
jQuery	http://jquery.com/
jQuery Mobile	http://jquerymobile.com/
PhoneGap	http://phonegap.com

Customization Files

Customization of Mobile POS should be limited, as much as possible, to the following two files:

- `mobilepos/www/css/customize.css`
- `mobilepos/www/js/customize.js`

These two files are loaded by Mobile POS (in `index.html`), but are not provided by Oracle. Therefore, any changes you put into these two files will not be overwritten when a new release or patch is installed.

Note: The Mobile POS JavaScript code interacts with the iPod Touch device using PhoneGap and several PhoneGap plug-ins. Modification of the PhoneGap plug-ins and development of new PhoneGap plug-ins are not covered in this document.

CSS Customization File

The styling of Mobile POS is controlled through Cascading Style Sheets (CSS). The style sheets used by Mobile POS are located in `mobilepos/www/css`.

To change the styling of Mobile POS, create a file named `customize.css` in `mobilepos/www/css` and place the style changes in this file. Although it is not provided in the Mobile POS distribution, `customize.css` is loaded after all other CSS style sheets, so any styles put into `customize.css` will override the other styles.

To change the color scheme of the large navigation buttons, create `customize.css` with the following:

Example 2–1 Changing the Color Scheme of Large Navigation Buttons

```
.large-navigation-button
{
    background-image: -webkit-linear-gradient(top, #FF170F, #EF5B5B);
}
```

JavaScript Customization File

Altering the business flow of or changing the text displayed by the Mobile POS UI requires coding in JavaScript.

Some custom code, such as the modification of displayed text, should not be executed until after the Mobile POS UI has successfully initialized. To handle this situation, bind to the `mpos-initialized` event in `customize.js` and provide a function to be executed once `mpos-initialized` is triggered. Mobile POS triggers this event when it successfully completes initialization.

Example 2–2 Triggering an Event at Completion of Mobile POS UI Initialization

```
(function()
{
    // Wait until mpos-initialized is triggered, indicating that MPOS is
    // initialized.
    $(document).bind("mpos-initialized", myCustomInitializationFunction);
})();

function myCustomInitializationFunction()
{
    // Custom initialization goes here.
    console.log("myCustomInitializationFunction");
}
```

In the JavaScript examples that follow, any reference to `myCustomInitializationFunction` assumes that it is called in response to the `mpos-initialized` event shown in Example 2–2.

Note: To write to the log file that is accessible from the About page, call `console.log` with the information to be logged.

iPod Touch Display

This section describes how to customize the iPod Touch display.

Changing the Application Name

To change the name that appears on the iPod Touch screen underneath the application icon:

1. Open the project in Xcode.
2. Open the Project Navigator panel (Command-1).
3. Click mobilepos at the top of the project hierarchy.
4. In the right panel, click mobilepos under Targets, click the Info tab, and then expand Custom iOS Target Properties.
5. Change the Bundle display name property to the desired application name and then repackage.

Alternatively, you can do the following:

1. Edit `mobilepos/mobilepos/mobilepos-Info.plist` and search for `CFBundleDisplayName`.
2. As shown in Example 2–3, change the line following the `CFBundleDisplayName` line to <string>*Application Name*</string>. Set *Application Name* to the name you want to use. Then repackage.

Example 2–3 Changing the Application Name

```
<key>CFBundleDisplayName</key>
<string>Gadgets POS</string>
```

Changing the Application Icon

To change the icon that appears on the iPod Touch screen, replace `/mobilepos/icon.png` and `/mobilepos/icon@2x.png` and then repackage using Xcode. The `icon.png` should be sized 57 pixels by 57 pixels, and `icon@2x.png` should by sized 114 pixels by 114 pixels. For more information, see the following web site:

http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/IconsImages/IconsImages.html

Changing the Splash Screen

The splash screen, also called the launch image, can be changed by replacing `/mobilepos/Default.png` and `/mobilepos/Default@2x.png` and then repackaging the application using Xcode. `Default.png` should be sized 320 pixels by 480 pixels, and `Default@2x.png` should by sized 640 pixels by 960 pixels. For more information, see the following web site:

http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/mobilehig/IconsImages/IconsImages.html#//apple_ref/doc/uid/TP40006556-CH14-SW5

Styling

This section describes changes that can be made to the style of the Mobile POS UI.

Changing the Color Scheme

Mobile POS uses the jQuery Mobile JavaScript library to style the UI elements. The styles are defined in `mobilepos/www/css/mobilepos.theme.css`, which is a copy of one of the five predefined swatches provided by the default jQuery Mobile theme.

To change the color scheme, copy `mobilepos.theme.css` into `customize.css` and override the colors. The following properties need to be adjusted:

- background
- background-image
- color
- text-shadow
- border

jQuery Mobile ThemeRoller is a tool written by the jQuery Mobile development team that can be used to select color combinations that work with jQuery Mobile.

Note: Plugging in the style sheet generated by ThemeRoller does not work with this release of Mobile POS.

Changing the Corporate Logo

An Oracle logo is shown at the top of the login and menu pages. To change the logo, copy a new image file into the `mobilepos/www/images` folder and then place the following in `mobilepos/www/css/customize.css`:

Example 2–4 Changing the Corporate Logo

```
[data-header="banner"]
{
    background-image: url("../images/company-logo.png");
}
```

In Example 2–4, the new logo file is named `company-logo.png`.

The image is scaled to 29 pixels high. If this height is not appropriate, the values of the height and background-size properties of the `[data-header="banner"]` CSS selector need to be adjusted as well.

Changing the Menu Button Color

The CSS style definition for the menu buttons is named `large-navigation-button`. The definition is found in `mobilepos/www/css/mobilepos.css`.

To change the style of the menu buttons, override the `large-navigation-button` class by putting in the new style attributes into `mobilepos/www/css/customize.css`.

To change the color of the menu buttons from a blue gradient to a red gradient, put the following into `customize.css`:

Example 2–5 Changing the Color of Menu Buttons

```
.large-navigation-button
```

```
{
     background-image: -webkit-linear-gradient(top, #FF170F, #EF5B5B);
}
```

If you change the background of the menu buttons, you should give a visual indication that the button is pressed by changing the background in the active pseudo-class. One way to do this is to swap the endpoints, as shown in Example 2–6:

Example 2–6 Changing the Background of Menu Buttons

```
.large-navigation-button:active
{
     background-image: -webkit-linear-gradient(top, #EF5B5B, # FF170F);
}
```

Changing the Menu Button Size

The size of the menu buttons is controlled by the size of the image, size of the text font, and padding around the text. Override the following properties in `customize.css`:

CSS Selector	Property
`.large-navigation-button img`	`height`
`.large-navigation-button-text`	`font-size`
`.large-navigation-button-text`	`padding`

To shrink the menu buttons, put the following into `customize.css`:

Example 2–7 Shrinking the Menu Buttons

```
.large-navigation-button img
{
     height: 40px;
}

.large-navigation-button-text
{
     padding-left: 5px;
     padding-right: 5px;
     padding-bottom: 5px;
     font-size: 12px;
}
```

Changing the Action Button Icons

Action buttons appear on the bottom of the screen in the footers. For example, the Pay button on the basket screen is an action button.

The icon displayed on an action button is controlled by the `data-icon` attribute, which can be seen in the HTML for the page. For example, `basket.html` shows that the `data-icon` attribute for the Pay button is `mpos-pay`.

```
<li><a href="#" data-icon="mpos-pay" id="basket-pay"></a></li>
```

To find the file that is used for the button icon, locate the class CSS class selector named `.ui-icon-` plus the value of the data-icon attribute in `mobilepos.css`. For example, the CSS class selector for the Pay button is `.ui-icon-mpos-pay`.

The file that contains the icon is specified in the background property. To change the icon, override the background property of the style in `customize.css`. Put the following in `customize.css` to change the Pay button icon to a file named `my-pay-icon.png`:

Example 2–8 Changing an Action Button Icon

```
.ui-icon-mpos-pay
{
     background:  url("../images/my-pay-icon.png") 50% 50% no-repeat;
}
```

Action button icons should be 30 pixels wide by 30 pixels high.

Hiding the Action Button Text

To gain additional content space on the screens with action buttons, the action button text can be hidden and the toolbar decreased in height. To do this, place the following code in `customize.css`:

Example 2–9 Hiding Action Button Text

```
[data-footer="square-row"] .ui-btn-text
{
     display: none;
}

[data-footer="square-row"]
{
    height: 45px;
}
```

This can be applied on a page-by-page basis by pre-pending the page identifier to the CSS selector. The page identifier is found in the HTML for the page. Look for the `div` with `data-role="page"`. The page identifier is the value of the id attribute. The page identifier for the basket page is `basket-page`, which is specified by this line in `basket.html`:

```
<div data-role="page" id="basket-page" data-theme="o">
```

Example 2–10 shows the code to place in `customize.css` to hide the button text only on the basket screen:

Example 2–10 Hiding Button Text on the Basket Screen

```
#basket-page [data-footer="square-row"] .ui-btn-text
{
     display: none;
}

#basket-page [data-footer="square-row"]
{
     height: 45px;
}
```

Changing Text

The text displayed for field labels, button text, and error messages is read from the resource bundles located in `mobilepos/www/js/translations`. The resource

bundles are named `Resources_language-identifier_region-identifier.js`. For example, the resource bundle for United States English is `Resources_en_US.js`.

The content of a resource file is structured as an object consisting of key-value pairs. Mobile POS uses a key string to look up the translated value in the resource file that corresponds to the locale currently configured on the mobile device.

To change the text that is displayed:

1. Inspect the JavaScript and resource bundles to determine the keys that are being used to retrieve the text from the resource bundle. For example, to change the "Welcome to Mobile POS" text, you can search the resource bundle for this text and see from the following line that the message key is `%login-welcome-text-1`:

   ```
   "%login-welcome-text-1": "Welcome to Mobile POS.",
   ```

2. In a function called after Mobile POS is initialized (see "JavaScript Customization File"), construct an object with the message keys found in Step 1 and the new values.

3. Call `Globalize.addCultureInfo` to replace the default message key values with the new values.

In Example 2–11, the login screen strings "Welcome to Mobile POS" and "Please Log In" are replaced with "Welcome to Custom Mobile POS" and an empty string.

Example 2–11 Changing Displayed Text

```
function onCustomMPOSInitialized()
{
    // Load custom strings into the current locale.
    var customStrings =
    {
       "%login-welcome-text-1": "Welcome to Custom Mobile POS",
       "%login-welcome-text-2": "",
    };

var globalizeLocale = MobilePOSDevice.getGlobalizeLocale();
Globalize.addCultureInfo(globalizeLocale,
{
   messages: customStrings
});
}
```

If the implementation of Mobile POS is supporting more than one locale, different values for the keys by language can be implemented by calling `Globalize.addCultureInfo` and passing in the specific locale.

Example 2–12 shows how to override the welcome strings for both Greek-Greece and German-Germany:

Example 2–12 Changing Displayed Text in Specific Locales

```
var customGreekStrings =
{
     "%login-welcome-text-1": "Καλώς ήρθατε στο Custom Mobile POS.",
     "%login-welcome-text-2": "",
};

Globalize.addCultureInfo("el-GR",
{
```

```
    messages: customGreekStrings
});

var customGermanStrings =
{
    "%login-welcome-text-1": " Willkommen bei Custom Mobile POS.",
    "%login-welcome-text-2": "",
};

Globalize.addCultureInfo("de-DE",
{
    messages: customGermanStrings
});
```

Notes: Some values, such as reason code descriptions, are provided to the UI by the POS Server and cannot be changed on the UI. They must be changed on the server.

The resource file name uses an underscore to separate the language and the region. However, the Globalize library uses a hyphen.

Business Flow

This section describes customizing business flows.

Changing the Menu

The menu entries are defined in the `menuEntries` table in `menu.js`. The JavaScript in `menu.js` dynamically creates the HTML for the menu page based on the entries in `menuEntries`.

Each entry in `menuEntries` has the following fields:

Field	Description
id	HTML ID of the menu button.
labelKey	Message key used to look up the button text in the resource bundle.
permission	POS access point used to enable or disable the button.
handler	Function called when button is pressed.
img	Path to image to display on the button.

Removing a Menu Button

To remove an item from the menu, call `menuController.removeMenuEntry` in `customize.js` after Mobile POS is initialized, passing in the ID of the button you want removed.

Example 2–13 shows how to remove the Close Till button:

Example 2–13 Removing the Close Till Button

```
function myCustomInitializationFunction()
{
    // Custom initialization goes here.
    menuController.removeMenuEntry("menu-close-till");
}
```

Adding a Menu Button

To add an item to the menu, call `menuController.addMenuEntry` in `customize.js` after Mobile POS is initialized, passing in the index where you want the menu entry inserted, as well as the menu entry fields previously described.

Example 2–14 shows how to add a Settings button to the menu:

Example 2–14 Adding a Settings Button to a Menu

```
function myCustomInitializationFunction()
{
    // Custom initialization goes here.
    menuController.addMenuEntry(2, "menu-settings", "%settings-menu-button",
        Permission.administration, onMenuSettings, "images/settings.png");
}
```

For instructions on how to code screens in response to a button click, see "Workflows".

Updating a Menu Entry

To update an item in the menu, call `menuController.updateMenuEntry` in `customize.js` after Mobile POS is initialized. Pass in the ID of the button you want updated as well as the fields you want updated. Specify undefined to leave a field unchanged.

The function signature for `menuController.updateMenuEntry` is:

```
function(id, labelKey, permission, handler, img)
```

The parameters are the same as the fields in the `menuEntries` table previously described.

Example 2–15 shows how to change the Item Lookup button to show a different icon and to call a different function when clicked:

Example 2–15 Changing a Menu Entry

```
function myCustomInitializationFunction()
{
    // Custom initialization goes here.
    menuController.updateMenuEntry("menu-item-lookup", undefined,
        undefined, onCustomItemLookup, "images/myCustomItemLookup.png")
}
```

Changing the Number of Columns

The number of columns of buttons shown on the menu defaults to 2. To change the number of columns, call `MobilePOSConfig.setMenuColumns` and specify the new number of columns. See Example 2–16.

Example 2–16 Changing the Number of Columns

```
function myCustomInitializationFunction()
{
    // Custom initialization goes here.
    MobilePOSConfig.setMenuColumns(3);
}
```

Workflows

The Navigator object defined in `navigator.js` provides a wrapper for page navigation. This object works with the entries in the NavigationMap to move from page to page.

Navigator uses jQuery Mobile's `changePage` method to navigate from page to page. For detailed information on `changePage`, see the following web site:

`http://jquerymobile.com/test/docs/api/methods.html`

Adding a New Page

To add a new page, code the following:

1. Call the `addPage` method of `Navigator`, passing in the following values:
   - `page`: string used as a key in `NavigationMap`.
   - `file`: name of the HTML file that defines the page structure.
   - `options`: object to pass to `changePage`.
   - `authorization`: POS permission to check before navigating to the page. If the permission is not granted to the logged-in user, an exception is thrown.
   - `initFunction`: function to be called before navigating to the page.
2. Insert a new function into `Navigator` that calls `gotoPage`, specifying the newly added page.
3. Develop the HTML for the new page.
4. Develop any required initialization in the function specified in `initFunction`.
5. Add functions to load the page fields and respond to events, such as button clicks.

Example 2–17 defines a settings page that is displayed when a custom Settings button is clicked on the menu:

Example 2–17 Defining a New Settings Page

```
(function()
{
    // Wait until mpos-initialized is triggered, meaning MPOS is initialized.
    $(document).bind("mpos-initialized", onCustomMPOSInitialized);
})();

function onCustomMPOSInitialized()
{
    // Add the settings page as a new menu item
    menuController.addMenuEntry(2, "menu-settings", "%settings-menu-button",
        Permission.administration, onMenuSettings, "images/settings.png");

    // Add the settings page to the Navigator
    Navigator.addPage("settings", "settings.html", undefined,
        Permission.administration, initSettingsPage);

    Navigator.gotoSettings = function(options)
    {
        this.gotoPage("settings", options);
    };

    // Load custom translation strings.
    var customLanguageBundle =
```

```
    {
       "%settings-menu-button": "Settings",
       "%error-header-settings": "Settings Error",
       "%settings-header": "Settings",
       "%settings-1-label": "Setting 1",
       "%settings-2-label": "Setting 2",
       "%settings-cancel-button": "Cancel",
       "%settings-save-button": "Save",
        "%settings-error": "Settings Error"
    };

    Globalize.addCultureInfo(MobilePOSDevice.getGlobalizeLocale(),
    {
       messages: customLanguageBundle
    });
}

function onMenuSettings(event)
{
    try
    {
       Navigator.gotoSettings();
    }
    catch(e)
    {
       handleException(e, "%error-header-settings");
       throw e;
    }
}

function initSettingsPage(options)
{
    console.log("initSettingsPage");
    $("#settings-page").die("pageinit", loadSettingsPageFields);
    $("#settings-page").live("pageinit", options, loadSettingsPageFields);
}

function loadSettingsPageFields(event)
{
    var options = event ? event.data : {};

    try
    {
        $("#settings-header").text(translate("%settings-header"));
        $("#settings-1-label").text(translate("%settings-1-label"));
        $("#settings-2-label").text(translate("%settings-2-label"));

        $("#settings-cancel").buttonText(translate("%settings-cancel-button"));
        $("#settings-save").buttonText(translate("%settings-save-button"));

        $("#settings-cancel").unbind("click", onSettingsCancel);
        $("#settings-cancel").bind("click", onSettingsCancel);

        $("#settings-save").unbind("click", onSettingsSave);
        $("#settings-save").bind("click", onSettingsSave);

        // TODO: Load settings

    }
    catch (e)
```

```
    {
        handleException(e, "%settings-error", {});
    }
}

function onSettingsCancel(e)
{
    console.log("Cancel settings");
    Navigator.gotoMenu();
}

function onSettingsSave(e)
{
    console.log("Save settings");
    // TODO: Save settings
    Navigator.gotoMenu();
}
```

Adding a New Workflow

New workflows can be added to Mobile POS by adding one or more new pages for the workflow (see "Adding a New Page") and then chaining the pages together through action buttons in the footer. The first page in the workflow can be triggered by a new menu button (see "Adding a Menu Button"), or it can be triggered in response to some other event.

Suppose a new workflow is needed consisting of two new pages. Develop the navigation of the new workflow:

1. Create the two new pages according to instructions in "Adding a New Page". In this example, the names of the functions added to `Navigator` are `gotoPage1` and `gotoPage2`.
2. On the first page, include back and next buttons:
    a. In the back button handler, call `Navigator.gotoMenu`.
    b. In the next button handler, call `Navigator.gotoPage2`.
3. On the second page, include back and done buttons:
    a. In the back button handler, call `Navigator.gotoPage1`.
    b. In the done button handler, perform the work of the workflow, then call `Navigator.gotoMenu`.

Inserting a Page into an Existing Workflow

To insert a new page into an existing workflow:

1. Create the new page. See "Adding a New Page".
2. Determine which `Navigator` function is used to navigate to the next page in the workflow by inspecting the JavaScript source that controls the page that displays just before the new page.
3. Override the function found in Step 2 to navigate to the new page created in Step 1. If multiple workflows use the same `Navigator` function, use the value of `$.mobile.activePage[0].id` to determine the active page and only override if the active page is part of the workflow being altered.
4. Update the new page to call the `Navigator` function found in Step 2, effectively resuming the original workflow.

For example, suppose after a successful tender, a page should be displayed with a script to be read to the customer:

1. Create the new page. See "Adding a New Page".
2. Inspect `receiptPage.js` and ensure that `Navigator.gotoMenuOrBasket` is called when the user is done with a transaction.
3. By searching through all of the code, ensure that `Navigator.gotoMenuOrBasket` is called from multiple places (such as cancel and suspend) and so `$.mobile.activePage.id` must be used to ensure that the new page is shown only when the active page is the receipt page.
4. Override `Navigator.gotoMenuOrBasket` to call the new page if the active page is "receipt-page", the page ID for the receipt page.

Example 2–18 Inserting a Page into an Existing Workflow

```
function onCustomMPOSInitialized()
{
    // Create new script page
    ...

    // Insert the script page after receipt before basket/menu
    Navigator._gotoMenuOrBasket = Navigator.gotoMenuOrBasket;
    Navigator.gotoMenuOrBasket = function(options)
    {
        var activePageId = $.mobile.activePage[0].id;

        if (activePageId === "receipt-page")
            Navigator.gotoScript();
        else
            Navigator._gotoMenuOrBasket();
}
```

5. In the handler for the Done or OK button on the new script page, call `Navigator.gotoMenuOrBasket` to complete the workflow.

Replacing a Page in an Existing Workflow

The HTML for a page can be replaced by using `Navigator.replacePage`. The same JavaScript is executed and styles are applied, provided the HTML elements retain the appropriate IDs and classes. This can be useful for placing additional fields on a screen or changing the look and feel without having to use JavaScript to dynamically add the fields.

Example 2–19 shows replacing the HTML in the login screen, `login.html`, with `customlogin.html`:

Example 2–19 Replacing a Page in an Existing Workflow

```
function onCustomMPOSInitialized()
{
    Navigator.replacePage("login", "customlogin.html", { transition: "fade" },
        undefined, initLoginPage);
}
```

Inserting a Static Field on a Page

To insert a static field on a page, use one of the DOM manipulation methods provided by jQuery. The methods that allow insertion *outside* an existing element are found at the following web site:

http://api.jquery.com/category/manipulation/dom-insertion-outside/

The methods that allow insertion *inside* an existing element are documented at the following web site:

http://api.jquery.com/category/manipulation/dom-insertion-inside/

Manipulation of the fields in the DOM cannot take place at application startup or initialization. It must take place after the page is loaded into memory. jQuery provides triggers to communicate when a page is loaded (`pageinit`), about to be shown (`pagebeforeshow`), or has just been shown (`pageshow`). These methods are documented at the following web site under page transition events:

http://jquerymobile.com/test/docs/api/events.html

In most cases, placing modification code in a function triggered by `pagebeforeshow` is sufficient. However, in some cases, such as when working with elements marked up by jQuery, modifications must be made in a function triggered by `pageshow`. When possible, `pagebeforeshow` should be used because the user can see the field being added when the insert is done in `pageshow`.

Example 2–20 shows adding a third welcome line on the login screen by using `$.after` in a function triggered by `pagebeforeshow`:

Example 2–20 Adding a Third Welcome Line on the Login Screen

```
function onCustomMPOSInitialized()
{
    // Load custom translation strings into the current locale.
    var customLanguageBundle =
    {
        "%login-welcome-text-3": "Have a Nice Day",
    };

    Globalize.addCultureInfo(globalizeLocale,
    {
        messages: customLanguageBundle
    });

    // Add third welcome text line
    $("#login-page").live("pagebeforeshow",
        function()
        {
            $("#welcome-text-2").after("<div id='welcome-text-3'></div>");
            $("#welcome-text-3").text(translate("%login-welcome-text-3"));
        });
    }
```

jQuery provides helper methods that enable the developer to easily set the value of an HTML element. However, the method to use is not always readily evident. Here are rules to follow:

- If the HTML element's value is between the beginning and ending tags, such as with paragraphs and divs, use `$.text`.

- If the HTML element's value is set in the value attribute, such as with input, use `$.val`.
- As a special case, if the HTML element is an input of type button or if it is an anchor (link) styled by jQuery mobile, use `$.buttonText`.

Inserting an API-Filled, Non-Input Field on a Screen

Inserting an API-filled non-input field on a screen is very similar to inserting a static field on a page. The only difference comes in setting the text of the field. Instead of passing a static value to the jQuery method used to set the value (`$.text`, `$.val`, or `$.buttonText`), pass in the value returned from the API call.

For example, to display the number of line items on the payment page, dynamically insert a field on the payment page (see "Inserting a Static Field on a Page"), then populate the field with the number of items in the transaction. The current transaction is available as the Transaction object defined in `transaction.js`. See Example 2–21.

Example 2–21 Inserting an API-Filled Non-Input Field on a Screen

```
function onCustomMPOSInitialized()
{
    // Load custom translation strings into the current locale.
    var customLanguageBundle =
    {
        "%payment-lines-label": "Number of Items",
    };

    Globalize.addCultureInfo(globalizeLocale,
    {
       messages: customLanguageBundle
    });

    $("#payment-page").live("pagebeforeshow",
        function()
        {
            $("#tender-entry").after(
                   "<div data-role='fieldcontain'>"
                +      "<label for='payment-lines' "
                +          "id='payment-lines-label'></label>"
                +      "<p id='payment-lines'></p>"
                +  "</div>");
            $("#payment-lines-label").text(translate("%payment-lines-label"));
            $("#payment-lines").text(Transaction.getTotalQuantity());
        });
}
```

Inserting an Input Field on a Screen

Inserting an input field on a screen follows the same pattern as inserting a non-input field on a screen, but an extra step must be taken to ensure that the jQuery Mobile styling is applied. To apply the jQuery Mobile styling, call `trigger("create")` on the parent of the element inserted. When using the `append` method, the call to `trigger` can just be chained to the returned object:

```
$("#parent").append(newHTML).trigger("create");
```

Example 2–22 shows inserting a third input field on the login page named Secret:

Example 2–22 Inserting an Input Field on a Screen

```
function onCustomMPOSInitialized()
{
    $("#login-page").live("pagebeforeshow",
        function()
        {
            var secret =
                   "<div data-role='fieldcontain' id='secret-container'>"
                +"<label for='login-secret' "
                +"id='login-secret-label'>Secret</label>"
                +"<input id='login-secret'/>"
                +"</div>";
            $("#login-fields").append(secret).trigger("create");

            // Shrink top margin of login button to keep entire button on the
            // screen.
            $("#login-button").css("margin-top", "20px");
        });
}
```

Note: Consider using `Navigator` to replace the entire HTML page if there are more than a few fields dynamically added. See "Replacing a Page in an Existing Workflow".

Changing Field Validation

Field validation is performed by functions located in `validation.js`. Most fields are validated using regular expressions that check both valid characters and overall length, plus additional checks for minimum and maximum values. Some, such as email address validation, are much more involved.

To change the validation of a field, there are two options:

- Change the regular expression and other fields used to validate the field. The values are defined in the `Validation` object defined in `validation.js`. To change the values, override the current values with the new values.

  Example 2–23 shows changing the minimum length of an serial number from 1 to 5:

Example 2–23 Changing Field Validation

```
function onCustomMPOSInitialized()
{
    Validation.serialNumber =
        { re: /^[0-9a-z]{5,25}$/i, minLength: 5, maxLength: 25 };
}
```

Note: The `minLength` value is not actually used in the validation process. It exists so that custom error messages can be specific about the minimum length of a valid serial number.

- The second option is to override the validation function completely. To do this, code the new function in `customize.js`, using the same function signature. Since `customize.js` is loaded after `validation.js`, the customized version will be used.

Changing the Business Logic Behind a Screen Event

Screen events, such as a button being clicked, are handled through jQuery event handlers. Documentation on jQuery events is available at the following web site:

http://api.jquery.com/category/events/

How to change the logic behind a screen event depends upon how the screen event is currently handled:

- If the event is bound to a named function, replace the named function.

  The click event on the basket page action buttons are each bound to named functions, as seen in `basketPageActions.js`. To replace the functionality of the Reprint button, for example, code a new function for `onBasketReprintReceipt` in `customize.js`.

- If the event is bound to an anonymous function, use `unbind` to prevent execution of the anonymous function, then use `bind` to allow execution of the custom code.

  This unbinding and rebinding must take place after the original binding takes place, so further inspection of the code is necessary to determine at what part of the parent page's lifecycle the original binding is taking place. Then, a later stage of the page lifecycle can be used to perform the rebinding. Put another way, if a button click is bound to an anonymous function during a `pageinit`, the rebind must be performed during `pagebeforeshow` or `pageshow`.

  For example, suppose the functionality of print receipt is customized and a new page is needed to collect additional information. To switch to this new page instead of immediately printing the receipt, inspect `ReceiptPage.js`. The Print Receipt button has an anonymous function bound to the click event during `pageinit`.

  To modify this behavior, rebind a new function during `pagebeforeshow`, as shown in Example 2–24:

Example 2–24 Changing the Business Logic Behind a Screen Event

```
function onCustomMPOSInitialized()
{
    $("#receipt-page").live("pagebeforeshow",
        function(e)
        {
            $("#receipt-print").unbind("click");
            $("#receipt-print").click(myCustomPrintReceiptFunction);
        });
}

function myCustomPrintReceiptFunction()
{
    Navigator.gotoPrintReceiptPage();
}
```

Server API Calls

The JavaScript objects that make the service calls to the POS Server are accessible by the `window.serviceManager` object. The objects are returned by calling `getService` with the name of the service. The following table outlines the different services:

Service Name	Function
account	Gift cards
auth	Login, logoff
itemlookup	Item lookup
lineitem	Add line item, item discounts, item quantity
posversion	Mobile POS Server version
receipt	Email and print receipt
register	Create session, close till
tenderline	Tender
transaction	Cancel, suspend, transaction discounts

Custom Service with Unaltered Method Signature

To use a custom service that uses methods with the same signature, use the `overrideService` method to install a new service object. Example 2–25 shows how to hook up a custom authorization service:

Example 2–25 Hooking Up a Custom Authorization Service

```
function MyCustomAuthService ()
{
    this.login = function(userID, password, hostport, version, deviceId, locale,
        timeoutSeconds, successCallback, errorCallback)
    {
        ...
    }

function onCustomMPOSInitialized()
{
    window.serviceManager.overrideService("auth", new MyCustomAuthService());
}
```

Custom Service with Modified Method Signature

If the method signature changes, ensure that all calls of the method are updated as well. This may necessitate replacing functions in both UI pages and business objects.

For example, if another parameter named `secret` is needed for logging in, you need to replace the following:

- The function that handles the login button click, `loginButtonClick`, to get the secret from the input field.
- The function that calls `MobilePOSUser.login`, `loginWithUserIDPassword`, to pass in the secret parameter.
- `MobilePOSUser.login`, to accept the secret parameter and pass it into the `login` method of the authorization service.

Custom Service with Modified Data Returned from the POS Server

Mobile POS is tolerant of extra information being returned from the server. The difficulty in processing a customized response therefore lies in being able to access the returned information.

With some services, such as `transaction` and `lineitem`, the entire transaction is returned and a reference is kept in `Transaction._transactionData`. In this case, accessing the custom information can be done by adding an access method to `Transaction` for that field.

However, in cases such as login, the entire response from the POS Server is not retained. In this case, a custom function that replaces the default needs to be written. All of the original functionality of the replaced code must be duplicated for the Mobile POS UI to function correctly.

For example, suppose a modification to the login API on the server returns the employee ID in the reply back to the UI. Since the login method of `MobilePOSUser` is the method that processes the login response, it needs to be replaced to save the employee ID, as shown in Example 2–26:

Example 2–26 Modification to the Login Method

```
function myCustomUserLogin(userId, password, successCallback, failureCallback,
    errorCallback)
{
    // All of the code from MobilePOSUser.login
    ...
    this.employeeID = response.employeeID;
}

function onCustomMPOSInitialized()
{
    MobilePOSUser.login = myCustomUserLogin;
}
```

Configurable Settings

The defaults object defined in `mobilepos.js` contains the default settings for some of the functionality of the Mobile POS UI. It does not have corresponding configuration settings in the POS server, including things such as battery level checks and whether to aggregate basket items with identical item numbers. To override these default settings, use the `MobilePOSConfig` setters defined in `config.js`.

To change the minimum battery level allowed to start applying payments, call `setMinBatteryLevelPayment` with the new value as shown in Example 2–27:

Example 2–27 Changing the Minimum Battery Level

```
function onCustomMPOSInitialized()
{
    MobilePOSConfig.setMinBatteryLevelPayment(0.02);
}
```

Changing the Security Mechanism

Generally, changing the security mechanism involves writing code in `customize.js` to override the logic in the following:

- `loginPage.js`: add or modify the input fields. See "Inserting an Input Field on a Screen".
- `loginErrors.js`: handle any additional errors that can be returned by the security mechanism.

- `user.js`: receive the new input fields from the UI layer and pass on to `AuthService`.
- `authService.js`: make the authorization service call. See "Server API Calls".

New Hardware

This section has information on adding new hardware.

Supporting New iOS Devices

Customizing Mobile POS to support a new iOS device would most likely involve only style changes in `customize.css`, provided the following libraries support the new iOS device:

- PhoneGap
- jQuery
- jQuery Mobile
- VeriFone Framework

However, if an upgrade to any of these libraries is required to support the new iOS device, additional Objective C or JavaScript programming may be needed to account for the differences between the library version used in the release and the updated library.

Supporting Alternate Mobile OS Devices

Customizing Mobile POS to support a non-iOS device involves significant work. In addition to styling changes to support different screen dimensions, the following PhoneGap plug-ins need to be created on the new mobile OS.

PhoneGap Plug-in	Description
`DeviceInfo`	Methods for getting information from the device, such as battery level.
`Internationalization`	Method for getting the device's configured language and locale.
`Log`	Methods for writing to the application log file.
`NetworkActivity`	Methods for turning the network activity indicator on and off.
`Sled`	Methods for working with the sled attached to the device.
`UserDefaults`	Methods for getting the login timeout and server setting.

Source code for the iPod Touch versions of these plug-ins is not available. The methods to implement in each plug-in can be determined by inspecting the JavaScript code that calls the plug-in.

Supporting Tablets

This section provides information on using tablets with the Mobile POS UI.

iPad

The Mobile POS UI runs on an iPad out of the box. However, it does not make any attempt to use the extra screen space in a meaningful manner. Instead, existing screen elements are simply stretched or centered.

To truly make good use of the iPad, new screens need to be created that present additional information on each screen, such as combining the item detail screen with the basket screen.

Non-iOS Tablet

To customize the Mobile POS UI to run on a non-iOS tablet, see "Supporting Alternate Mobile OS Devices".

Supporting Alternate Sleds

To support a sled other than the VeriFone VX600, the PhoneGap plug-in that handles the communication between the JavaScript portion of the application and the sled must be replaced. Guidance on how to write PhoneGap plug-ins is available on the PhoneGap web site.

The replacement plug-in must implement all of the methods implemented by the VeriFone VX600 PhoneGap plug-in. The following table lists the methods:

Method	Description
`activateScanner`	Turns the scanner on.
`activateSwipeDetection`	Activates the magnetic strip reader.
`assignPageCallback`	Specify JavaScript function to call in response to a barcode scan.
`assignSwipeCallback`	Specify JavaScript function to call in response to a magnetic stripe read.
`clearLogFile`	Clears (removes) the physical log file from the device.
`deactivateScanner`	Deactivates the bar code scanner.
`deactivateSwipeDetection`	Deactivates the magnetic stripe reader.
`initializeScanner`	Activates the bar code scanner.
`queryBatteryLevel`	Returns the sled battery level.
`queryDeviceState`	Returns `PGCommandStatus_OK` if the device is initialized and ready to scan barcodes, otherwise returns `PGCommandStatus_ERROR`.

`querySledInfo`	Returns sled information in JSON format:
	`{`
	`"Serial Number": "value",`
	`"Model Number": "value",`
	`"Manufacturer": "value",`
	`"Hardware Revision": "value",`
	`"Firmware Revision": "value"`
	`}`

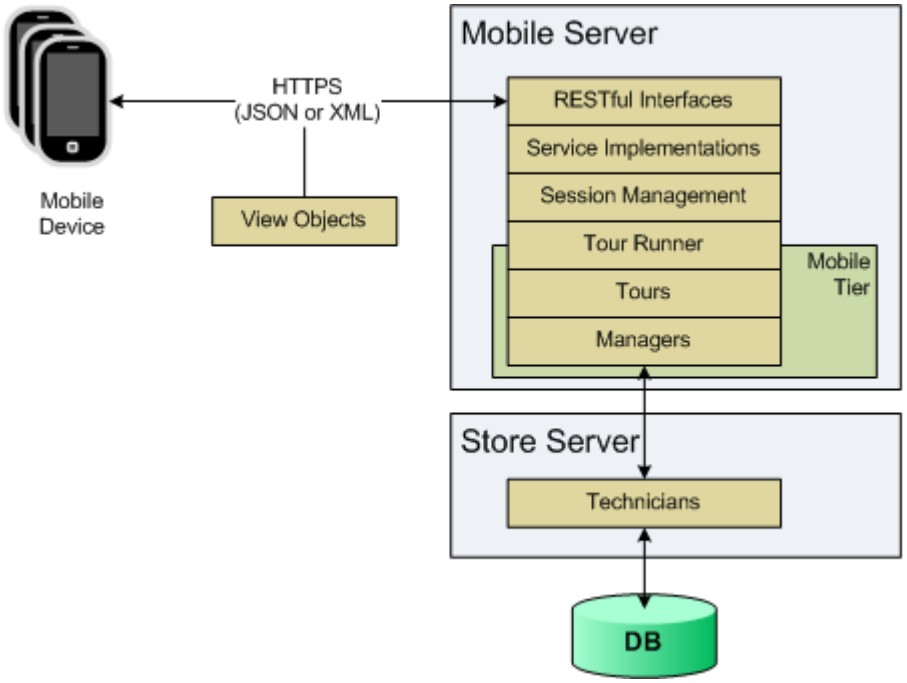
Specifics on the inputs to and outputs from the plug-in methods can be deduced by inspecting the JavaScript code that calls the plug-ins. Search the JavaScript code for `window.plugins.sled.` plus the name of the method from the preceding table. For example, search for `window.plugins.sled.querySledInfo`.

3

Mobile POS Server Overview

The Mobile POS server application provides a REST-based interface into the Oracle Retail Point-of-Service application. The application is comprised of six high-level component areas, as depicted in Figure 3–1.

Figure 3–1 Mobile POS Components



The components are deployed as a war file into a WebLogic application server that has been configured to support REST through the Jersey libraries. The war file contains a mobile POS tier that provides access to the tour engine and the tours used to support the programming API.

Solution Components

This section describes the components shown in Figure 3–1.

REST Interfaces and Service Implementations

The API is exposed through a REST-like API, allowing HTTP clients to easily access the services. These services are also available through a Java API that sits behind the REST interface. The service interface and implementation classes can be found in the `oracle.retail.stores.mobilepos.web.service` package and its sub-packages. The services are organized into five service areas:

- About for version information APIs
- Auth for authentication APIs
- Item for item lookup APIs
- Register for register APIs
- Transaction for retail transaction APIs

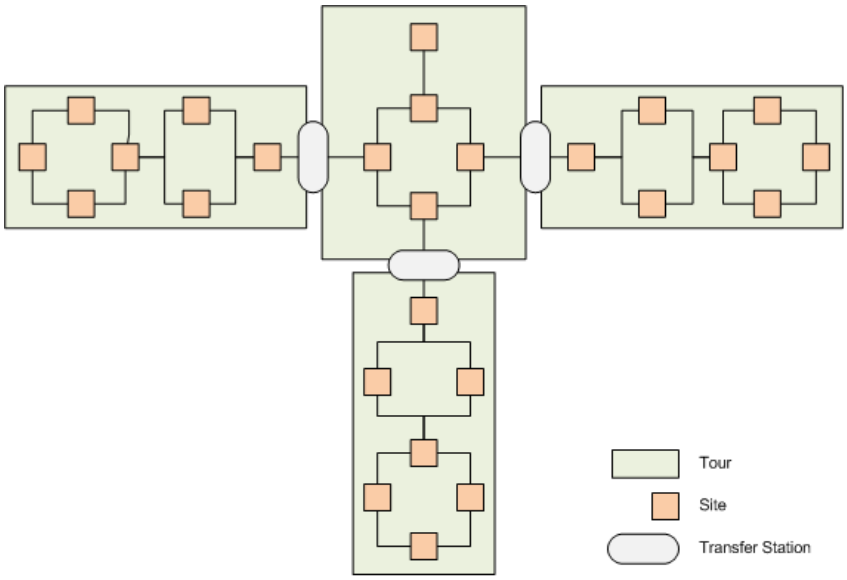
View Objects

The REST layer returns the results of each method call through a Status instance, marshaled to either JSON or XML, depending on the MIME-TYPE requested by the client. The view objects carry the data requested by the caller, as well as status information, in the form of error codes and their associated messages. These classes are in the `oracle.retail.stores.mobilepos.status` package and its sub-packages.

Access to Tours

The services use a mobile POS tier and the TourRunnerIfc to access POS tour code. A traditional POS tour map is a single connected graph of sites, organized into tours connected by transfer stations. See Figure 3–2.

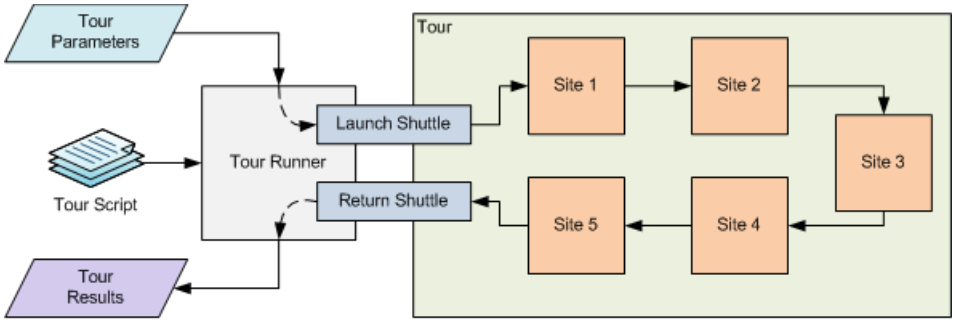
Figure 3–2 Traditional POS Tour Map



Once the POS client tier is loaded, the initial tour is started and the thread of execution never leaves the tour engine. It transitions from site to site as letters are mailed and waits at each site until the next letter is processed.

Unlike traditional POS tours, Mobile POS service calls run a single branch of a tour, not the entire graph.

Figure 3–3 Tour Access



The following describes Figure 3–3:

1. The Tour Runner acts as an entry point into a tour script, mimicking a transfer station to provide shuttle behavior for the tour.
2. Each service API method runs a configured tour script and retrieves the tour results through configured shuttles. Input parameters for the service are packaged up into Tour Parameters and passed into the Tour Runner.
3. The Tour Runner then loads the specified tour from the Tour Script and uses the Launch Shuttle to convert the Tour Parameter into the cargo used by the Tour.
4. Once the tour completes its processing, the Return Shuttle is called with the populated cargo. The Return Shuttle converts the cargo into the Tour Results object. Those results are then returned to the calling service API.

The tours executed by the API are the same tours defined in a typical ORPOS client, some with minor modifications to support head-less operation. As such, the tours make use of existing Manager/Technician pairs and access the store server just like the register-base POS client. The classes used to launch and return from these tours are located in the `oracle.retail.stores.mobilepos.tours` package and its sub-packages. The mobile tier classes needed to run these tours are found in the `oracle.retail.stores.mobilepos.manager.tier` package.

Application State Management

This API is designed to keep the client application as thin as possible. To aid in that goal, the Mobile POS server tracks the state of key objects within the servlet container. Each authenticated user has a separate session space, and, within that space, the following information is tracked:

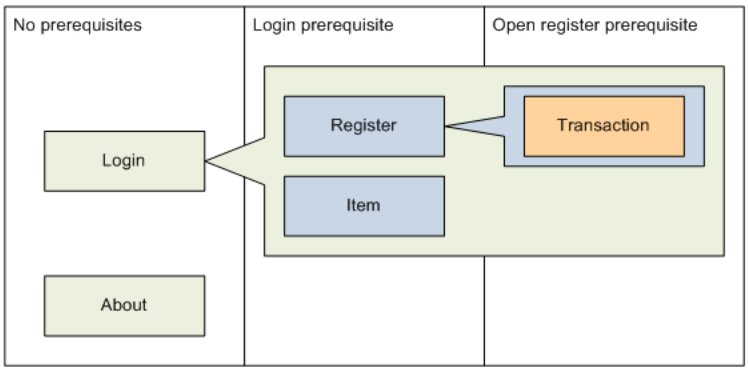
- Register status
- Store status
- Authenticated operator
- Current retail transaction
- Most recently completed retail transaction, if available

Unlike traditional REST APIs, this API does not require that the client pass object state into the API. Instead, the API accepts parameters and returns the affected object's state to the caller. Once a user's session is terminated, either by logging out or due to

inactivity, all objects in the session are discarded. When this happens, any active transactions are cancelled.

Because the API is stateful, there is a dependency order to some of the API calls.

Figure 3–4 Service Area Dependencies



These dependencies follow the service area boundaries closely. The Login and About service areas have no prerequisites. Their APIs can be called at any time. The Register and Item service areas require that the login API be called before accessing their APIs. And finally, the Transaction service APIs require that the open register API be called before trying to work with transactions.

Device Profiles

The Mobile POS API uses the concept of a device profile. These profiles include the mapping of a unique vendor ID to a store and register ID, store currency information, relevant parameter values, and other needed configuration information. The profiles are managed through Spring and can be found in the `DeviceContext.xml` file.

Note: The register IDs used in the `DeviceContext.xml` file are assumed to be contiguous. When you are assigning IDs to the mobile registers, do not intersperse fixed register IDs with mobile register IDs.

Note: All the mobile devices running on the same Mobile POS server share the same parameter values. If any parameter values are changed, all the mobile devices get the same updates.

Authentication and Authorization

The Mobile POS API uses two levels of authentication due to its unique deployment configuration. As a web application, the servlet container authenticates the request using the same custom plug-in the other POS Suite web applications use. Once the request is authenticated, the stateful services require that the user call the login API before several other API calls. This second authentication uses the existing POS login tour to establish the user in the session.

Authorization is handled by the tours, just as in the register-based POS client. In order to provide the credentials to the POS tour, HTTP Basic Authentication must be used when accessing the login API. All service APIs are protected resources, so

authenticated sessions are required for access, but the login API requires HTTP Basic Authentication to succeed.

Due to the multi-client nature of the Mobile POS application, security access is managed by register. As a result, you must provide the register number as the `appID` to any cargo used by a Mobile POS tour. This ID maps the user credential to the active mobile session. Doing this insures that the proper user credential is checked by any Site needing to verify access permissions. Additionally, the AbstractRegisterWebService provides the checkAccess (Status status, int roleFunctionID, String registerID) API for checking permissions at the service layer.

4

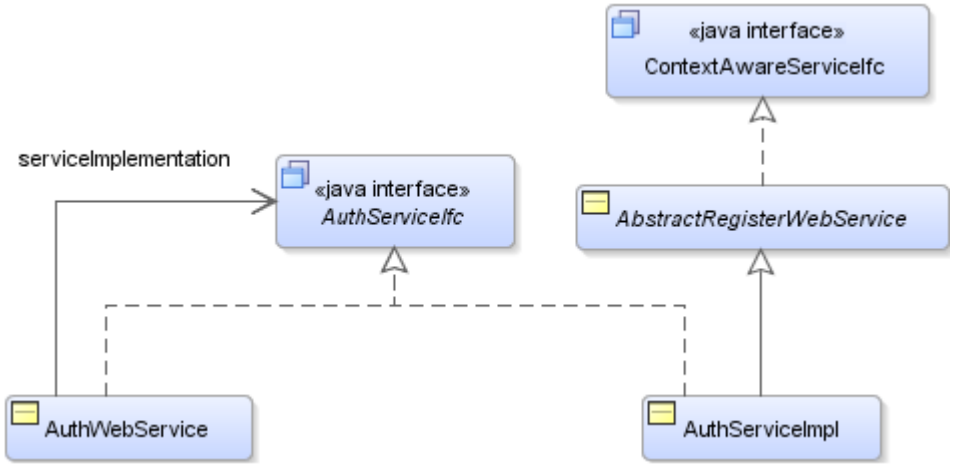
Mobile POS Frameworks

There are several frameworks at work in the Mobile POS server application. These include the service REST service layer, the tour runner framework, and the view object hierarchy. Each is discussed in detail.

REST Service Layer

The REST service layer is composed of a REST-annotated shell class backed by a service implementation class. Each service class follows the same pattern, as seen in the `AuthWebService` example in Figure 4–1.

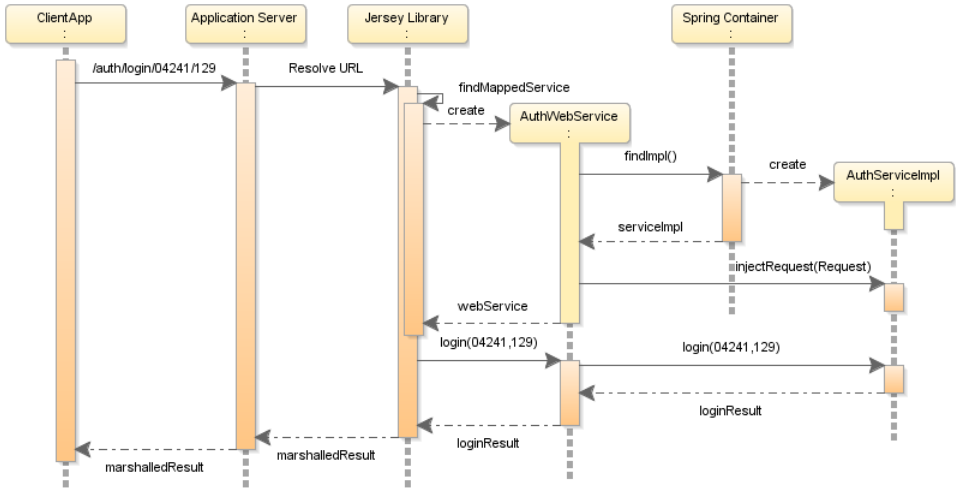
Figure 4–1 Service Framework



The WebService classes are tagged with REST annotations that the Jersey libraries use to marshal messages in and out of the API. Since only concrete classes can be annotated, the implementation of the services is delegated to a paired `ServiceImpl` class. The two are kept in API synchronization by implementing the same `ServiceIfc`. These pairings are defined in the `ServiceContext.xml` file to allow for extension and customization. The out-of-the-box `ServiceImpl` classes all implement the base `AbstractRegisterWebService` class that provides common session management functionality, along with other shared behaviors. The `ContextAwareServiceIfc` is used to pass the request context into the delegated `ServiceImpl` instance since the Jersey libraries only inject the request context into the annotated class. All out-of-the-box `ServiceImpl` classes implement this interface.

Figure 4–2 shows the lifecycle of a Mobile POS service call.

Figure 4–2 Service Call Sequence



When the client application calls a service API, the Jersey library's registered servlet intercepts the call and directs it to the API mapped to the URL requested, extracting parameters where necessary. Once the class and method are identified, the Jersey library instantiates the class and injects the request information into the new instance. The constructors of the provided REST classes look up their paired implementation class from the Spring container and inject the provided request context into them, provided they implement the `ContextAwareServiceIfc`.

By default, the `ServiceImpl` classes are singletons, so they are only instantiated once and simply retrieved on each subsequent call to the same service area. After Jersey has successfully instantiated the target REST class, it executes the mapped method. The REST class simply takes those parameters and passes them to the paired class' matching method.

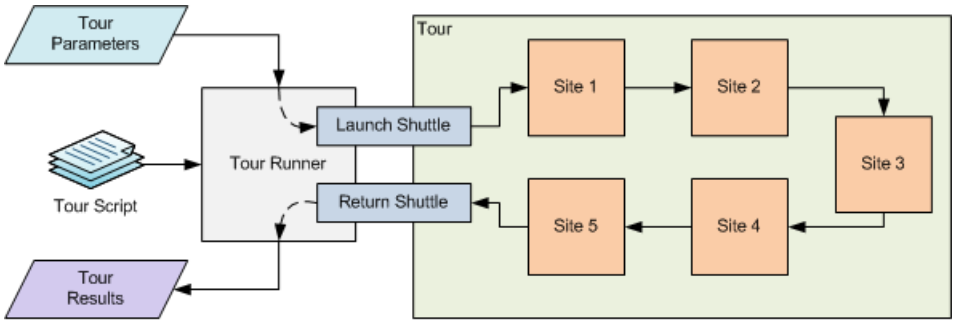
Tour Runner and Tour Access

Most of the methods in the `ServiceImpl` classes are backed by a POS tour. The only four that are not are:

- `AboutServiceImpl.getVersion()`
- `AuthServiceImpl.logout()`
- `RegisterServiceImpl.getRegisterProfile()`
- `SaleTransactionServiceImpl.getSaleTransaction()`

Inside the remaining `ServiceImpl` method, the parameters are validated and then used to populate the `TourParameters` instance appropriate for the tour to be executed.

Figure 4–3 Tour Runner



Once populated, the tour parameters instance is sent into the configured tour runner instance and the mapped tour is executed. The tour runner configuration is done through the `ServiceContext.xml` file, where you can configure what tour script to execute and which launch and return shuttles to use. Example 4–1 is an example of the Item Delete API configuration:

Example 4–1 Tour Runner Configuration

```
<bean id="service_ItemDeleteTourRunner"
      class="oracle.retail.stores.mobilepos.tours.TourRunner"
      lazy-init="true" singleton="false">
  <property name="tourScript"
      value="classpath://oracle/retail/stores/…/delete/itemdelete.xml"/>
  <property name="launchShuttleClass"
      value="oracle.retail.stores.….delete.ItemDeleteLaunchShuttle"/>
  <property name="returnShuttleClass"
      value="oracle.retail.stores.….delete.ItemDeleteReturnShuttle"/>
</bean>
```

For more information on POS tours and their development and extension model, see *Oracle Retail POS Suite Implementation Guide, Volume 2 - Extension Solutions*.

Another aspect of the TourRunner that can be configured is the UI manager used while the tour is being executed. The out-of-the-box Mobile POS uses a customized UI manager called the `InstructibleUIManager`. This UI manager allows the tours to run to completion without displaying any screens. Traditional POS tours have a mixture of business logic and view logic. In order to reuse these tours, the Mobile POS API needs a way to avoid showing screens, as this blocks the tour thread preventing the tour from completing.

Example 4–2 InstructibleUIManager Configuration

```
<bean id="service_ItemModifySalesAssocTourRunner"
      class="oracle.retail.stores.mobilepos.tours.TourRunner"
      lazy-init="true" singleton="false">
  <property name="tourScript"
            value="classpath://oracle/retail/stores/…/modifyitem.xml"/>
  <property name="launchShuttleClass"
            value="oracle.retail.stores.….modify.ItemModifyLaunchShuttle"/>
  <property name="returnShuttleClass"
            value="oracle.retail.stores.….modify.ItemModifyReturnShuttle"/>
  <property name="managerData">
     <map>
       <entry key="UIManager" value-ref="service_ManagerData" />
     </map>
```

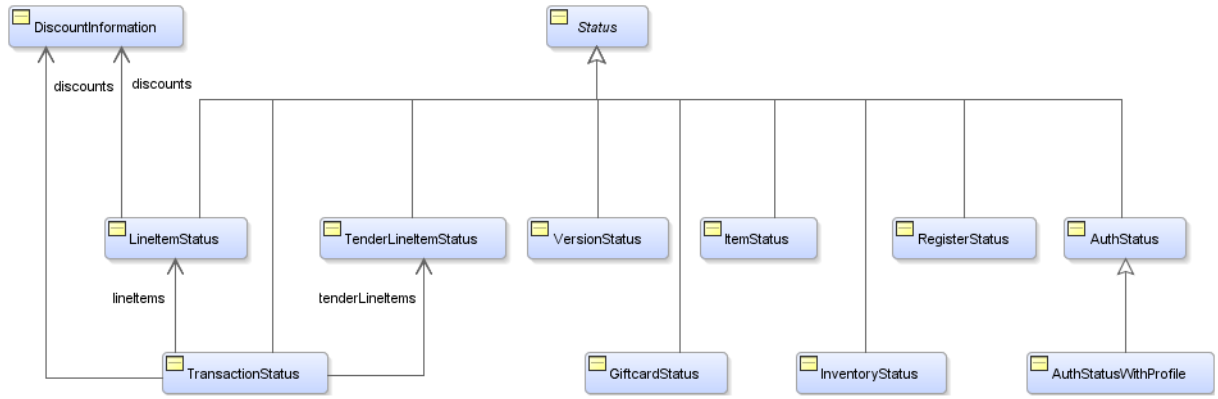
```
  </property>
  <property name="siteLetters">
      <map>
         <entry key="ModifyItemMenu" value="SalesAssociate" />
         <entry key="ModifyItemSalesAssociate" value="Next" />
      </map>
  </property>
</bean>
```

The `InstructibleUIManager` allows the tour runner to inject the letters that need to be sent when a show screen event is fired for a particular site. In most cases, this is simply a `Yes` or `Continue` type letter on a dialog. However, some situations require a little more information. For those situations, an *instruction* can be configured that the `InstructibleUIManager` will use to decide how to handle that particular screen.

Status and Tour Parameter Object Frameworks

The view objects are used to return information to the calling application. They are collected into the `Status` object hierarchy.

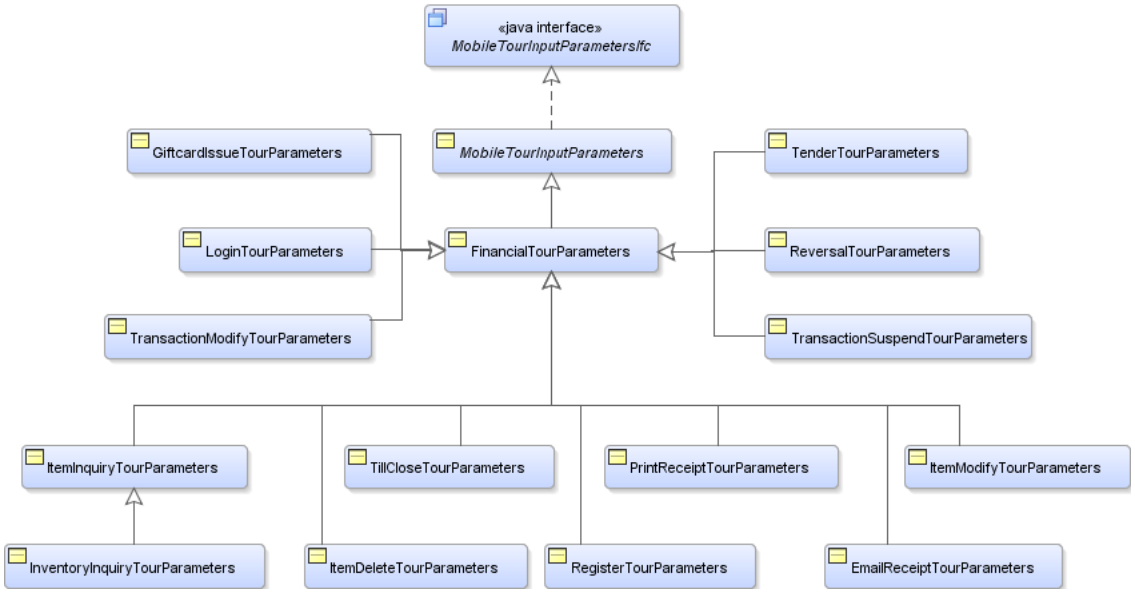
Figure 4–4 Status Object Hierarchy



The `Status` base class provides an error collection that allows the API to return error codes and messages to the calling application. Each `Status` sub-class represents the state of a particular retail object: Register, Item, Transaction, and so on.

The Tour Parameter objects provide the inputs to the tours run by the service API.

Figure 4–5 Tour Parameter Object Hierarchy



The inputs to the service API are collected into various tour parameter objects and passed into the tour associated with mapped tour runner. The API instantiates these classes using the `MobilePOSObjectFactoryIfc`. This allows the objects to be extended and customized in a manner similar to POS domain objects.

5

Mobile POS Extension Guidelines

This chapter provides some examples of common extensions and customizations to the Mobile POS server application. This includes how to override an existing service call, add a new service call, configure the `InstructibleUIManager`, and add a new service area. The screenshots for Eclipse are found in Appendix C.

Customize an Existing Service API

There are several points available for customizing an existing service call's behavior. The following are described in this section:

- Override a service method
- Customize the Tour Runner
- Customize the target Tour

Service Method Override

Replacing or extending the behavior of the base service method requires the implementation of a new Java class and the update of a configuration file. For this case, assume that you want to add some additional validation logic to the results of the existing `AuthWebService login` API.

1. Create a new Eclipse project or update an existing one. For detailed instructions, see "Eclipse Project Creation" in Chapter 6.
2. Create a new class by right-clicking on Java Resources under the mobilepos project. The New Java Class dialog appears. Provide a Java package and class name. Set the Superclass to the following:

   ```
   oracle.retail.stores.mobilepos.web.service.auth.AuthServiceImpl
   ```

   See Figure C–1.
3. Press Finish. The new class is generated similar to the following example:

   ```
   package oracle.retail.stores.mobilepos.example;

   import oracle.retail.stores.mobilepos.web.service.auth.AuthServiceImpl;

   public class AuthServiceLoginExtension extends AuthServiceImpl
   {

   }
   ```

4. Implement the extension by overriding the target method. See the following example:

```
@Override
public AuthStatusWithProfile login(String hardwareID, String locale)
{
   AuthStatusWithProfile result = super.login(hardwareID, locale);

   // Do some additional validation here

   ContextError someError =
           new ContextError("NEW_ERROR_CODE", "Some error message.");

   result.addError(someError);
   return result;
}
```

5. Update the `ServiceContext.xml` file to set the custom class as the implementation for the `AuthWebService`. See the following example:

```
<bean id="service_AuthService"
      class="oracle.retail.stores.mobilepos.example.AuthServiceLoginExtension"
      lazy-init="true"
      singleton="true"/>
```

6. Build and deploy the war file to test.

Customize the Tour Runner

You can also change an existing API's behavior by customizing the Tour Runner configuration to execute a tour differently, execute a different tour, or handle additional tour parameters by changing the launch and return shuttles.

Tour Runner Configuration

Start by examining the Tour Runner configuration script. The numbers in bold in Example 5–1 refer to major elements of the configuration that are described following the example.

Example 5–1 Tour Runner Configuration

```
<bean id="service_ItemInquiryTourRunner"
           class="oracle.retail.stores.mobilepos.tours.TourRunner"
           lazy-init="true" singleton="false">
    <property name="tourScript" (1)
        value="classpath://oracle/retail/stores/.../iteminquiry/ItemInquiry.xml"/>
    <property name="launchShuttleClass"  (2)
        value="oracle.retail.stores.....inquiry.ItemInquiryLaunchShuttle"/>
    <property name="returnShuttleClass"  (3)
        value="oracle.retail.stores.....inquiry.ItemInquiryReturnShuttle"/>
    <property name="managerData"> (4)
        <map>
            <entry key="UIManager" value-ref="service_ManagerData" />
        </map>
    </property>
    <property name="siteLetters"> (5)
        <map>
            <entry key="ShowItem" value="Next" />
            <entry key="ShowItemList" value="Next" />
        </map>
```

```
    </property>
</bean>
```

The following are the major elements of the Tour Runner configuration

1. **Tour script:** Fully qualified file name of the tour XML defining the script to run, generally looked up on the CLASSPATH.
2. **Launch shuttle class:** Fully qualified class name of the shuttle to use when the tour launches.
3. **Return shuttle class:** Fully qualified class name of the shuttle to use when the tour exits.
4. **Manager data (optional):** Reference to the UI Manager to use while running the tour.
5. **Site letters (optional):** Collection of letters to mail from a particular site keyed by the situation that mails them.

The first three properties are required for a tour runner to execute successfully. Properties four and five are optional and generally are used as a pair. If you configure a UI Manager, you generally do so to enable letter manipulation.

InstructibleUIManager

As mentioned earlier, the *InstructibleUIManager* allows the tour runner to inject the letters that need to be sent when a show screen event is fired for a particular site. This behavior is needed since the Mobile POS server runs in a request-response mode instead of an interactive mode like the traditional POS. This manager extends the `POSUIManager` and overrides the showScreen behavior. When a POS tour site requests a screen or dialog to be displayed, the `InstructibleUIManager` intercepts the call and checks to see if any letters have been mapped for that particular site.

Example 5–2 Letters

```
<bean id="service_ItemInquiryTourRunner" …>
     ….
    <property name="siteLetters">
        <map>
            <entry key="ShowItem" value="Next" />
            <entry key="ShowItemList" value="Next" />
        </map>
    </property>
</bean>
```

If a letter has been mapped for the site making the request, the letter is mailed and the tour continues forward. If no letter is mapped, the default implementation throws an error by mailing the `UnknownException` letter, and the tour exits. In addition to providing letters for sites, the `InstructibleUIManager` can also accept instructions for a particular site. The default implementation supports two instructions:

- `Instruction.DoNotMail`
- `Instruction.MailWithErrors:<Letter Name>`

The first is used in cases where the POS tour will show a screen that merely shows some status and moves on without a letter getting mailed. In those cases, the value `Instruction.DoNotMail` should be provided. When the `InstructibleUIManager` receives this mapping, the `showScreen` call is exited without any letter being mailed.

Note: Caution should be used when applying this instruction because the tour could hang if the site in question really should mail a letter and no letter is mailed.

The second instruction is used when an error is encountered, but you want the tour to continue on instead of exiting. The InstructibleUIManager will parse out the letter name and mail it while logging the error into the cargo. This allows tours that encounter recoverable errors to report the error back to the caller while continuing the tour.

Error Handling in the Mobile POS Server Unlike the traditional POS, the Mobile POS server is designed to handle multiple registers running in the same Java virtual machine. As such, a site cannot assume a register for a particular action. The associated register must be provided. This is particularly important for error handling. To assist with this situation, the Mobile POS server provides an error code context to place errors into. The `MobileErrorCodes` and `ContextError` classes provide APIs for recording the errors into a specific error context.

`MobileErrorCodes` is a utility class that allows the implementer to simply provide an error code to record an error. The error message is pulled from the resource bundle based on a set naming standard (MobileApi.*errorCode*).

ContextError collects errors based on context and allows the service layer to retrieve and return them to the caller. This context is generally the register number requesting the action. Access to the context is provided by cargo classes that implement the `MappableContextIfc` interface. If the cargo does not implement this interface, the cargo class is reflectively scanned for no-argument methods that return a `RegisterIfc` so it can glean the context from the cargo. Once the context is established, the error code and any associated message are added to the `ContextError` collection keys by the given context.

Tour Runner Inputs and Outputs

The Tour Runner framework also allows you to customize the parameters provided to a tour and the results returned from a tour. The `MobileTourInputParameters` hierarchy of objects provides the input data for all default tour runners. These objects carry the data that is provided to each underlying tour as cargo attributes. The Status hierarchy of objects provides the output data and errors for all default tour runners. As discussed in the Status and Tour Parameter Object Frameworks section, these objects can be customized by implementing a custom object factory that extends the `MobilePOSObjectFactory`. For information on how these objects can be customized, see the *Oracle Retail POS Suite Implementation Guide, Volume 2 - Extension Solutions*.

Customize Tour

The tours executed by the tour runner are configured and extended just like existing POS tours. They are organized in a tour map XML file and referenced by XML file name. Any tour run by the Mobile POS server can be customized using the same mechanisms used to customize POS tours. There are two things you must keep in mind when reusing an existing POS tour in the Mobile POS server or implementing a new tour for the Mobile POS server:

- The tours must be thread safe. Sites loaded into the foundation layer are treated as singletons. As such, each request thread can run through the site simultaneously, so all tour code must be thread safe.

- The tour will need to be headless. Any existing tour will need to be analyzed to ensure all screen interactions are accounted for in the `InstructibleUIManager` letter mappings and all error conditions are handled properly. Any new tours should be written in such a way as to avoid the need for intermittent screen interaction.

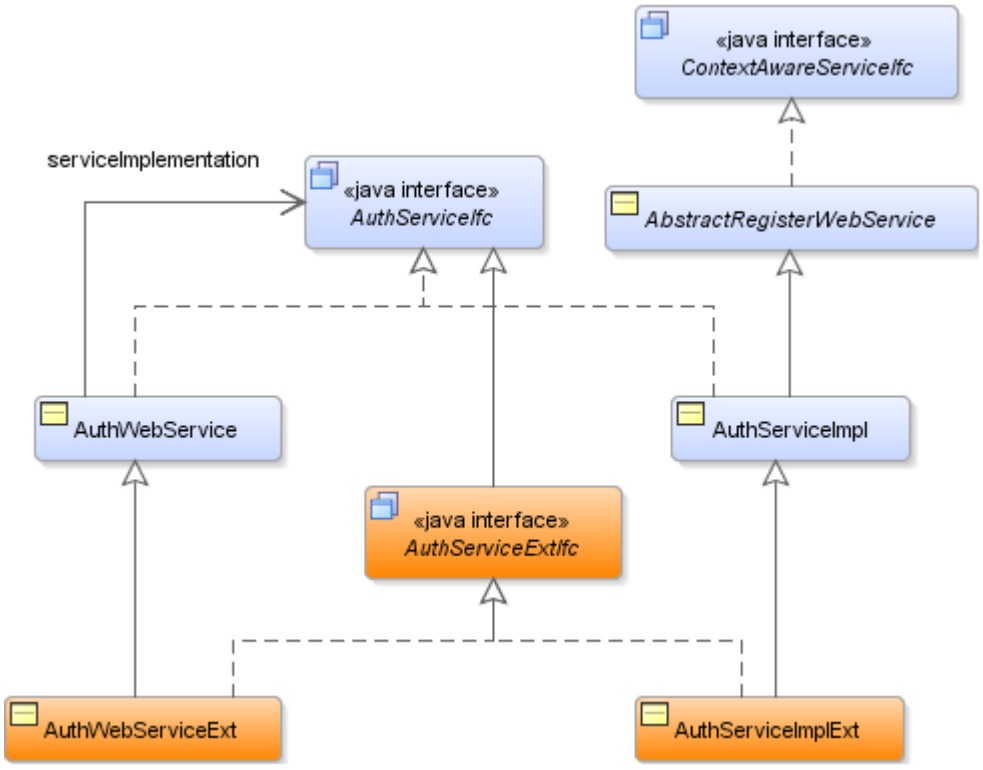
For information on extending POS tours, see the *Oracle Retail POS Suite Implementation Guide, Volume 2 - Extension Solutions*.

Extend an Existing Service Area

When adding new functionality to an existing service area, you should follow the patterns established for these services:

- Extend the service framework classes
  - Service interface
  - REST shell
  - Service implementation
- Configure a new tour runner
- Create tour input and output objects

Figure 5–1 Service Extension Model



The naming used here is an example. Follow your local naming conventions for the extensions. The new APIs should be declared on the interface and implemented on the two concrete classes. Be sure to annotate the `WebServiceExt` class and methods with REST annotations.

Example 5–3 Service Extension

```
@Path("auth/1.1")
public class AuthWebServiceExt extends AuthWebService implements AuthServiceExtIfc
{
    @Path("/login/extension/{storeID}/{registerID}")
    @GET
    @Produces({MediaType.APPLICATION_JSON,MediaType.APPLICATION_XML})
    public AuthStatusExt loginExtension(@PathParam("storeID")    String storeID,
                            @PathParam("registerID") String registerID,
                            @QueryParam("locale")    String locale)
    {
        return ((AuthServiceImplExt)serviceImplementation)
                                .loginExtension(storeID, registerID, locale);
    }
}
```

If the functionality of the new API requires a tour, identify an existing tour that performs the action or write a new one. You then need to configure a tour runner to execute that tour. For input into the tour, you may need a new Tour Parameters object. Again, use the framework provided and extend where appropriate. The same applies for the tour results. A new Status object may be required, and it too should take advantage of the Status hierarchy and extend where appropriate. The body of the new service API would look something like this:

Example 5–4 New Service API

```
// Validate the parameters
...

// Load the status object
YourStatus status = MobilePOSObjectFactory.getInstance().getYourStatus();

// Build tour parameters
YourTourParameters parms =
MobilePOSObjectFactory.getInstance().getYourTourParameters();

parms.setParameterValue1(parameter1);
parms.setParameterValue2(parameter2);

// launch tour
try
{
    MobileTourResultIfc result = runTour(parms);
    if (result.getTourResult() instanceof YourStatus)
    {
status = (YourStatus)result.getTourResult();
return status;
    }

    logger.warn("Invalid result from tour: " + result);
    return null;
}
catch (Exception e)
{
    return MobilePOSObjectFactory.getInstance().getYourStatus(toErrorCode(e));
}
```

Once all of the classes are in place, update the `ServiceContext.xml` file as described in the previous section to configure the new tour runner and update the implementation class for the extended service.

Adding a New Service Area

Much like adding a new API to an existing service area, adding a new service area involves following the frameworks available. Take advantage of the base classes described in the previous sections to bootstrap your efforts. Be sure to distinguish your services by placing them on REST paths that do not conflict with the shipped services.

Customizing Authentication

The default authentication scheme uses the store database to read employee credentials. If you want to use a different repository for authentication, you need to update both authentication mechanisms described in the overview. For the application server plug-in, you need to provide a new implementation that uses the other repository. If you have already developed such a plug-in for any of the other POS Suite web applications, you should be able to reuse it. The only additional requirement the Mobile Server has over the other POS Suite web applications is support for the `VersionableApplicationProvider` interface. This interface is required to support the import of the Jersey libraries into the Mobile Server application server domain.

You also need a custom POS tour to access the alternate repository. Again, if POS has been customized to use this repository, you should be able to reuse that custom tour from the Mobile Server. You just need to update the login tour runner configuration (`service_LoginTourRunner` in ServiceContext.xml) to point to this alternate tour.

Working with Device Profiles

Device profiles provide a way to configure the handheld devices as registers and provide them with settings that are shared across the devices. The device profiles are stored in the `DeviceContext.xml` file under the `device_MobileRegisterProfileConfiguration` bean key. The profile configuration consists of two main areas: device-to-register mappings and device settings.

Device Mappings

The device mapping section allows you to map a unique hardware identifier to a register in a store.

Example 5–5 Device Mappings

```
<property name="deviceMappings">
<map>
    <entry key="Hardware ID 1">
<bean class="oracle.…..RegisterProfileConfiguration.StoreRegisterPair">
    <property name="storeID" value="04241" />
    <property name="registerID" value="100" />
</bean>
    </entry>
    <entry key="Hardware ID 2">
<bean class="oracle….RegisterProfileConfiguration.StoreRegisterPair">
    <property name="storeID" value="04241" />
```

```
    <property name="registerID" value="101" />
</bean>
    </entry>
</map>
</property>
```

The default implementation uses the UVID for iOS devices, but any unique ID can be configured. The installer allows for five devices to be configured, but there is no limit to the number of devices that can be in this map.

> **Note:** The register IDs are assumed to be a contiguous set, so it is best to keep the mobile register IDs separate from the traditional POS register IDs.

Device Settings

The second set of information available in the device profile is configuration settings. These settings apply to all devices using a device profile. There are three types of setting supported: configuration settings, POS parameters, and reason codes.

Configuration Settings

This set of key/value pairs represents configuration information used by mobile devices that fall outside the other two areas.

Example 5–6 Configuration Settings

```
<property name="configurationSettings">
    <map>
        <entry key="serviceCallTimeout" value="30"/>
        <entry key="serviceCallTenderTimeout" value="300"/>
        <entry key="inactiveWarningTimeout" value="840"/>
        <entry key="inactiveLogoutTimeout" value="900"/>
     </map>
</property>
```

The default settings deal with various timeout values, but any key/value pair can be used. To add a new setting, edit the `DeviceContext.xml` file and insert the new key/value pair. All mobile devices receive the new setting during login.

POS Parameters

This set of values represents the POS parameters returned to the mobile device.

> **Note:** All the mobile devices running on the same Mobile POS server share the same parameter values. If any parameter values are changed, all the mobile devices get the same updates.

Example 5–7 POS Parameters

```
<property name="parameterNames">
    <list>
        <value>CreditCardsAccepted</value>
        <value>GiftCardsAccepted</value>
        <value>TimeoutInactiveWithoutTransaction</value>
        <value>TimeoutInactiveWithTransaction</value>
        <value>IdentifyCashierEveryTransaction</value>
```

```
    </list>
</property>
```

The Mobile POS server resolves each parameter name to its configured value and returns the name/value pair to the mobile device as part of the profile. Only parameters that have a direct affect on the UI should be included in the list. Parameters affecting the POS tours used by the server API need not be included.

Reason Codes

Much like the POS parameter list, this property lists the reason code sets that should be returned to the mobile device as part of the profile.

Example 5–8 Reason Codes

```
<property name="reasonCodeSetNames">
    <list>
        <value>PriceOverrideReasonCodes</value>
        <value>ItemDiscountByAmount</value>
        <value>ItemDiscountByPercentage</value>
        <value>TransactionDiscountByAmount</value>
        <value>TransactionDiscountByPercentage</value>
        <value>TransactionSuspendReasonCodes</value>
    </list>
</property>
```

The Mobile POS server resolves each reason code name to its configured value set and returns it to the mobile device as part of the profile. The default set reflects the reason codes available in the POS sample data. If you customize the reason codes, be aware that this list needs to be updated and the Mobile POS UI needs to reflect those changes.

6

Implementation Environment

This chapter describes how to set up a single-user development environment for Oracle Retail Mobile Point-of-Service extension and customization. The setup enumerates the files, tools, and resources necessary to build and deploy the Mobile POS server application. When you complete the steps in this chapter, you will have a local development workspace with the ability to build and deploy the Mobile POS application.

This chapter assumes that you are using WebLogic Application Server and the Oracle database; together, they form the officially supported platform for the current release of the Mobile POS server. Your development environment may use different tools, and you may develop variations on this procedure. Specific property file settings, in particular, may need to be modified in your environment. For more information about product versions, see the *Oracle Retail Point-of-Service Installation Guide, Volume 1 - Oracle Stack*.

Eclipse Project Creation

The Mobile POS server application uses the Eclipse platform for development. These instructions and screenshots use Eclipse Indigo Java EE IDE. The Eclipse screenshots are shown in Appendix C.

To create the Eclipse project:

1. Copy the `mobilepos.war` file to a temporary location, that is, do not use the deployed war file.
2. Create a new workspace or update an existing workspace.
3. Go to the Workspace Preferences to ensure you have the WebLogic Runtime. See Figure C–2.
4. This release of Mobile Point-of-Service was developed using Oracle WebLogic 10.3.5. Verify you have the correct WebLogic release. See Figure C–3.
5. Import the war file into Eclipse by selecting File and then Import. See Figure C–4.
6. Select WAR file under the Web folder. See Figure C–5.
7. Browse to the temporary location and select the WAR. See Figure C–6.
8. Set the target runtime as appropriate and select **Next**.
9. Select the `oracle.stores.mobilepos-config.jar` as a Web Library to allow for updates to the various Mobile POS configuration files. Additionally, select any language bundle JARs that may need updating. See Figure C–7.
10. Select **Finish**. Eclipse generates the projects. For an example, see Figure C–8.

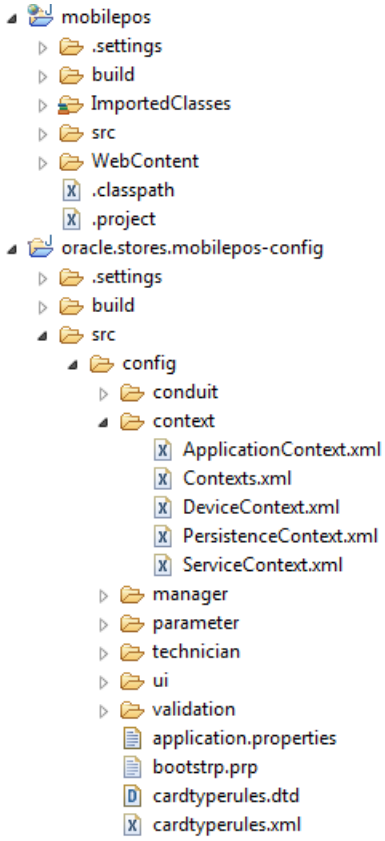
11. Update the mobilepos project to include the Jersey library dependencies by selecting the project properties, Java Build Path, and then **Add Library**. See Figure C–9.
12. Select **Next**.
13. Select **Browse** and then select jsr311.api. See Figure C–10.
14. Select **OK**. The project is ready to use.

Mobile POS Server Deployment Model

There are no build scripts provided for Mobile Point-of-Service. Extension projects should use the Eclipse tools to build and deploy the application WAR to the WebLogic server. If command-line build processes are required, use Eclipse's Ant build file export capabilities to generate the necessary Ant scripts.

Once you have set up your Eclipse workspace, you should see a project that looks something like Figure 6–1:

Figure 6–1 Eclipse Project Structure



The `mobilepos` project is where you develop your customizations, and the `oracle.stores.mobilepos-config` project contains all of the configuration files you generally find with the POS client.

When developing extensions to Mobile POS Server, keep in mind that the JARs delivered with the Mobile Point-of-Service add-on are different from those delivered with POS Suite release 13.4.1. There have been updates made to the POS application,

foundation, and domain that the Mobile POS Server relies on. You cannot use base 13.4.1 POS JARs in place of those delivered with the Mobile POS Server. To take advantage of any customizations you have made to the base POS application, you need to provide those customizations in separate JARs and treat them as third-party libraries to the Mobile POS Server.

7

Internationalization

Internationalization is the process of creating software that can be translated more easily. Changes to the code are not specific to any particular market.

Oracle Retail applications have been internationalized to support multiple languages.

For information on updating Mobile Point-of-Service text for different locales, see "Changing Text" in Chapter 2.

Translation

Translation is the process of interpreting and adapting text from one language into another. Although the code itself is not translated, components of the application that are translated include the following:

- Graphical user interface (GUI)
- Error messages

The following components are not translated:

- Documentation (online help, release notes, installation guide, user guide, operations guide)
- Batch programs and messages
- Log files
- Configuration tools
- Reports
- Demonstration data
- Training materials

The user interface has been translated into the following languages:

- Chinese (Simplified)
- Chinese (Traditional)
- Croatian
- Dutch
- French
- German
- Greek
- Hungarian

- Italian
- Japanese
- Korean
- Polish
- Portuguese (Brazilian)
- Russian
- Spanish
- Swedish
- Turkish

A

Appendix: API URLs

This appendix provides an overview of the API URLs provided by the solution, along with their parameters and return types. For complete details on these APIs, refer to the Mobile Point-of-Service Javadoc.

About API

The following table describes the About API.

URL	HTTP Method	Parameters	Return Type
about/1.0/version			
Retrieve POS version	GET	None	VersionStatus

Auth API

The following table describes the Auth API.

URL	HTTP Method	Parameters	Return Type
auth/1.0/login			
Login using hardware ID	POST	hardwareID locale	AuthStatusWithProfile
auth/1.0/login/{storeID}/{registerID}?locale=			
Login	GET	storeID registerID locale	AuthStatus
auth/1.0/logout/{storeID}/{registerID}			
Logout	GET	storeID registerID	AuthStatus

Item API

The following table describes the Item API.

URL	HTTP Method	Parameters	Return Type
item/1.0/{storeID}/{registerID}/{itemNumber}			

URL	HTTP Method	Parameters	Return Type
Item lookup	GET	storeID registerID itemNumber	ItemStatus
item/1.0/{storeID}/{registerID}/{itemNumber}/inventory			
Item inventory inquiry, local store	GET	storeID registerID itemNumber	InventoryStatus
item/1.0/{storeID}/{registerID}/{itemNumber}/inventory/{otherStore}/{minimumQuantity}			
Item inventory inquiry, buddy store	GET	storeID registerID itemNumber otherStore minimumQuantity	InventoryStatus
item/1.0/{storeID}/{registerID}			
Gift card inquiry	POST	storeID registerID account	GiftCardStatus

Register API

The following table describes the Register API.

URL	HTTP Method	Parameters	Return Type
register/1.0/{storeID}/{registerID}			
Open register	GET	storeID registerID	RegisterStatus
register/1.0/{storeID}/{registerID}			
Close till	DELETE	storeID registerID	RegisterStatus
register/1.0/profile			
Get profile	GET	hardwareID locale	RegisterProfileConfiguration

Transaction API

The following table describes the Transaction API.

URL	HTTP Method	Parameters	Return Type
saletxn/1.0/{storeID}/{registerID}/{txnID}/associate/{associateID}			

URL	HTTP Method	Parameters	Return Type
Update sales associate on the transaction	GET	storeID registerID txnID associateID	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}			
Cancel transaction	DELETE	storeID registerID txnID	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/suspend/{reasonCode}			
Suspend transaction	POST	storeID registerID txnID reasonCode	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{itemID}			
Add line item	GET	storeID registerID txnID itemID	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{lineItemNumber}/quantity/{quantity}			
Update item quantity	GET	storeID registerID txnID lineItemNumber quantity	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{lineItemNumber}/associate/{associateID}			
Update line item associate	GET	storeID registerID txnID lineItemNumber associateID	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{lineItemNumber}/serial/{serialNumber}			
Update line item serial number	GET	storeID registerID txnID lineItemNumber serialNumber	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{lineItemNumber}/priceoverride/{price}/reasoncode/{reasonCode}			

URL	HTTP Method	Parameters	Return Type
Update line item price	GET	storeID registerID txnID lineItemNumber price reasonCode	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{itemID}/amount/{amount}			
Add gift card	POST	storeID registerID txnID lineItemNumber amount account	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{lineItemNumber}/discount/percent/{percentOff}/reasoncode/{reasonCode}			
Update line item discount - percent	GET	storeID registerID txnID lineItemNumber percentOff reasonCode	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{lineItemNumber}/discount/amount/{amountOff}/reasoncode/{reasonCode}			
Update line item discount - amount	GET	storeID registerID txnID lineItemNumber amountOff reasonCode	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{lineItemNumber}/damagediscount/percent/{percentOff}			
Update line item damage discount - percent	GET	storeID registerID txnID lineItemNumber percentOff	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{lineItemNumber}/damagediscount/amount/{amountOff}			
Update line item damage discount - amount	GET	storeID registerID txnID lineItemNumber amountOff	TransactionStatus

URL	HTTP Method	Parameters	Return Type
saletxn/1.0/{storeID}/{registerID}/{txnID}/lineitem/{lineItemNumber}			
Delete line item	DELETE	storeID registerID txnID lineItemNumber	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/accountreentry			
Update gift card account	POST	storeID registerID txnID account	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/discount/percent/{percentOff}/reasoncode/{reasonCode}			
Add transaction discount - percent	GET	storeID registerID txnID percentOff reasonCode	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/discount/amount/{amountOff}/reasoncode/{reasonCode}			
Add transaction discount - amount	GET	storeID registerID txnID amountOff reasonCode	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/tenderlineitem/{tenderAmount}			
Add tender line item	GET	storeID registerID txnID tenderAmount	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/tenderlineitem/{tenderLineNumber}/idverified/{idVerified}			
Update tender line item - custom ID verified	GET	storeID registerID txnID tenderLine Number idVerified	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/tenderlineitem/{tenderLineNumber}			
Void tender line item	DELETE	storeID registerID txnID tenderLine Number	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/callreferral/{tenderAmount}/{approvalCode}			

URL	HTTP Method	Parameters	Return Type
Add call referral tender line item	GET	storeID registerID txnID tenderAmount approvalCode	TransactionStatus
saletxn/1.0/{storeID}/{registerID}			
Get current transaction	GET	storeID registerID	TransactionStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/printreceipt			
Print receipt	GET	storeID registerID txnID	ReceiptStatus
saletxn/1.0/{storeID}/{registerID}/{txnID}/emailreceipt/{emailAddress}			
Email receipt	GET	storeID registerID txnID emailAddress	ReceiptStatus

B

Appendix: API Result Formats

This appendix covers the messages returned from the various API methods, organized by Status object and format.

VersionStatus

JSON:

```
{
   "buildNumber": "120723.0917",
   "versionNumber": "13.4.1"
}
```

XML:

```
<versionStatus>
   <buildNumber>120723.0917</buildNumber>
   <versionNumber>13.4.1</versionNumber>
</versionStatus>
```

AuthStatus

JSON:

```
{
   "roles":    [
            {
         "roleId": "37",
         "roleTitle": "Administration"
      },
            {
         "roleId": "49",
         "roleTitle": "Bank Deposit"
      },
            {
         "roleId": "4",
         "roleTitle": "Void"
      }
   ],
   "userName": "someUser"
}
```

XML:

```
<authStatus>
   <roles>
```

```
      <roleId>37</roleId>
      <roleTitle>Administration</roleTitle>
   </roles>
   <roles>
      <roleId>49</roleId>
      <roleTitle>Bank Deposit</roleTitle>
   </roles>
   <roles>
      <roleId>4</roleId>
      <roleTitle>Void</roleTitle>
   </roles>
   <userName>boadmin</userName>
</authStatus>
```

AuthStatusWithProfile

JSON:

```
{
   "roles":    [
    {"roleId": "37","roleTitle": "Administration"},
    {"roleId": "49","roleTitle": "Bank Deposit"},
    {"roleId": "4", "roleTitle": "Void"}
   ],
   "userName": "boadmin",
   "profile":    {
      "configurationSettings": {"entry":       [
       {"key": "serviceCallTimeout","value": "30"},
       {"key": "serviceCallTenderTimeout","value": "300"},
       {"key": "inactiveWarningTimeout","value": "840"},
       {"key": "inactiveLogoutTimeout","value": "900"}
      ]},
      "parameterSettings": {"entry":       [
  {"key": "TimeoutInactiveWithTransaction","value": "15"},
  {"key": "IdentifyCashierEveryTransaction","value": "N"},
  {"key": "GiftCardsAccepted","value": "Y"},
  {"key": "CreditCardsAccepted","value": "Y"},
  {"key": "TimeoutInactiveWithoutTransaction","value": "15"}
      ]},
      "reasonCodeSets": {"entry":       [
                  {
            "key": "TransactionSuspendReasonCodes",
            "value": {"item":             [
            {"code": "43","text": "Operator Request"},
            {"code": "42","text": "Customer Request"}
            ]}
         },
                  {
            "key": "PriceOverrideReasonCodes",
            "value": {"item":             [
            {"code": "3","text": "Defective"},
            {"code": "5","text": "Signage Error"},
            {"code": "2","text": "Competition Price"},
            {"code": "1","text": "Ad Price"},
            {"code": "4","text": "Manager's Special"}
            ]}
         },
      ]},
      "registerID": "227",
      "storeCurrency":       {
```

```
         "@isoCode": "USD",
         "@formattedValue": "0.00",
         "@decimalValue": "0.00",
         "@currencySymbol": "$"
      },
      "storeID": "04241"
   }
}
```

XML:

```
<authStatusWithProfile>
   <roles>
      <roleId>37</roleId>
      <roleTitle>Administration</roleTitle>
   </roles>
   <roles>
      <roleId>49</roleId>
      <roleTitle>Bank Deposit</roleTitle>
   </roles>
      <roleId>4</roleId>
      <roleTitle>Void</roleTitle>
   </roles>
   <userName>boadmin</userName>
   <profile>
      <configurationSettings>
         <entry>
            <key>serviceCallTimeout</key>
            <value>30</value>
         </entry>
         <entry>
            <key>serviceCallTenderTimeout</key>
            <value>300</value>
         </entry>
         <entry>
            <key>inactiveWarningTimeout</key>
            <value>840</value>
         </entry>
         <entry>
            <key>inactiveLogoutTimeout</key>
            <value>900</value>
         </entry>
      </configurationSettings>
      <parameterSettings>
         <entry>
            <key>TimeoutInactiveWithTransaction</key>
            <value>15</value>
         </entry>
         <entry>
            <key>IdentifyCashierEveryTransaction</key>
            <value>N</value>
         </entry>
         <entry>
            <key>GiftCardsAccepted</key>
            <value>Y</value>
         </entry>
         <entry>
            <key>CreditCardsAccepted</key>
            <value>Y</value>
         </entry>
         <entry>
```

```
            <key>TimeoutInactiveWithoutTransaction</key>
            <value>15</value>
         </entry>
      </parameterSettings>
      <reasonCodeSets>
         <entry>
            <key>TransactionSuspendReasonCodes</key>
            <value>
               <item>
                  <code>43</code>
                  <text>Operator Request</text>
               </item>
               <item>
                  <code>42</code>
                  <text>Customer Request</text>
               </item>
            </value>
         </entry>
         <entry>
            <key>PriceOverrideReasonCodes</key>
            <value>
               <item>
                  <code>3</code>
                  <text>Defective</text>
               </item>
               <item>
                  <code>5</code>
                  <text>Signage Error</text>
               </item>
               <item>
                  <code>2</code>
                  <text>Competition Price</text>
               </item>
               <item>
                  <code>1</code>
                  <text>Ad Price</text>
               </item>
               <item>
                  <code>4</code>
                  <text>Manager's Special</text>
               </item>
            </value>
         </entry>
      </reasonCodeSets>
      <registerID>202</registerID>
      <storeCurrency isoCode="USD"
                     formattedValue="0.00"
                     decimalValue="0.00"
                     currencySymbol="$"/>
      <storeID>04241</storeID>
   </profile>
</authStatusWithProfile>
```

ItemStatus

JSON:

```
{"itemStatus": {
   "currentPrice":    {
      "@isoCode": "USD",
```

```
      "@formattedValue": "49.99",
      "@decimalValue": "49.99",
      "@currencySymbol": "$"
   },
   "description": "Chess set",
   "itemID": "20020002",
   "screenSaleMessage": "",
   "url": "http://localhost:7009/mobilepos/image?item=20020002"
}}
```

XML:

```
<itemStatuses>
 <itemStatus>
   <currentPrice isoCode="USD"
       formattedValue="49.99"
       decimalValue="49.99"
       currencySymbol="$"/>
   <description>Chess set</description>
   <itemID>20020002</itemID>
   <screenSaleMessage/>
   <url>http://localhost:7009/mobilepos/image?item=20020002</url>
 </itemStatus>
</itemStatuses>
```

InventoryStatus

JSON:

```
{"inventoryStatus": [
   {"inventoryResults":    {
      "@vendorReturnedQuantity": "3.00",
      "@unavailableQty": "8.00",
      "@transferReservedQuantity": "1.00",
      "@totalSOH": "10.00",
      "@storeID": "01211",
      "@onOrderQty": "0.00",
      "@itemID": "100160823",
      "@inTransitQty": "2.00",
      "@customerReservedQuantity": "0.00",
      "@availableQty": "2.00",
      "storeInfo":       {
         "@storeID": "01211",
         "@geoCode": "",
         "address":          {
            "@state": "",
            "@postalCodeExtension": "",
            "@postalCode": "",
            "@country": "",
            "@city": "",
            "@addressType": "-1"
         },
         "localizedLocationNames":
           {"defaultLocale": null,
            "textMap": {"item":
             [{"locale": {"@variant": "","@language": "zh","@country": ""},
               "string": ""},
              {"locale": {"@variant": "","@language": "en","@country": ""},
               "string": ""}
```

```
            ]}
         },
         "storeDistrict":          {
            "@identifier": "",
            "localizedDescriptions":
              {"defaultLocale": null,
               "textMap": {"item":
                [{"locale": {"@variant": "","@language": "zh","@country": ""},
                  "string": ""},
                 {"locale": {"@variant": "","@language": "en","@country": ""},
                   "string": ""}
                ]}
              }
         },
         "storeRegion":          {
            "@identifier": "",
            "localizedDescriptions":
              {"defaultLocale": null,
               "textMap": {"item":
                [{"locale": {"@variant": "","@language": "zh","@country": ""},
                  "string": ""},
                 {"locale": {"@variant": "","@language": "en","@country": ""},
                  "string": ""
                  }
                ]}
            }
         }
      }
   }}
]}
```

XML:

```
<inventoryStatuses>
   <inventoryStatus>
      <inventoryResults vendorReturnedQuantity="3.00"
                        unavailableQty="8.00"
                        transferReservedQuantity="1.00"
                        totalSOH="10.00"
                        storeID="01211"
                        onOrderQty="0.00"
                        itemID="100160823"
                        inTransitQty="2.00"
                        customerReservedQuantity="0.00"
                        availableQty="2.00">
         <storeInfo storeID="01211" geoCode="">
            <address state=""
                     postalCodeExtension=""
                     postalCode=""
                     country=""
                     city=""
                     addressType="-1"/>
            <localizedLocationNames>
               <defaultLocale/>
               <textMap>
                  <item>
                     <locale variant="" language="zh" country=""/>
                     <string/>
                  </item>
                  <item>
                     <locale variant="" language="en" country=""/>
```

```
                  <string/>
               </item>
            </textMap>
         </localizedLocationNames>
         <storeDistrict identifier="">
            <localizedDescriptions>
               <defaultLocale/>
               <textMap>
                  <item>
                    <locale variant="" language="zh" country=""/>
                     <string/>
                  </item>
                  <item>
                     <locale variant="" language="en" country=""/>
                     <string/>
                  </item>
               </textMap>
            </localizedDescriptions>
         </storeDistrict>
         <storeRegion identifier="">
            <localizedDescriptions>
               <defaultLocale/>
               <textMap>
                  <item>
                     <locale variant="" language="zh" country=""/>
                     <string/>
                  </item>
                  <item>
                     <locale variant="" language="en" country=""/>
                     <string/>
                  </item>
               </textMap>
            </localizedDescriptions>
         </storeRegion>
      </storeInfo>
   </inventoryResults>
</inventoryStatus>
</inventoryStatuses>
```

GiftCardStatus

JSON:

```
{
   "currentBalance":    {
      "@isoCode": "USD",
      "@formattedValue": "75.00",
      "@decimalValue": "75.00",
      "@currencySymbol": "$"
   },
   "giftcardStatusCode": "Active"
}
```

XML:

```
<giftcardStatus>
   <currentBalance isoCode="USD"
                   formattedValue="75.00"
                   decimalValue="75.00"
                   currencySymbol="$"/>
```

```
   <giftcardStatusCode>Active</giftcardStatusCode>
</giftcardStatus>
```

RegisterStatus

JSON:

```
{
   "operatorId": "0",
   "registerId": "227",
   "registerStatus": "1",
   "storeId": "04241",
   "storeStatus": "1",
   "tillId": "22701",
   "tillStatus": "1"
}
```

XML:

```
<registerStatus>
   <operatorId>0</operatorId>
   <registerId>227</registerId>
   <registerStatus>1</registerStatus>
   <storeId>04241</storeId>
   <storeStatus>1</storeStatus>
   <tillId>22702</tillId>
   <tillStatus>1</tillStatus>
</registerStatus>
```

RegisterProfileConfiguration

JSON:

```
{
   "configurationSettings": {"entry":    [
         {"key": "serviceCallTimeout","value": "30"},
         {"key": "serviceCallTenderTimeout","value": "300"},
         {"key": "inactiveWarningTimeout","value": "840"},
         {"key": "inactiveLogoutTimeout","value": "900"}
   ]},
   "parameterSettings": {"entry":    [
     {"key": "TimeoutInactiveWithTransaction","value": "15"},
     {"key": "IdentifyCashierEveryTransaction","value": "N"},
     {"key": "GiftCardsAccepted","value": "Y"},
     {"key": "CreditCardsAccepted","value": "Y"},
     {"key": "TimeoutInactiveWithoutTransaction","value": "15"}
   ]},
   "reasonCodeSets": {"entry":    [
     {"key": "TransactionSuspendReasonCodes",
      "value": {"item":
         [
           {"code": "43","text": "Operator Request"},
           {"code": "42","text": "Customer Request"}
         ]}
      },
      {"key": "PriceOverrideReasonCodes",
       "value": {"item":
          [
           {"code": "3","text": "Defective"},
           {"code": "5","text": "Signage Error"},
```

```
            {"code": "2","text": "Competition Price"},
            {"code": "1","text": "Ad Price"},
            {"code": "4","text": "Manager's Special"}
         ]}
      }
   ]},
   "registerID": "202",
   "storeCurrency":    {
      "@isoCode": "USD",
      "@formattedValue": "0.00",
      "@decimalValue": "0.00",
      "@currencySymbol": "$"
   },
   "storeID": "04241"
}
```

XML:

```
<registerProfileConfiguration>
   <configurationSettings>
      <entry>
         <key>serviceCallTimeout</key>
         <value>30</value>
      </entry>
      <entry>
         <key>serviceCallTenderTimeout</key>
         <value>300</value>
      </entry>
      <entry>
         <key>inactiveWarningTimeout</key>
         <value>840</value>
      </entry>
      <entry>
         <key>inactiveLogoutTimeout</key>
         <value>900</value>
      </entry>
   </configurationSettings>
   <parameterSettings>
      <entry>
         <key>TimeoutInactiveWithTransaction</key>
         <value>15</value>
      </entry>
      <entry>
         <key>IdentifyCashierEveryTransaction</key>
         <value>N</value>
      </entry>
      <entry>
         <key>GiftCardsAccepted</key>
         <value>Y</value>
      </entry>
      <entry>
         <key>CreditCardsAccepted</key>
         <value>Y</value>
      </entry>
      <entry>
         <key>TimeoutInactiveWithoutTransaction</key>
         <value>15</value>
      </entry>
   </parameterSettings>
   <reasonCodeSets>
      <entry>
```

```
            <key>TransactionSuspendReasonCodes</key>
            <value>
               <item>
                  <code>43</code>
                  <text>Operator Request</text>
               </item>
               <item>
                  <code>42</code>
                  <text>Customer Request</text>
               </item>
            </value>
         </entry>
         <entry>
            <key>PriceOverrideReasonCodes</key>
            <value>
               <item>
                  <code>3</code>
                  <text>Defective</text>
               </item>
               <item>
                  <code>5</code>
                  <text>Signage Error</text>
               </item>
               <item>
                  <code>2</code>
                  <text>Competition Price</text>
               </item>
               <item>
                  <code>1</code>
                  <text>Ad Price</text>
               </item>
               <item>
                  <code>4</code>
                  <text>Manager's Special</text>
               </item>
            </value>
         </entry>
      </reasonCodeSets>
      <registerID>202</registerID>
      <storeCurrency isoCode="USD"
                     formattedValue="0.00"
                     decimalValue="0.00"
                     currencySymbol="$"/>
      <storeID>04241</storeID>
   </registerProfileConfiguration>
```

TransactionStatus

JSON:

```
{
   "amountDue":    {
      "@isoCode": "USD",
      "@formattedValue": "0.00",
      "@decimalValue": "0.00",
      "@currencySymbol": "$"
   },
   "associateID": "boadmin",
   "associateName": "Application Administrator",
   "discountTotal":    {
```

```
      "@isoCode": "USD",
      "@formattedValue": "1.00",
      "@decimalValue": "1.00",
      "@currencySymbol": "$"
   },
   "grandTotal":    {
      "@isoCode": "USD",
      "@formattedValue": "9.47",
      "@decimalValue": "9.47",
      "@currencySymbol": "$"
   },
   "lineItems":    {
      "associateID": "boadmin",
      "associateName": "Application Administrator",
      "description": "CoolBox",
      "discounts":       {
         "amount":          {
            "@isoCode": "USD",
            "@formattedValue": "1.00",
            "@decimalValue": "1.00",
            "@currencySymbol": "$"
         },
         "method": "Amount",
         "name": "Senior Citizen",
         "reasonCode": "2311",
         "type": "Manual"
      },
      "extendedDiscountedPrice":       {
         "@isoCode": "USD",
         "@formattedValue": "9.00",
         "@decimalValue": "9.00",
         "@currencySymbol": "$"
      },
      "extendedPrice":       {
         "@isoCode": "USD",
         "@formattedValue": "10.00",
         "@decimalValue": "10.00",
         "@currencySymbol": "$"
      },
      "itemNumber": "1234",
      "lineNumber": "0",
      "price":       {
         "@isoCode": "USD",
         "@formattedValue": "10.00",
         "@decimalValue": "10.00",
         "@currencySymbol": "$"
      },
      "priceModifiable": "true",
      "quantity": "1",
      "quantityModifiable": "true"
   },
   "subtotal":    {
      "@isoCode": "USD",
      "@formattedValue": "10.00",
      "@decimalValue": "10.00",
      "@currencySymbol": "$"
   },
   "taxTotal":    {
      "@isoCode": "USD",
      "@formattedValue": "0.47",
```

```
      "@decimalValue": "0.47",
      "@currencySymbol": "$"
   },
   "tenderLineItems":    {
      "amount":       {
         "@isoCode": "USD",
         "@formattedValue": "9.47",
         "@decimalValue": "9.47",
         "@currencySymbol": "$"
      },
      "description": "Credit",
      "tenderLineNumber": "0"
   },
   "totalQuantity": "1",
   "transactionId": "042412350011",
   "transactionStatus": "2"
}
```

XML:

```
<transactionStatus>
   <amountDue isoCode="USD" formattedValue="0.00" decimalValue="0.00"
 currencySymbol="$"/>
   <associateID>boadmin</associateID>
   <associateName>Application Administrator</associateName>
   <discountTotal isoCode="USD" formattedValue="1.00" decimalValue="1.00"
 currencySymbol="$"/>
   <grandTotal isoCode="USD" formattedValue="9.47" decimalValue="9.47"
 currencySymbol="$"/>
   <lineItems>
      <associateID>boadmin</associateID>
      <associateName>Application Administrator</associateName>
      <description>CoolBox</description>
      <discounts>
         <amount isoCode="USD" formattedValue="1.00" decimalValue="1.00"
 currencySymbol="$"/>
         <method>Amount</method>
         <name>Senior Citizen</name>
         <reasonCode>2311</reasonCode>
         <type>Manual</type>
      </discounts>
      <extendedDiscountedPrice isoCode="USD" formattedValue="9.00"
 decimalValue="9.00" currencySymbol="$"/>
      <extendedPrice isoCode="USD" formattedValue="10.00" decimalValue="10.00"
 currencySymbol="$"/>
      <itemNumber>1234</itemNumber>
      <lineNumber>0</lineNumber>
      <price isoCode="USD" formattedValue="10.00" decimalValue="10.00"
 currencySymbol="$"/>
      <priceModifiable>true</priceModifiable>
      <quantity>1</quantity>
      <quantityModifiable>true</quantityModifiable>
   </lineItems>
   <subtotal isoCode="USD" formattedValue="10.00" decimalValue="10.00"
 currencySymbol="$"/>
   <taxTotal isoCode="USD" formattedValue="0.47" decimalValue="0.47"
 currencySymbol="$"/>
   <tenderLineItems>
      <amount isoCode="USD" formattedValue="9.47" decimalValue="9.47"
 currencySymbol="$"/>
      <description>Credit</description>
```

```
      <tenderLineNumber>0</tenderLineNumber>
   </tenderLineItems>
   <totalQuantity>1</totalQuantity>
   <transactionId>042412350012</transactionId>
   <transactionStatus>2</transactionStatus>
</transactionStatus>
```

ReceiptStatus

JSON:

```
{
   "emailed": "true",
   "printed": "false",
   "signatureSlipPrinted": "false"
}
```

XML:

```
<receiptStatus>
   <emailed>true</emailed>
   <printed>false</printed>
   <signatureSlipPrinted>false</signatureSlipPrinted>
</receiptStatus>
```

C

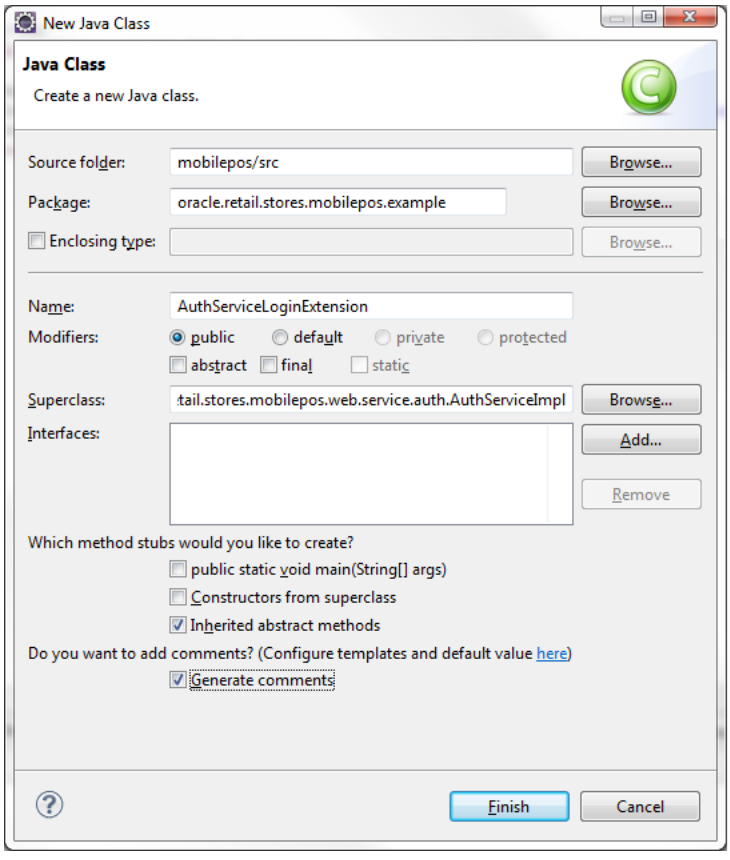
Appendix: Eclipse Screens

This appendix contains the screenshots for using Eclipse when extending Mobile POS and setting up an implementation environment.

Extending Mobile POS

This screen is referenced in Chapter 5.

Figure C–1 Eclipse New Java Class



Setting Up an Implementation Environment

These screens are referenced in Chapter 6.

Figure C–2 Eclipse Workspace Preferences

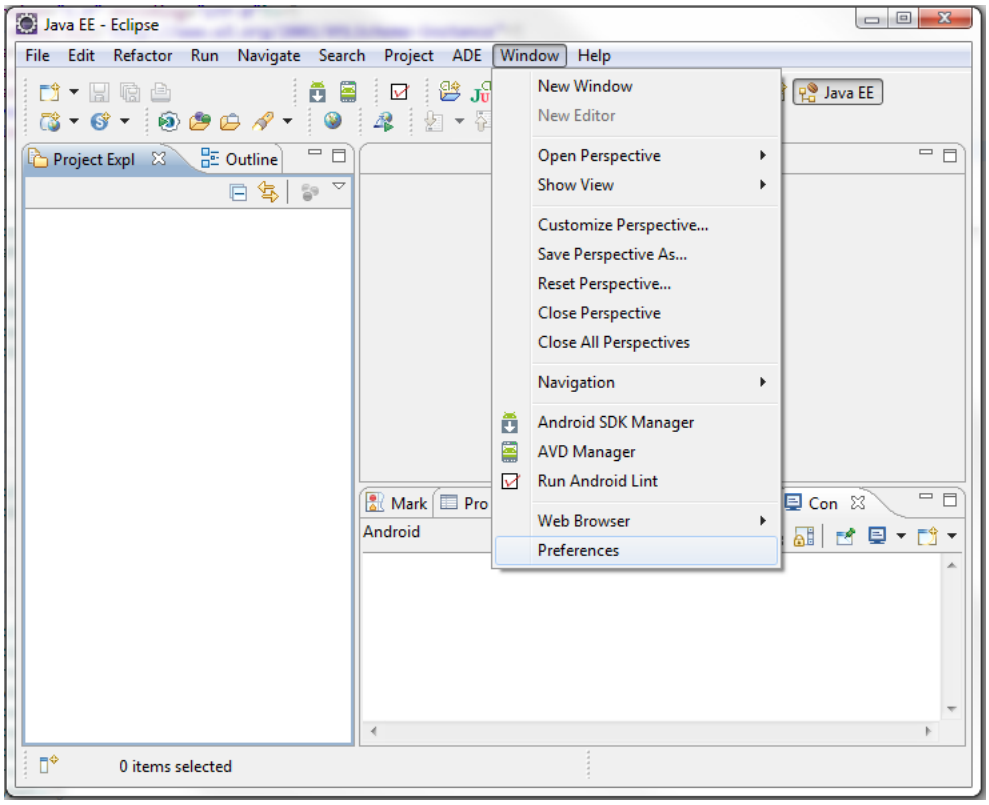


Figure C–3 Eclipse Runtime Environments

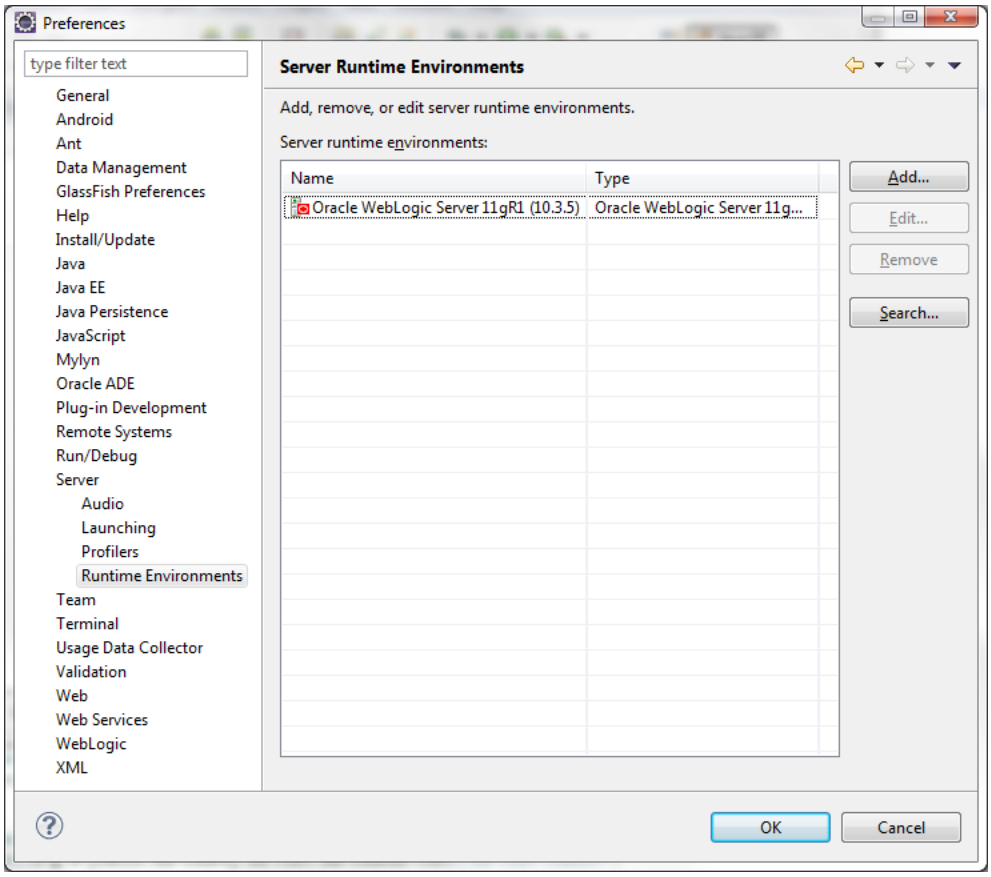


Figure C–4 Eclipse WAR Import Menu

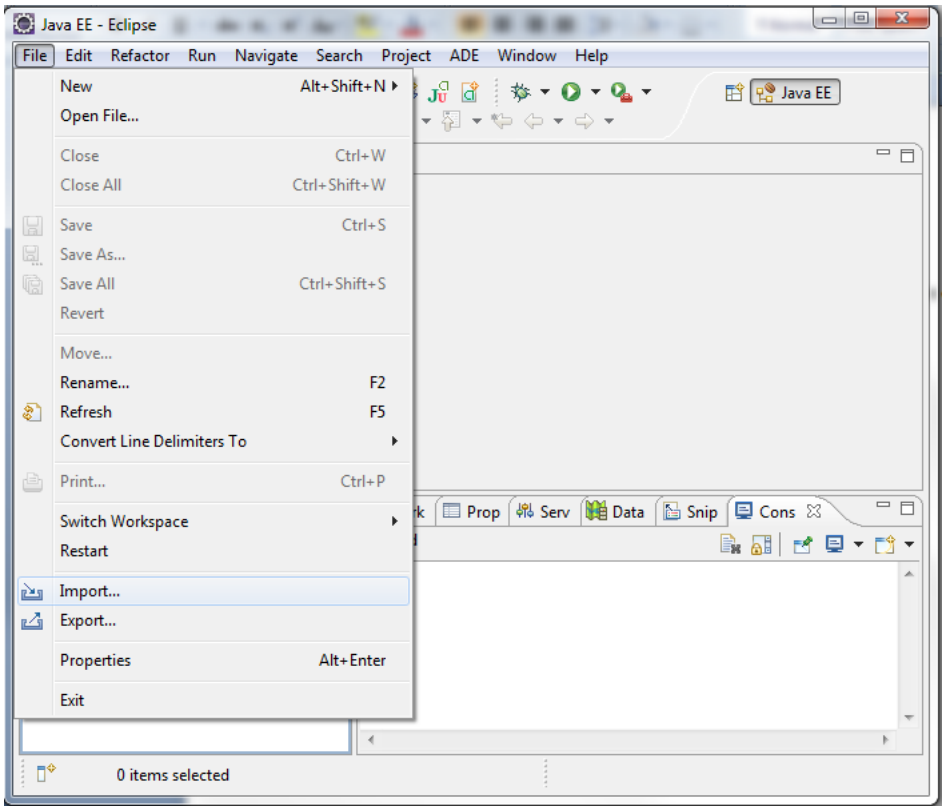


Figure C–5 Eclipse WAR Import Wizard

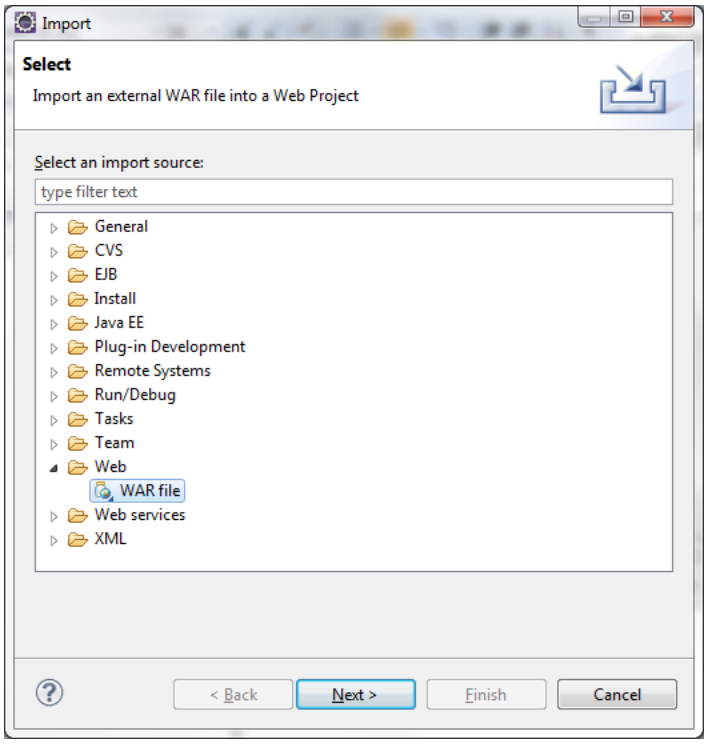


Figure C–6 Eclipse WAR Import Dialog

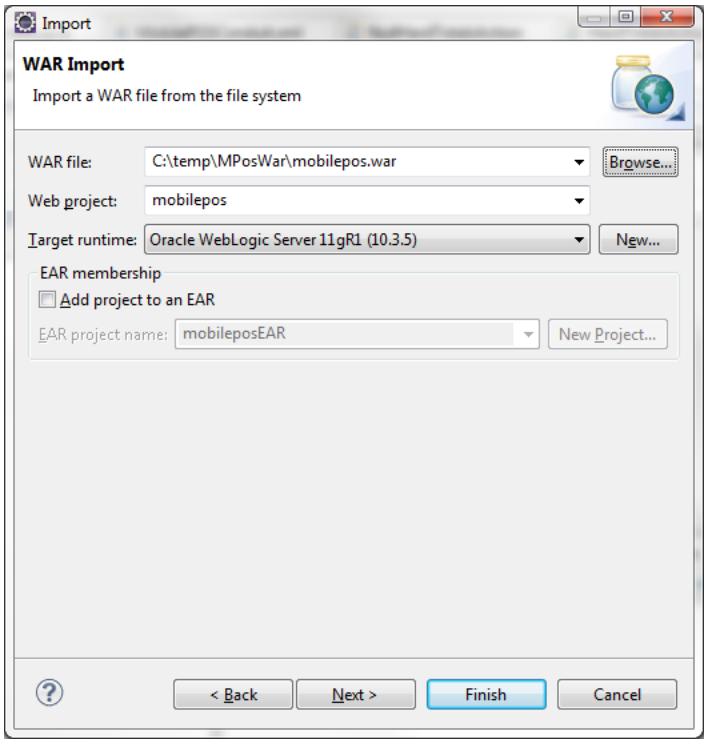


Figure C–7 Eclipse WAR Import: Web Libraries

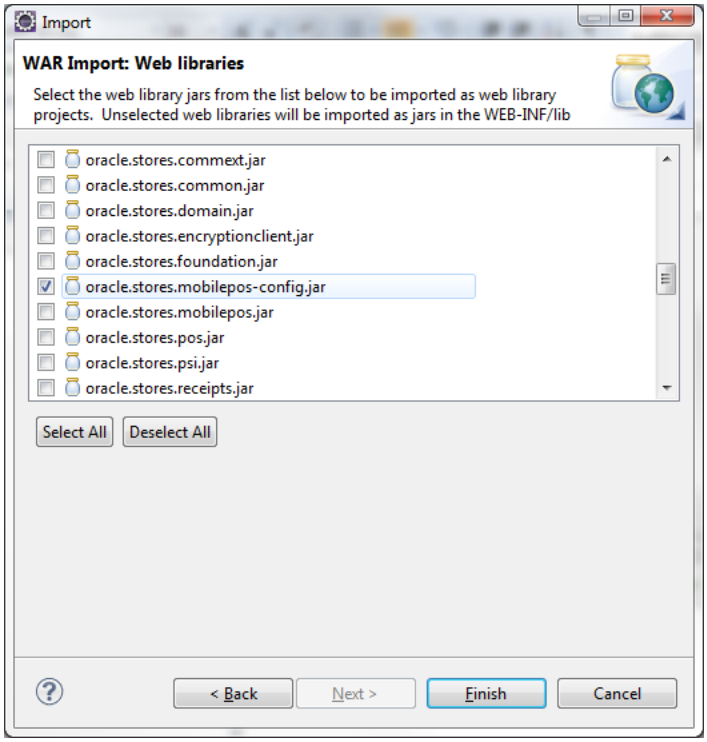


Figure C–8 Eclipse Mobile POS Workspace

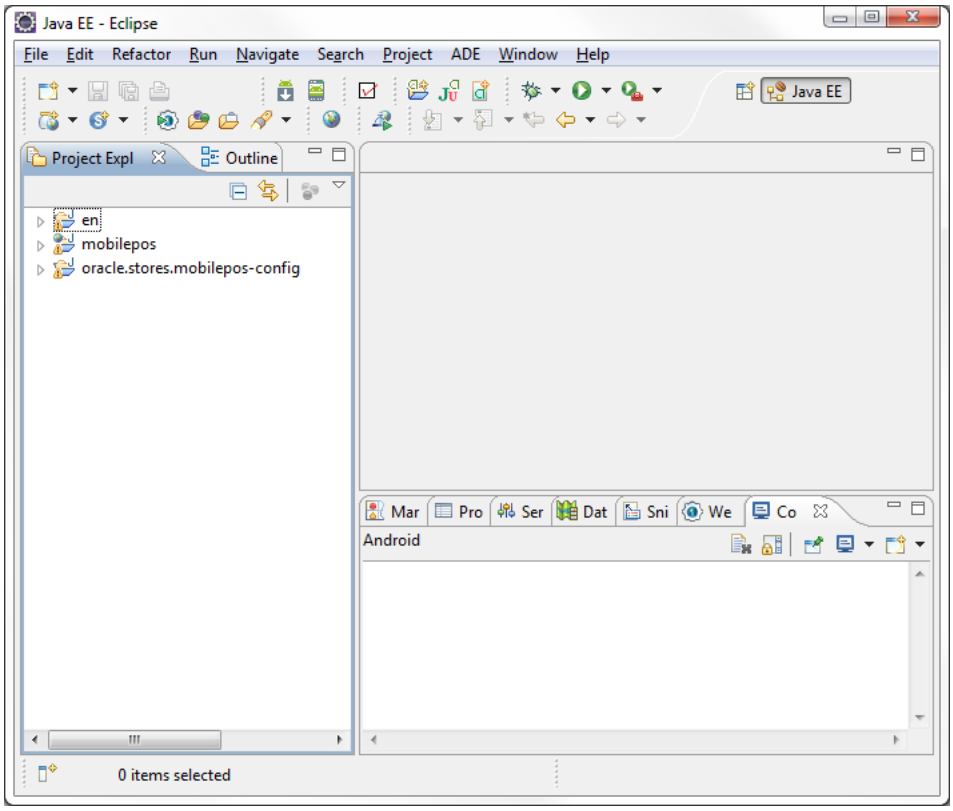


Figure C–9 Eclipse Add Library

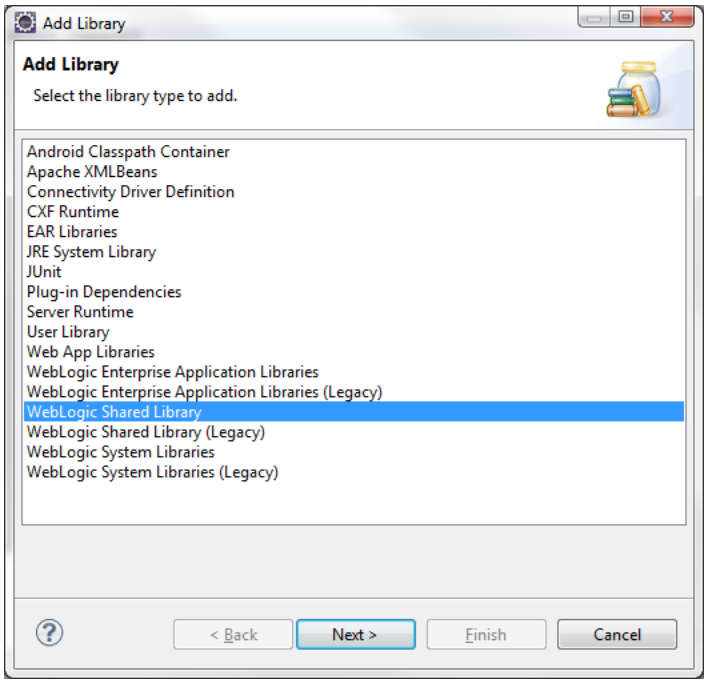


Figure C–10 Eclipse Select WebLogic Shared Library