Pepper
Software Development Kit
User Guide
3.2

Revision: 002.001.001

Author: Kyle Nitzsche

Table of Contents

Hello World: Getting Started

Hello World 1: Application Creation

Hello World 2: Pages and ToolBars

Building Applications

Adding and Distributing Applications

Sample Page Files

XML Reference

Preface

Description of this document

This document provides information about the Pepper® Software Development Kit (SDK).

Intended audience

The SDK and this book are intended for software developers who want to develop and customize Pepper applications.

The level of experience required varies depending on the task. In general, some experience with the following is appropriate:

- Java
- JavaScript
- HTML
- CSS
- XML
- XSL
- Ant
- Linux

How this document is organized

This document provides introductory, explanatory, tutorial, building and deployment, reference and supporting material, including:

- Introductory chapters

  Preface, Introduction and Getting Started with the SDK

- Explanatory chapter

  Framework and Application Architecture

- Tutorial chapters

  Hello World: Getting Started, Hello World 1: Application Creation, Hello World 2: Pages and ToolBars, Hello World 3: Getting Started with Java, Hello World 4: Advanced Java and Sample Page Files

  Source files for the tutorial are provided in the SDK.

- Build and deployment chapters

  Building Applications and Adding and Distributing Applications

- Reference appendix

  XML Reference

- Supporting appendix

  Glossary

Documentation conventions

This guide uses fonts and text styles for different purposes, as explained in the following table.

Table –1 Font usage and stylistic conventions

Font style	Used for	Examples

Monospace	Code samples	`public void MyClass() {`
		`...`
		`}`

	Command line output	`BUILD SUCCESSFUL` `Total time: 16 seconds`
Monospace bold	Commands	Use the **`ant rebuild`** command.
Bold	User interface literal names	Click the **Done** button.
Italic	Directory and file names	*pepper-sdk/applications* *package.ppld*
	Emphasis and new terms	This section takes a closer look at an application as a dynamic structure of *Sections* and a Section as a dynamic structure of *Pages*.

Table –1 Font usage and stylistic conventions (continued)

Font style	Used for	Examples
Italic enclosed in parentheses	Implementation specific names	*(yourApp)* is implementation specific in the following: *pepper-sdk/applications/(yourApp)/dist*
		(dns) is implementation specific in the following: http://(*dns*)/index.html

1

Introduction

What is Pepper?

The Pepper environment is software for the Internet era that's been designed specifically for mobile devices and low-cost computers. It's designed to be very easy to use by mainstream consumers, easy to build and develop applications for, highly customizable by device makers and distributors, and automatically upgradable for maintenance-free use by anyone.

The Pepper environment supports Web-connected applications that start with Mozilla and go much further than browser-only applications can go. The Pepper environment's unique application framework seamlessly integrates Mozilla, Java, Flash and media players with a persistent XML storage model to enable a whole new class of extremely sophisticated Web and media applications.

Applications can take advantage of persistent XML page storage, integrated Web services and Web pages, a secure execution environment, and access to leading-edge media players and all of the most widely used media formats. An application's user interface can be written using Java Swing/2D/3D, JavaScript/HTML/XHTML/CSS (“Ajax”), JavaScript/XUL, Flash Actionscript or basic HTML. New Web-connected applications can be quickly developed and deployed. In short, the Pepper environment has everything you need to create *applications that go beyond the browser*.

The Pepper environment’s unique and flexible graphical user interface eliminates clutter and gets out of the user’s way. Instead of lots of hard to organize, overlapping windows, the Pepper UI has tabs to organize active applications. It's also designed to minimize screen real estate usage -- a real plus on small-screen devices. The entire look and feel can be customized in many dimensions, from which applications are present, to colors, fonts, background images, language (locale), help text, layout, and more.

Automatic software updates of applications and of the Pepper framework itself guarantee users have what they need as their needs change.

The Pepper environment is the key part of Pepper Linux, a light-weight Linux distribution for mobile Internet devices and low-cost computers. The Pepper environment also runs as an application on other operating systems, such as Windows XP and most standard Linux distributions, including Fedora, Ubuntu, Suse, and others.

What is the Pepper SDK?

The SDK is a set of files, tools and instructions for developing Pepper applications that run in the Pepper Desktop Environment.

What can you do with the SDK?

You can use the SDK to:

- Develop fully integrated Pepper applications.

  Such applications run on the Pepper Application Framework. They make use of the framework and conform to framework user interface conventions. They can use the Default Section/Page framework, be Java based, or be a mixture of the two.

- Customize the Pepper Application Framework, including:
  - Defining the applications that can be launched.
  - Customizing the Tabs (Sections) available within the Keeper application, including adding and removing Tabs.

    Note: The *Keeper* is the root application in the Pepper Application Framework that, among other things, provides the **Applications** Tab containing icons for all other applications. For information, see “Keeper application” on page 2-10.
  - Customizing toolbars associated with a particular Page by adding and removing buttons.
  - Customizing the visual styling, including colors, fonts, images and overall layouts in the Pepper application framework.

    See “Customization” on page 10-151.

What Pepper versions are supported?

Applications developed with the 3.2 SDK are compatible with the following Pepper software versions:

- Pepper 3.2
- Pepper 3.1, except that applications cannot use methods or classes marked *since 3.2* in the Javadoc.

Applications developed with 3.0 SDK are compatible with the following Pepper software versions:

- Pepper 3.1
- Pepper 3.2, except that the classes `com.pepper.platform.program.NetworkListener` and `com.pepper.platform.program.NetworkEvent` have been removed from Pepper 3.2.

Applications that use either of these classes need to be modified to use the corresponding `classes com.pepper.platform.net.NetworkListener` or `com.pepper.platform.net.NetworkEvent,` which were introduced in Pepper 3.2.

Backwards compatibility

The javadoc for the Pepper Java Application Programming Interface (API) indicates the following:

- Any method or class introduced in version 3.2 or later is noted with *since <versionNumber>* in the relevant javadoc.

  Note: Be sure to set the minimum Pepper version required by your applications in each *package.ppld* file with the `<requiredKeeperVersion>` element to be as high as the highest Pepper version whose methods or classes you use. For example, if you use a method introduced in 3.2.0, you must set the `<requiredKeeperVersion>` to 3.2.0.

- Some methods or classes are marked in javadoc as *deprecated.*

  Deprecated methods or classes should be avoided. There is no guarantee that deprecated methods or classes will be available in future Pepper version.

Where does Pepper 3.2.0 run?

Pepper version 3.2.0 runs on:

- Pepper devices above Pepper Linux
- Windows as Pepper Desktop

  Note: The Pepper version is displayed in the Keeper's **Help** Section.

What Java versions are supported on Pepper Pads?

Java support varies by Pepper Pad, as follows:

- Pepper Pad 3

  Uses Java 5.0 (formerly known as Java 1.5). Applications developed using language features and APIs introduced in Java 5.0 work on Pepper Pad 3 and Pepper Desktop. They do not work on Pepper Pad 2. The default behavior of the build scripts included with the SDK is to target Java 1.4, so developers writing applications that use features or APIs introduced in Java 5.0 must change their build scripts to target 5.0.

- Pepper Pad 2

  A hybrid Java environment. The core classes are Java 1.3. The java.awt.* classes and javax.swing.* classes are Java 1.4.

What does the SDK distribution include?

The SDK distribution includes the following:

- This document
- The application build system

  See “Building Applications” on page 11-189.
- SDK libraries and executables including jar files for the Pepper framework classes, the XML parser (Xerces), the XSL processor (Xalan) and JDOM (for accessing XML elements)
- The Hello World Tutorial

  This takes you through four phases of application development that demonstrate many critical aspects of developing Default type and Java type applications.

  See “Hello World: Getting Started” on page 5-85.
- A template for new applications

  Use of the application template is described in Phase One of the Hello World Tutorial.

  See “Hello World 1: Application Creation” on page 6-89.
- Javadoc documentation of framework Java classes and methods you can use during application development.

  See “Javadoc” on page 2-20.

What development platforms are supported?

Pepper applications can be developed and built on a Windows, Linux or Mac system that supports Java and has the SDK installed.

> **Note:** Since the Pepper Application Framework is not officially supported on Mac or Linux, applications developed on these platforms are generally tested on the Pepper Pad.

Java integrated development environment

You can develop the Java aspects of your applications using any industry-standard Java integrated development environment (IDE), for example Eclipse. Setting up such an environments may require additional steps that are not covered in detail in this document.

> **Note:** The SDK directory structure is determined by integral Ant build scripts. If you modify the directory structure you will need to handle building the Java and creating the application distribution package.

2

Getting Started with the SDK

SDK directories

The SDK has a specific directory structure. To develop an application, you have to work within this structure and follow its rules. When you build an application, the build system expects files and directories to be in certain places and creates new directories with new and copied files in specific locations.

So, it is essential to be familiar with the SDK directory structure and to understand how it changes during a build.

The SDK's base directory is *pepper-sdk*.

SDK root directories

The *pepper-sdk* directory contains the following subdirectories:

- *applications*

  Contains subdirectories for each application under development or included with the SDK, an application template directory and Hello World Tutorial source files.

- *bootstrap*

  Has files and subdirectories required by the build system. None of these files or directories should be edited.

- *doc*

  The *SDK javadoc.zip* file resides here. This is complete documentation of the Pepper Java API., including classes, methods, interfaces, and etc.

  The SDK User Guide PDF file also resides here.

- *lib*

  Contains required libraries (jar files, zip files, etc.) included with the SDK.

- *themes*

  Use this to develop custom themes that control the Pepper Application Framework's visual styling. See "Custom themes" on page 10-167.

SDK files

The *pepper-sdk* directory contains the following files:

- *setup.bat*

  Upon execution in a command line window on a Windows system, *setup.bat* sets environment variables required by the SDK build system (for the current command line session).

  Note: You must ensure *setup.bat* sets the environment correctly with respect to your SDK installation directories. See "Setting the Build Environment Variables" on page 11-190.

- *setup.sh*

  Upon execution in a command line window on a Linux system, sets environment variables required by the SDK build system (for the current command line session).

  Note: You must ensure *setup.sh* sets the environment correctly with respect to your SDK installation directories. See "Setting the Build Environment Variables" on page 11-190.

- *README.txt*

  States the SDK version, provides a pointer to this PDF file, and includes some important legal statements.

pepper-sdk/applications directory

pepper-sdk/applications is the root directory for application development work. Each application under development or being customized in the SDK is contained in a directory you create inside *pepper-sdk/applications*.

In addition to application-specific subdirectories you create, the SDK contains a template directory, a tutorial resources directory, and a directory containing a sample application, as follows:

- *applicationTemplate*

  Contains a set of files and folders you can use as a template for creating new applications. This is a template directory and, as such, it should not be built directly. Instead, copy and paste this directory. Only build the new directory.

  See Chapter 6, "Hello World 1: Application Creation".

- *HelloWorldResources*

  Contains source files for the various phases of the Hello World Tutorial included with the SDK.

  See Chapter 6, "Hello World 1: Application Creation".

Application-specific directories

As noted, each application requires a dedicated root directory inside *pepper-sdk/applications*. Each application root directory requires a number of specific subdirectories for certain types of files. The build system depends on finding these specific subdirectories and files. The build system also creates certain application-specific directories, specifically, *env* and *dist*.

Each application has the following SDK directories:

- *build*

Contains the *build.xml* file for this package. This file controls how the ant build system builds the application. This file must be modified for new applications.

See "Setting up Application's Build System" on page 11-190

Not created by the build

- *design*

  Contains many files that define the structure of your application and its visual styling. You customize these files as you develop an application, as discussed throughout this book. During application building, all files in *design* are zipped into *design.zip* and placed in the *dist* directory. When the application is installed in the Keeper, the *design.zip* file is copied to the application's Pepper Application Framework installation directory.

  If the Pepper Application Framework is placed in Design mode, *design.zip* is extracted. See "Design Mode" on page 2-13.

  Not created by the build

- *prebuilt*

  Contains pre-built Pages, typically segregated into subdirectories.

  See "Pre-built Pages" on page 4-69.

  Not created by the build

- *src*

  All Java source files must be in the application's *src* directory, as follows:

  pepper-sdk/applications/(application)/src.

  Within *src*, you may optionally have subfolders. However, if subfolders are present they must be consistent with the Java package statements at the start of each Java source file.

  For example, if a file's Java package statement is:

```
package com.pepper.HW;
```

  Then the files must reside here:

  pepper-sdk/HelloWorldTutorial/src/com/pepper/HW

  Not created by the build

- *data*

  Created by the build as a temporary container. Contains the application's XML instance files. After this directory is created by the build, it is zipped up and placed in the *pepper-sdk/applications/(application)/dist* directory. It is zipped into *data.zip* and, during application installation, is copied into the application's Pepper Application Framework installation directory. During application execution, these files are accessed. If they are modified, or if new XML instance files are created, a *data* directory is created in the application's installation directory and the modified or new files are placed there. None of these files should be edited.

  Created by the build

- *dist*

Contains the built files that are specific to this application. When adding a built application to a Pepper, you copy only this directory to the Pepper.

Created by the build

- *env*

  Created by the build as a temporary container during building. Contains the jar file for the compiled Java source class(es) for this application.

  Created by the build

- *javadoc*

  Contains javadoc HTML files that document the Java source files in this application if you have entered javadoc comments.

  Created by the build

Pepper Application Framework installation directories

The root directory of an installed Pepper Application Framework on a Pepper device is:

/opt/pepper

The root directory of an installed Pepper Application Framework on Pepper Desktop on Windows is:

C:\Documents and Settings\(user)\My Documents\Pepper

Note: Throughout this book the Pepper Application Framework root run-time directory is generally referred to as *Pepper*.

Subdirectories are of two types:

- The Pepper Application Framework has a set of subdirectories required for framework operations.
- Each application has a subdirectory.

Pepper Application Framework run-time directories

The framework has several subdirectories that are not associated with specific applications. Most are used by the framework and their files should never be modified.

The following are framework directories that contain files that you may want to access or modify:

Note: On the Pepper devices, the root directory uses a lower case “p” in *pepper*. On Pepper Desktop on Windows, the root directory has an upper case “P,” as in *Pepper*.

- *Pepper/Logs*

  Contains Pepper Application Framework log files. For information, see “Keeper event log” on page 2-19.

  Note: Pepper devices also have Pepper Linux log files stored in other directories.

- *Pepper/resources*

When the Pepper Application Framework is in Design Mode and no custom theme has been developed and installed, this directory contains images, XSL files, CSS files, JavaScript files and other files that control the framework's visual styling. You can customize the visual look and feel of the Pepper Application Framework by modifying these files.

For information about Design mode, see "Design Mode" on page 2-13.

For information about customization, see "Customization" on page 10-151.

- *Pepper/sounds*

  Contains mp3 sound files used by the Pepper Application Framework and applications. You can add your own sound files and configure the framework to associate them with specified events.

Application run-time directories

Each application has its own run-time directory that is created when the application is installed. For example:

Pepper/AnApplication-0

Application run-time directory name

The application directory name has two parts:

- The first part of an application's directory name is derived from the application's `<title>` element in the application's *package.ppld* file, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+">
    <information>
        <title>AnApplication</title>
...
```

  This `<title>` results in an run-time directory named *AnApplication-0*

  For reference information about the *package.ppld* file and the meaning and use of its XML elements, see "XML Reference" on page B-207.

- The second part of an application's run-time directory name is either "*-0*" or "*-(GUID)".*

  Note: The GUID is an alpha-numeric string that uniquely identifies the application.

Application directory contents

Every application has at least the following files:

Note: These files are either specified with the `<resources>` element in the application's *package.ppld* file or created as an inherent part of application building and installation.

- The application's jar file

  At a minimum, this contains the application's base class. For information, see "AbstractPepperProgram life cycle" on page 4-29.

  It also includes Java classes you developed for the application.

- Additional jar files the application needs

  You can include jar files and use them as resources in your application. For information, see "Adding Existing Jar Files to the Build" on page 11-194.

- *data.zip*

  Contains the initial version of the application's built *data* directory. The built *data* directory contains the XML instance files for the application's Sections and Pages before run-time events add new XML instance files. If new XML instance files are created, the *(application)/data* directory is created and they are placed there.

  For information, see "Application structure — a run-time sample" on page 4-36.

- *design.zip*

  Contains the complete contents of the application's *design* directory in the SDK.

  When you place the Pepper Application Framework in Design mode, *design.zip* is expanded and extracted, and the application uses the extracted files instead of those in *design.zip*. This enables you to modify these files on the fly and observe their effects in the running application, instead of having to rebuild an application every time non-Java files are modified. For information, see "Design Mode" on page 2-13.

Keeper application

The Keeper application is the Pepper Application Framework's root application that is launched upon framework startup. The Keeper application manages all other applications.

The Keeper application's directory has, for the most part, the same subdirectories and key files as other applications.

Debug Mode

Debug Mode is a mode of framework execution that is useful when developing applications.

When you enter debug Mode, two new menu items appear above the usual GUI: **Debug** and **Support**. The functions available from these two menus may vary from release to release, but important functions include:

- Accessing Support menu items, including such functions as displaying the Pepper log file, resetting the Pepper, an application, Sync History, and more.
- Adding an application.

  In normal operational mode, applications are added when the framework is updated as a part of the normal update process. Debug Mode enables you to manually add a custom application.

  See "Adding an application" on page 2-12.

- Updating the framework's *design.zip* and/or *data.zip* files.

  See "Updating framework zip files" on page 2-13.

- Automatically detecting rebuilt applications and refreshing them in the framework.

  See "Automatically refreshing applications under development" on page 2-12.

- Viewing the XML, HTML, and DOM of the current Page.

  As explained in “Framework and Application Architecture” on page 4-27, Pages are transformed into HTML and rendered for display. Debug Mode enables you to view the current Page’s underlying XML file and its generated HTML.

  See “Viewing Page’s underlying HTML, XML and DOM” on page 2-12.

- Viewing the framework log.

  See “Keeper event log” on page 2-19.

- Refreshing the current Page.

  See “Refreshing the current Page” on page 2-12.

Entering Debug Mode

A Key combination is available that toggles between normal execution mode and Debug Mode. You can also launch the framework in Debug Mode by editing a framework launch configuration file.

Toggling Debug Mode during framework execution

Toggle between Debug Mode and normal mode with the follow key combination:

ctrl + **shift** + **0**

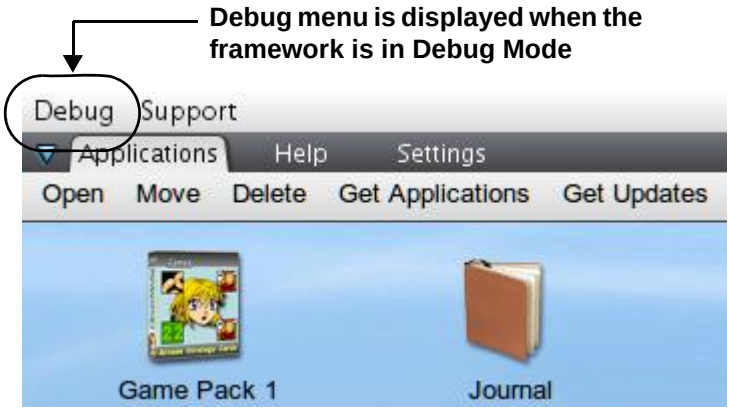
Launching the framework in Debug Mode

See “Setting framework system properties” on page 2-15.

Debug menu

When the framework is in Debug Mode, a **Debug** menu is available that provides access to various debug functions.

Figure 2–1 Debug Mode



Adding an application

When the framework is in Debug Mode you can manually add an application, if the application's built files are accessible to the framework (see "Making an application's files accessible to the framework" on page 12-197).

For the procedure to add an application manually in Debug Mode, see "Adding a local application to the framework" on page 12-201.

Automatically refreshing applications under development

When you launch an application in Debug Mode, the Pepper refreshes the application, when possible, from the application's distribution package *before launching it.* This makes it easy to iteratively test sequential development changes because you can simply refresh the distribution package on the Pepper platform with the newly built files, then relaunch the application (in Debug Mode) to evaluate the changes.

The automatic refresh occurs:

- When the application has been manually added in Debug Mode (by **Debug > Add Application** or by clicking on an application distribution URL)
- When the framework is running in Debug Mode

  Note: When you make development changes to the application that do not involve Java files (that is, when you modify CSS, JavaScript and XSL files), you can use Design Mode to auto-detect and update the application. See "Design Mode" on page 2-13.

Viewing Page's underlying HTML, XML and DOM

When the framework is in Debug Mode, you can view the current Page's XML, HTML, or DOM structure.

- Viewing the current Page's underlying XML

  Select **Debug** > **View Page XML**

  The XML is displayed in the framework log.

  For information, see "Keeper event log" on page 2-19.
- Viewing the current Page's underlying HTML

  Select **Debug** > **View Page Source HTML**

  The Page's HTML is displayed in a pop-up window.
- Viewing the current Page's underlying DOM

  Select **Debug** > **View Rendered Page DOM**

  The Page's DOM is displayed in the framework log.

Refreshing the current Page

When the framework is in Debug Mode, you can refresh the current Page. There are two levels of Refresh.

- **Debug** > **Refresh**

  Uses the Page's cached XSL transform object to regenerate the current HTML page. This is useful when developing and modifying non-XSL aspects of the Page, such as CSS, JavaScript and images.

- **Debug** > **Deep Refresh**

  Reloads the Page's XSL transform from disk and uses it to regenerate the current HTML page completely from scratch using no cached objects. This is useful when developing and modifying XSL transform files.

Updating framework zip files

When the framework is in Debug Mode, you can use the **Debug** menu to update the executing framework from a version you have customized in the SDK.

See "Customization" on page 10-151

- **Debug** > **Update Zip Files**

  After selecting this menu item, browse to and select the customized Keeper *data.zip* and/or *design.zip* files in the SDK. The zip file(s) for the currently executing Keeper are then replaced with the file(s) from the SDK. Restart the framework to use the new files.

Design Mode

Design Mode is a time-saving mode useful when developing or modifying an application's or the framework's *design files*. When in Design mode, you can modify design files for a running application and for the framework itself and see the changes without having to relaunch the application or the framework.

> **Note:** A **Refresh** or **Deep Refresh** of the Page is typically required to see changes reflected. See "Refreshing the current Page" on page 2-12.

Design files include:

- XSL files
- HTML files
- JavaScript files
- CSS files
- Message catalog files
- Image files

> **Note:** When modifying Java aspects of an application, Debug Mode can be a time-saver. See "Debug Mode" on page 2-10.

Where are the design files?

In normal operational mode (not Design Mode):

- An application's design files are in its *pepper/(application)/design.zip* archive file.

  Note: This is also true for the Keeper application. See "Keeper application" on page 2-10.

- Framework design files are in *pepper/common-resources.zip*.

  Note: If you are using a custom theme, framework design files are in that theme's equivalent to *common-resources.zip*. For example, you may have a theme whose design file is *myTheme-resources.zip*. For more information about custom themes, see "Customization" on page 10-151.

In Design mode, design archives are extracted in the running framework. The framework and its applications use the extracted design files instead of the files in the archive. Modifications to the extracted design files are reflected in the running application and framework, although a Debug Mode's **Refresh** or **Deep Refresh** may be required.

Note: *design.zip* contains some files that are not design files, such as *FactoryBuild.xml* and *PageTemplates.xml*, which are considered definition files. Modifications to these definition files are not reflected in the running application or framework.

When does extraction occur?

For Pepper Application Framework design archives, the extraction occurs when the Pepper Application Framework launches.

Pepper Application Framework design archives include:

- *common-resources.zip* (or its equivalent if you are using a custom theme).
- The Keeper application's *design.zip.*

Each application's design archive (its *design.zip*) is extracted when the application launches.

Design Mode overwrite

Design Mode can be launched in overwrite mode and in non-overwrite mode.

The two modes are used at different stages of design and development.

- Non-overwrite mode

  In non-overwrite mode, design files in previously extracted design archives are not overwritten when design archives are extracted. This enables you to continue to accumulate changes to design files through framework and application relaunches in Design mode. Use this mode when doing ongoing development of design files. This is the default mode.

- Overwrite mode

  In overwrite mode, design files in previously extracted design archives are overwritten by design files that are extracted. This enables you to start designing from built files pulled from the design archives. Take care with overwrite mode, because you are not prompted before files are overwritten, and any changes you have made to extracted design files are lost. Use this mode when first starting a design session.

Overwrite mode is set when configuring Design Mode launch options. See "Setting framework system properties" on page 2-15.

Design Mode caution: don't lose your changes

After completing your designing, it's important to remember to copy the design files you've modified back into the application's SDK directory so that your changes are incorporated into future application builds.

Starting and stopping the framework

You can start, stop and restart the Pepper Application Framework on a Pepper device from the command line in an Xterm window, as follows.

Procedure:

1. Open an Xterm window by pressing **ctrl** + **shift** + **1**.
2. To start the framework, enter the following command:

   ```
   service pepper start
   ```
3. To stop the framework, enter the following command:

   ```
   service pepper stop
   ```
4. To restart the framework, enter the following command:

   ```
   service pepper restart
   ```

   Or type the following at any time (not in an Xterm window):

   ctrl + (shft) + **k**

Procedure complete

Setting framework system properties

The Pepper Application Framework launches in various modes. For example, you can launch the framework in Design Mode, in Debug Mode, and with a custom theme.

These are controlled by Java system properties ("-D" arguments) that are passed to the framework at launch time. The system properties are contained in a script file that is read during the launch process.

The system properties file and the syntax for entering the -D arguments differs by platform.

For information about configuring system properties, see "Configuring system properties" on page 2-17.

Configuring system properties

The following table shows the applicable system properties and explains their values, options, and meanings.

Table 2–1 Pepper Application Framework Java system properties

System property	Value	Description/Sample	Default value
-Ddebug=(*boolean*) See "Debug Mode" on page 2-10.	true	**Description** Enables Debug Mode. **Sample** -Ddebug=true	false
	false	**Description** Disables Debug Mode. **Sample** -Ddebug=false	
-Ddesign.mode=(*boolean*) See "Design Mode" on page 2-13.	true	**Description** Enables Design Mode. **Sample** -Design.mode=true	false
	false	**Description** Disables Design Mode. **Sample** -Design.mode=false	
-Ddesign.mode.overwrite=(*boolean*) See "Design Mode" on page 2-13.	true	**Description** Causes overwrite of theme files from current theme archive when launching in Design Mode. (This system property has no effect when launching with -Ddesign.mode=false.) Useful when starting theme development from an archive, not from previously extracted files. **Sample** -Ddesign.mode.overwrite=true	false
	false	**Description** Causes previously extracted theme files not to be overwritten when launching in Design Mode. This is the usual mode. It is useful when a Design Mode session extends through framework launch cycles. **Sample** -Ddesign.mode.overwrite=false	
-Dtheme=(*themeName*) See "Custom themes" on page 10-167.	(*themeName*)	**Description** Sets the theme by theme name. **Sample** -Dtheme=common	common

Table 2–1 Pepper Application Framework Java system properties (continued)

System property	Value	Description/Sample	Default value
-Duser.language=(*language*) See "How to localize for different languages" on page 10-177. *	Two-letter Java language code	**Description** Sets the language through Java localization. Used when supporting multiple languages. **Sample** -Duser.language=en	en
-Duser.region=(*region*) See "How to localize for different languages" on page 10-177.	Two-letter Java region code	**Description** Sets the region through Java localization. Used when supporting multiple regions. **Sample** -Duser.region=US	US
* On some Pepper devices you set the language by editing */etc/pup/pepperlang.conf*.			

Configuring system properties

The system properties file differs by platform, as explained in the following sections.

Setting system properties on Windows

This procedure explains how to configure Pepper Application Framework system properties on Windows.

On Windows, the launch resource file is:

...\My Documents\Pepper\Pepper.lax

Procedure:

1. Make a back up copy of *Pepper.lax*.
2. Open *My Documents\Pepper\Pepper.lax* for editing.
3. Find the following line of text:

```
lax.nl.java.option.additional=-Xbootclasspath/p:xalan.jar;xercesImpl.jar;xml-apis.jar -Dtheme=common -Dpepper.debug=false -Ddesign.mode=false -Ddesign.mode.overwrite=false
```

4. Modify the appropriate arguments as necessary.

   For detailed descriptions of system properties, see "Configuring system properties" on page 2-15.

5. Save and close the file.

   The modified launch settings become active on the next Pepper Application Framework launch.

Procedure complete.

Setting system properties on a Pepper device

This procedure explains how to configure Keeper system properties on a Pepper device.

On a Pepper device, the launch script is:

/etc/init.d/pepper

Procedure:

1. Make a back up copy of the launch script.
2. Open the launch script for editing.

   One way to do this is opening an Xterm window (**ctrl** + **shift** + **1**) then using vi or an editor of your choice.
3. Find the `PEPPEROPTIONS` section.

```
The precise text may vary between Pepper devices.
PEPPEROPTIONS="-Xms50m -Xmx150m \
    -Dsun.java2d.pmoffscreen=true \
    -DJREX_DEBUG=false \
    -Duser.path=$PATH \
    -Duser.display=$DISPLAY \
    -Dswing.aatext=true \
    -Djava.class.path=$PEPPER_HOME:$PEPPER_HOME/pepper.jar \
    -Djava.library.path=/lib:/usr/lib:$PEPPER_HOME/jrex_gre:$PEPPER_HOM$
    -Dpepper.home=$PEPPER_HOME \
    -Dpepper.security \
    -Djava.security.policy==$PEPPER_HOME/pepper.policy \
    -Djava.security.debug=failure \
    -Dtheme=common \
    -Dpepper.debug=false \
    -Ddesign.mode=false \
    -Ddesign.mode.overwrite=false \
    -Djava.util.prefs.syncInterval=2000000"
```

4. Modify the appropriate arguments as necessary.

   For detailed descriptions of system properties, see “Configuring system properties” on page 2-15.
5. Ensure every line for every argument except for the last argument line ends in:

   ” \” (a space plus the backslash character).

   ” \” indicates the command is continuing on the next line.

   Note: Failure to do this causes incorrect command line arguments and unexpected results.
6. Save and close the file.

   The modified launch settings become active on the next Keeper launch.

Procedure complete.

Keeper event log

This section explains the Keeper event log.

Overview

The Keeper automatically writes important events and information to a log file. A new log file is created each time the Keeper launches. Log files are text files you can read with a text editor. Log files reside in */opt/pepper/Logs* on the Pepper devices and *My Documents/Pepper/Logs* on Windows. The directory contains the current log file (the log associated with the currently executing Keeper) and all closed log files associated with previous Keeper sessions.

The current log file is named:

- *LATEST-0.log* on Windows and most Pepper devices.
- *LATEST.log* on Pepper Pad 2.

Previous log file names indicate the date and time at which they were closed. The log file is closed when the Keeper shuts down.

Your Java code can write events and information to the log file. For information on using Java to write to the Keeper log file, see “Event notification” on page 4-77.

> **Tip:** Checking the log file is a good first place to start when debugging your application during development. Many Keeper and application errors are written to the log.

Viewing the log

You can view the Keeper log file in two ways:

- Open the log file with any text editor.

  For information about log file location and file name, see “Overview” on page 2-19.
- When the Keeper is in Debug Mode, you can display the current log file directly as follows:

  Debug > **Show Log File**

Key combinations

Table 2–2 shows useful key combinations.

Table 2–2 Useful key combinations

Key combination	Description	Platforms
ctrl + (shift) + **1**	Launches an Xterm window.	Pepper devices
ctrl + (shift) + **0**	Toggles the Keeper between Debug Mode and normal mode. Debug Mode provides a number of functions that are required for application development, including adding an application to the Keeper, viewing the current page's HTML source, XML, or DOM tree, displaying log files, and more.	Pepper devices, Pepper Desktop
ctrl + (shift) + **k**	Restarts the Keeper.	Pepper devices
ctrl + (tab key)	Invokes the window manager's built-in application switcher. It can be used to toggle between any number of windows, and works best when using an external USB or bluetooth keyboard.	Pepper devices

Javadoc

Javadoc is used for two purposes:

- To document the SDK API — see "Javadoc of the SDK API" on page 2-20
- To document your own Java code

Javadoc of the SDK API

Javadoc documentation of the SDK's Java API is provided with the SDK.

Extracting SDK API javadoc

Before you can use the SDK API javadoc, you have to extract it, as follows:.

Procedure:

View the SDK javadoc documentation, as follows:

1. Browse to:

   pepper-sdk/doc

2. Extract *javadoc.zip*

Procedure complete

Javadoc of your application

Javadoc HTML documentation of your application's Java is automatically created from source file javadoc comments when the application is built with the **`ant`** command.

Note: For information about Ant and building applications, see “Building Applications” on page 11-189.

View the javadoc documentation of an application, as follows:

Procedure:

1. Browse to:

   pepper-sdk/applications/(yourApplication)/javadoc

2. Launch the javadoc by clicking an HTML file, for example: *index.html*

Procedure complete

3

User Interface

Anatomy of the user interface

While there is considerable flexibility in the design of an application's user interface, its basic components are:

- *Tabs,* called *Sections* in the code
- *SectionPages* and *Pages*
- *ToolBars*

From the user interface perspective, an application consists of one or more Tabs, where:

- Each Tab has one or more Pages.
- Each Page typically has a toolbar with buttons for user-initiated actions.

Figure 3–1 on page 3-24 shows these and other user interface items for the Journal application.

Table 3–1 on page 3-24 describes each identified item.

Figure 3–1 Application user interface components

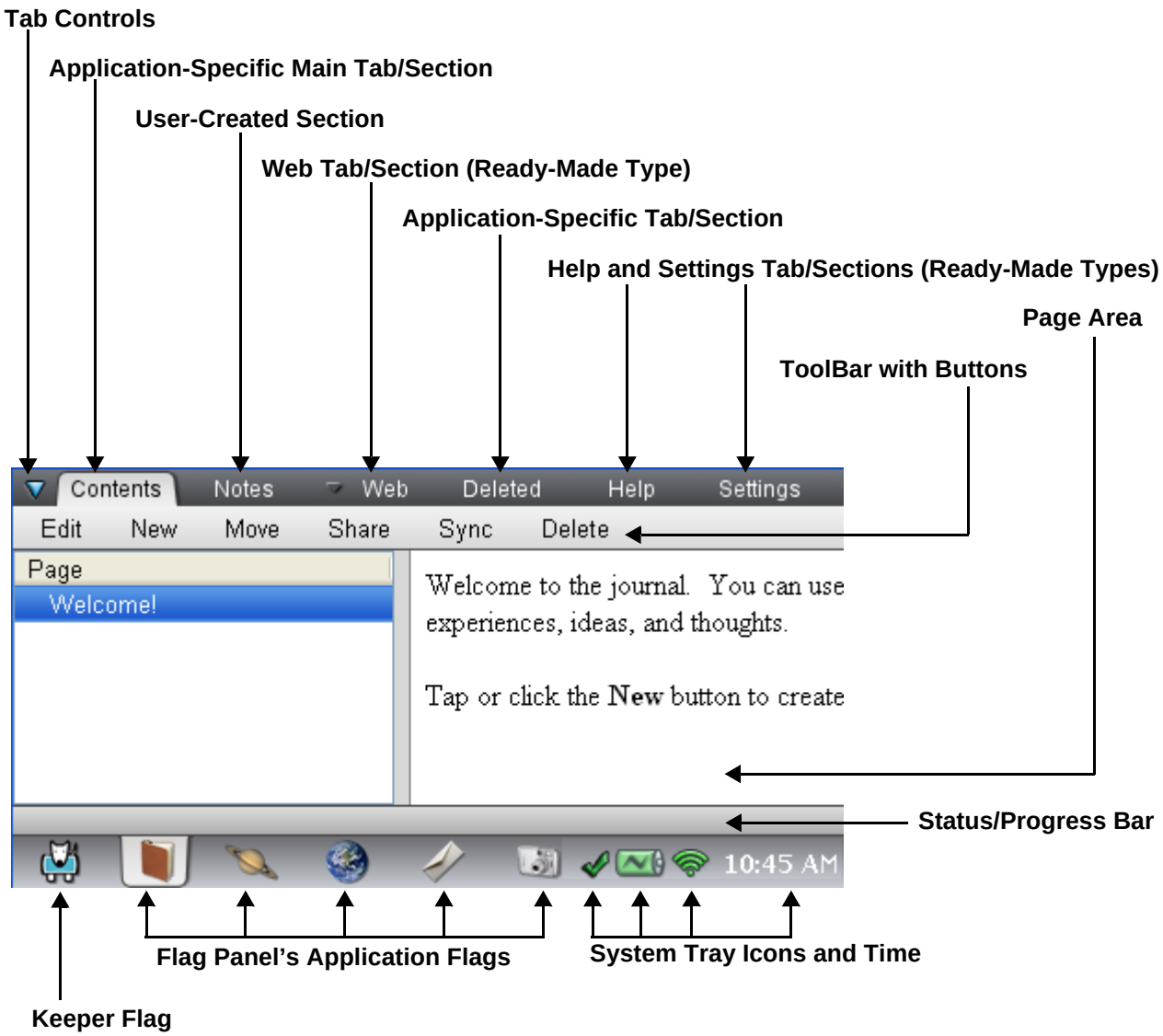


Table 3–1 Description of application user interface components

Component	Description
Tab Controls	Applications can enable or disable Tab controls. When Tab controls are enabled, the user can create new instances of Sections that allow it, rename them, move them, and delete them.
Application-Specific Main Tab/Section	Each application has a main Section that is instantiated as the left-most Tab. This Tab has the focus by default at application launch.

*Table 3–1 **Description of application user interface components** (continued)*

Component	Description
User-Created Section	Applications can allow creation of new Sections. The type of new Section can be determined programatically. Without explicit steps, the default Section type is created.
Web Tab/Section (Ready-Made Type)	You can easily include three types of ready made Tabs into your application: Web, Help and Settings.
Application-Specific Tab/Section	Applications can have as many types of Section/Tab types as required.
Help and Settings Tab/Sections (Ready-Made Types)	You can easily include three types of ready made Tabs into your application: Web, Help and Settings.
Application Page Area	The user's working area of the application. Each Section has a default Page (the SectionPage). The application can be developed to permit the Tab to subsequently display additional Pages.
Application ToolBar	Applications can optionally provide a toolbar. The toolbar provides access to ready-made and developer-created actions implemented in Java or JavaScript.
Status/Progress Bar	Displays messages from the framework and from the current application. For information about writing to the Status Bar, see "Event notification" on page 4-77. Also displays a progress bar as dictated by application logic.
Flag Panel	Displays icons ("Flags") for active applications.
Keeper Flag	The Flag for the Keeper, which is always visible.
System Tray	Displays system oriented icons such as battery and WiFi. Note that the icons displayed vary depending on the Pepper platform.
Application Flags	On framework start-up, each application that was executing at the time of framework shutdown has an application Flag in the System Tray. Click the Flag to launch the application. During execution, a Flag displays for each executing applications. Click a Flag to switch applications.

Terminology: programmatic and user interface

As noted previously, there are sometimes two terms for the same thing. One term describes a programmatic object, the other an instance of the object in the user interface. Table 3–2 on page 3-26 shows some of these.

Table 3–2 Terminology

User interface term	Programmatic term
Application	Package
Tab	Section
Page	SectionPage or Page

4

Framework and Application Architecture

Framework overview

The Pepper Application Framework is the primary user interface on Pepper devices (when you turn on a Pepper device, you see only the framework). The Pepper Application Framework also executes as a separate application on Windows.

The framework is an integrated application execution environment based on Java, XML, XSL, HTML, CSS and JavaScript.

Let's start with two high level definitions:

- The *Pepper Application Framework*

  The framework provides an environment in which applications execute.

  As a container for applications, the framework manages each application through its life cycle, providing, for example, user control over application starting and stopping. The framework also provides a set of services that are accessed by applications through Java classes and methods, such as writing to the Status Bar and the framework log, creating, reading and writing Sections and Pages, and accessing the browser, the browser search field, and URLs.

- *Pepper application*

  An entity that runs within the Keeper environment.

  From a programming perspective, it is a Java Object that extends com.pepper.platform.program.AbstractPepperProgram.

  It is by extending AbstractPepperProgram that applications exist as framework applications and progress through their framework life cycle. As extensions of AbstractPepperProgram, each application gains access to a wide range of classes and methods that enable programmatic interaction with framework services as discussed previously.

The following figure provides a diagram of the framework with applications and services.

Table 4–1 on page 4-28 provides information about items in the figure.

Figure 4–1 The framework

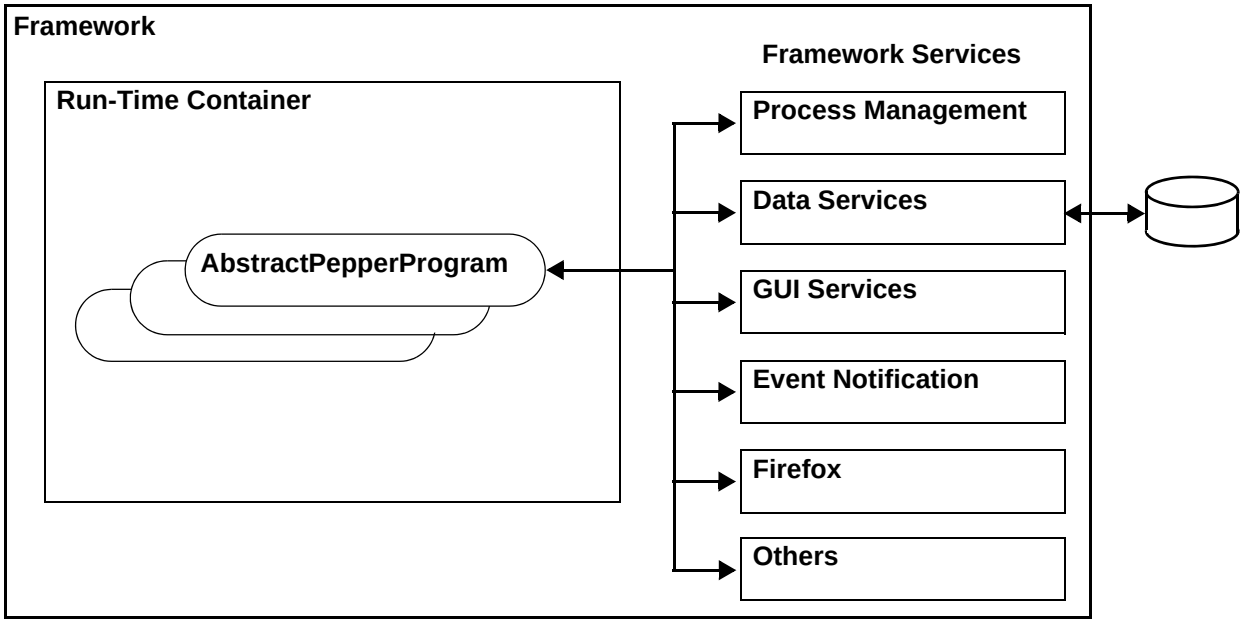


Table 4–1

Item	Description
Process Management	The application life cycle, focus switching between applications
Data services	Includes auto-saving form data to XML, Pages and Sections, Logging
GUI services	Includes the Status Bar, System Tray, Flag Panel, progress bars, and the browser
Event notification	Includes interfaces whose methods you can implement to react programatically to various events such as a Page being modified (PageChangeListener) and a Section gaining focus (SectionChangeListener)
Firefox	Integrated full-featured Firefox browser for displaying content. Supports JavaScript, LiveConnect and plugins for application enhancement.
Others	Other framework services include the LiveConnect JavaScript-to-Java bridge to access Java Actions from HTML pages, mime-type handling, and message catalogues for language localization

Application files and the distribution package

An application consists of Java code, data and design files. Many of these are created during development. The build creates additional files and the application's distribution package. The

distribution package consists of archive files (containing the application's Java code, data and design files) and the *package.ppld* file. When you install an application, its distribution package is copied over. As the application is used and its files are modified, they are copied from the archive onto disk and modified.

Java code

Java plays a fundamental role in Pepper applications. For example, every application requires a base Java class that extends com.pepper.platform.program.AbstractPepperProgram.

You can create Action classes that are executed from HTML pages, typically from a JavaScript method. You can also interact with the framework programmatically to directly access and create Sections and Pages, respond to events, display messages on the Status bar. You can also create a Java user interface instead of using HTML.

AbstractPepperProgram life cycle

The application base class is specified in the *package.ppld* file. This class is the entry point for application execution. Once executing, applications have a life cycle that is reflected in the base class's instances of key AbstractPepperProgram methods.

Specifying the base class

To launch an application, the framework needs to know how to find the base class, that is, the class that extends AbstractPepperProgram.

You configure this in the application's *package.ppld* file. The `<application-desc>` element's `main-class` attribute specifies the base class, including its package path.

For example, if the base class is named HelloWorld, and it exists in the following package path: *com.pepper.HW*, then the *package.ppld* must have the following:

```
<application-desc main-class="com.pepper.HW.HelloWorld">
   ...
</application-desc>
```

Application initialization

Initialization is embodied in the class constructor and its `init()` method.

The initialization begins with a call to the class constructor. The framework passes the constructor an process ID. The constructor must pass the process ID to the super class, as follows:

```
public HelloWorld(Integer pid) {
   super(pid);
}
```

Then, the application `init()` method is called and passed a PepperProgramConfig object.

In general, this method is where Actions are registered, the user interface is initialized, and any other initialization tasks are performed.

See "Java Actions" on page 4-66.

However the first thing your `init()` method must do is call `supt.init(conig)` to pass the program's configuration to the parent class, which ensures proper framework initialization occurs, as follows:

```
public void init(PepperProgramConfig config)
   throws PepperProgramException {
   super.init(config);
   //Action registration
   //User interface and other initization
}
```

Application destruction

When the application terminates, its `destroy()` method is called. Use this method to perform any application clean-up tasks.

Application data

As noted, an application's data is stored in XML Page files.

Pages provided with an application reside in its *data.zip* file. These are either created during the build, created during program execution, or provided by the developer as pre-built Pages.

See "Pre-built Pages" on page 4-69.

If any Pages included with the application distribution are modified at run time, the framework copies the Page to the *data* directory and modifies the copy appropriately. Once a Page exists in *data*, the equivalent page in *data.zip*, if any, is ignored.

The framework provides ways to connect HTML form fields to the underlying Page's XML elements. This enables you to automatically populate the fields with Page data and to automatically save the field data into the Page.

See "The Section user interface" on page 4-44.

You can also save data from Java Sections into Page XML files.

See "Hello World 4: Advanced Java" on page 9-135.

The framework is a *Page factory* that builds new instances of Pages as they are needed by an application based on its Page definition files. For example, the Journal application enables you to create new Journal entries, each of which is a new Page. To create a Page, the framework creates a new XML file based on its definition.

As noted, you can also provide Pages that come with an application and are available at application first launch. Default web bookmarks are an example. These are called *pre-built* Pages. Such Pages already have an XML instance file that was created by the developer and included by the application build in its distribution.

Design

Files used to control the application's design and styling are contained in the application's *design* directory in the SDK. This category includes a number of items that may not be considered *design*, strictly speaking. However, the key point is that the *design* directory contains all application specific files except for Java source files and pre-built Pages, including:

- Application declaration and definition files such as *FactoryBuild.xml*, *PageTemplates.xml*, *package.ppld* and *PackageStrings.properties*
- XSL transforms

- JavaScript files to include in your HTML
- Application-specific CSS files
- Image files (typically stored in a subdirectory: *design/images*)

When the application is built, all of these files are bundled into the *design.zip* file, which is included in the application's distribution package, as discussed below.

The following sections introduce the key application definition and declaration files.

Application definition: *package.ppld*

Each application's highest level characteristics are defined in its *package.ppld* file. This file resides in the application's *design* directory.

package.ppld specifies such things as: the application's unique type, its displayed title, icons used to represent it in the framework, required resources (such as jar files and archives), the application's base class name and location, mime type registration, and so on.

Page definition: *PageTemplates.xml*

Every Page type is defined in the application's *PageTemplates.xml* file. This file resides in the application's *design* directory. When you define the Page, you specify a set of key information, such as how it is to be displayed, its XML structure for holding Page data, and whether the framework should apply *caching rules* (explained below) to the Page.

Initial Section instances: *FactoryBuild.xml*

During an application build, one Section instance is created for each `<section>` element in *FactoryBuild.xml*. Section instances can be created at build time based on a definition in *PageTemplates.xml*, or they can be pre-built by the developer.

Message definition: *PackageStrings.properties*

PackageStrings.properties is Java properties file (also called a *message catalog*). It contains key-value pairs used to load text displayed in the application, for example on Tabs, buttons, and other user interface widgets. This mechanism supports Java localization by region and language.

See "How to localize for different languages" on page 10-177.

This file resides in the application's *design* directory.

Application distribution package

The build creates the application's distribution package.

See "Building Applications" on page 11-189.

The distribution package consists of the following:

- *data.zip*

  Contains all Page files created by the build or provided as pre-built Pages

See “Pre-built Pages” on page 4-69.

- *design.zip*

  Contains all files in the application’s SDK *design* directory (and subdirectories, if any)

- The application jar file

  See “How Java is Compiled and Jarred During the Build” on page 11-193.

- Any other jar files or resources you application needs

  See “Adding Existing Jar Files to the Build” on page 11-194.

- *package.ppld* file

  While this is a design file and does exist in *design.zip*, it is broken out as an individual file in the distribution package. This is the key file through which you add the application to a framework and update an application from an update server.

  See “Adding and Distributing Applications” on page 12-195.

Sections and Pages

Let’s take a closer look at an application as a dynamic structure of *Sections* and a Section as a dynamic structure of *Pages*.

We first describe Sections and Pages conceptually with respect to their Java and XML aspects. Then, we examine how the dynamic state of the hierarchical structure is maintained by the framework in XML files. Then, we provide a list of helpful Java methods for accessing and manipulating Sections and Pages.

Pages

A Page has two general aspects:

- A Page is the basic unit for containing related pieces of application data.

  Page data is stored together in an XML file on disk.

  Page data is generally displayed together.

- A Page is a Java interface (and implementing subclasses) through which you programmatically load, edit, and store data.

So, the root concept of a Page is that it is the wrapper for data storage and the means to use it programatically.

The framework itself stores data on disk in Pages. For example, the dynamic structure of applications (its Sections and Pages) are stored by the framework in the *package.xml* Page file. There’s also a framework Page file used to store the current list of applications.

Applications use Pages too. Pages enable you to organize the Section’s user interface and data into effective chunks. By containing and organizing Pages, Sections provide a high level of organization within an application.

Every Section has:

- One instance of a special kind of Page that anchors every Section: a com.pepper.platform.page.SectionPage.
- Any number of additional Pages (objects that implement com.pepper.platform.pagePage interface).

  Note: The SectionPage class extends a class that implements the Page interface, so the SectionPage is a type of *Page*. This can lead to some difficulties with terminology. In this guide, the term *SectionPage* generally refers to the special Page that anchors every Section. The term *Page* generally refers to non-SectionPages (the other Pages of which a Section may be composed). When the Page *interface* is the topic, explicit steps are taken to make that clear.

The Page interface provides methods you can use for all Page objects to get, modify and save the Page's data. For example, every Page has a *Page ID* and exists in a Section with a *Section ID*. You often use these arguments to identify and access particular Pages using Page interface methods.

The Page interface (and therefore its implementing classes) also provides methods that give you access to the underlying Page data. This data is represented programmatically as a JDOM Document (org.jdom.Document). You use the Page's JDOM Document object to load, edit, and save Page data. The framework automatically saves the JDOM Document to disk in a page XML file (more on this below).

Note: JDOM is a Java API for manipulating XML provided with the SDK.

What does a Page consist of?

- A Page has a *definition*.

  Pages are defined in the application's *PageTemplates.xml* file with either the `<sectionPage>` or the `<basePage>` element. This definition is used by the framework when creating new Pages, either during the application build or dynamically at run-time.

  For reference information about elements in *PageTemplates.xml* and the other application definition and declaration files, see "XML Reference" on page B-207.

  The Page definition contains static information about the Page. For example, it includes a `<template>` element that specifies the XSL transform used to generate the HTML for the Page (or the XUL file). Or, the `<template>` element can specify a local HTML or a URL to an HTML file to display for the Page.

  Note: The framework supports XUL for displaying Pages, but this document covers only HTML.

  The Page definition also defines the *caching rules.* Caching rules exist only in `<sectionPage>` definitions. Caching rules determine the information the framework is to automatically cache from Pages into the SectionPage, if any. For example, in the Hello World Tutorial application, the "worlds" SectionPage contains the current list of worlds the user has created. Data about each "world" are contained in world Pages.

  The Page definition also defines the optional additional XML element structure of the Page. The XML elements and their attributes are used to store the Page's data on disk (described below).

- A Page has an object.

  As noted, Pages are instances of any class that implements the framework com.pepper.platform.page.Page interface.

Note: For non-SectionPages (generally, just called *Pages*), you usually use methods that return a Page, but you don't actually know the object type. But you do know that it implements the Page interface and therefore can use its methods. This approach keeps the framework generic and easily extensible.

- A Page has an XML file.

  Every Page has an XML instance file that contains the Page's data (both meta data necessary for the framework and data inserted as a result of application logic and user action). You access the Page data through its JDOM document object, as described next.

- A Page's data is represented by a JDOM Document object.

  The Page's data has two aspects. It is stored persistently in the Page's XML file. However, the data is accessed and managed programmatically through a JDOM Document object. The Document object represents everything inside the Page's `<body>` element, not the complete XML file. This part of the file is considered the *Page data*.

  The element structure (including the root element) inside the JDOM Document object varies depending on whether it is for a SectionPage or non-SectionPage.

  The JDOM document object for a SectionPage always has a root `<section>` element. The JDOM document object for a non-SectionPage always has a root `<page>` element.

  Tip: Understanding the root element of the JDOM document for each of the Page types is essential when accessing or manipulating data programatically from Java and XSL.

The following figure illustrates key aspects of the generic Page concept.

Figure 4–3 shows the differences in the data portion of a SectionPage and a non-SectionPage.

Figure 4–2 Page concept and structure

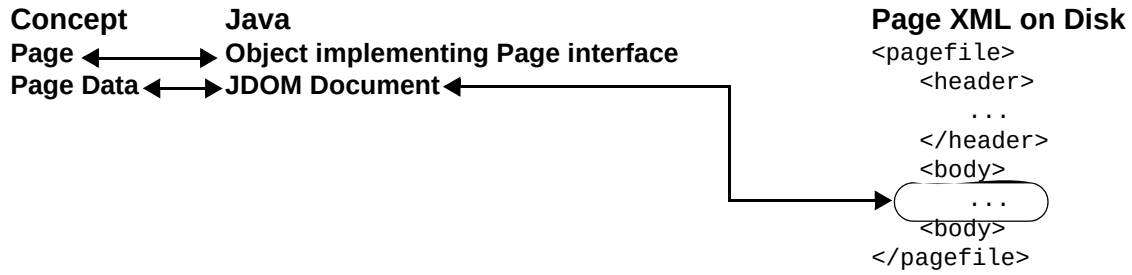
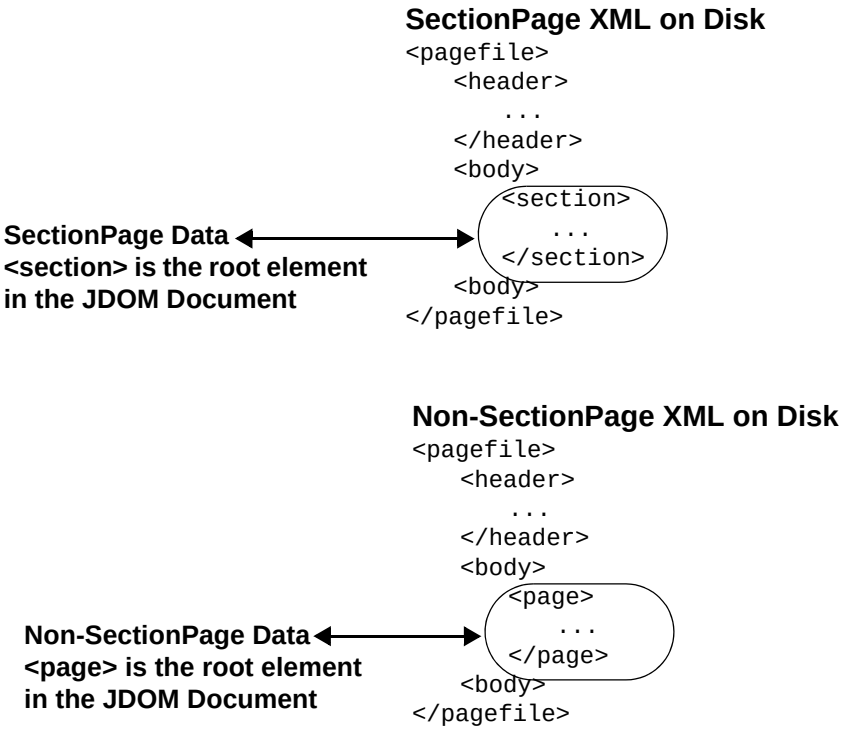


Figure 4–3 SectionPage and non-SectionPage concept



Creating Pages

Page XML files come into existence in two ways:

- Created by the framework during the application build or at run time based on the Section's definition in *PageTemplates.xml*

  Each Page definition has a type. When the new Page's type is specified, the appropriate definition is used. Otherwise the definition whose type is "default" is used. Such a "default" type Page definition in *PageTemplates.xml* is required by the build.

  See "Useful Java methods for Sections and Pages" on page 4-44.

- Provided as a pre-built Page by the developer and included in the application distribution by the build

  See "Pre-built Pages" on page 4-69.

Sections

As noted, an application is divided into one or more *Sections*. The main purpose of Sections is to separate and present the different logical parts of an application and to organize the application's data storage (in Pages) at a high level.

Sections display as application *Tabs*. For example, the Keeper application has three Sections: **Applications**, **Help** and **Settings**, each of which displays as a Tab.

Each Section presents a user interface and organizes and provides access to the SectionPage and optionally to additional Pages. That is, a Section can present a single Page (the SectionPage), and the SectionPage can provide access to any number of additional Pages.

There are two types of Sections: Default and Java. The two types share certain characteristics but also differ in key ways. The main difference is that Default Sections typically display as HTML (or XUL) and store their data into Page XML files. Java Sections present a Java user interface and can store data however they wish, including using XML Page files.

See "The Section user interface" on page 4-44.

What does a Section consist of?

- Every Section has a definition.

  An application's Sections are defined by the developer in the application's *PageTemplates.xml* file with `<sectionPage>` elements.

- Every Section has a com.pepper.platform.program.Section Object.

  Every Section instance exists as a Section object. Sections can be created and modified programmatically from Java using this object. The Section object provides access to the JDOM Document that represents a Section's data, and you can use it to access, modify and save the data persistently (to the SectionPage XML file).

- Every Section has a com.pepper.platform.page.SectionPage object that corresponds to the SectionPage XML file.

  Note: Typically one uses the Section object to access the Section data rather than the SectionPage object.

- Every Section has a SectionPage XML file.

  Every Section has an XML instance file that contains all of the Section's data on disk (both meta data necessary for the framework and data inserted as a result of application logic and user action).

Application structure — a run-time sample

Previously, we introduced the concept that Sections, SectionPages, and Pages are defined during development in *PageTemplates.xml*.

We noted that during program execution, Sections and Pages can be created and destroyed, depending on the application design, at run-time. We also noted that that Sections and Pages can be created during the application build and pre-built and provided with an application.

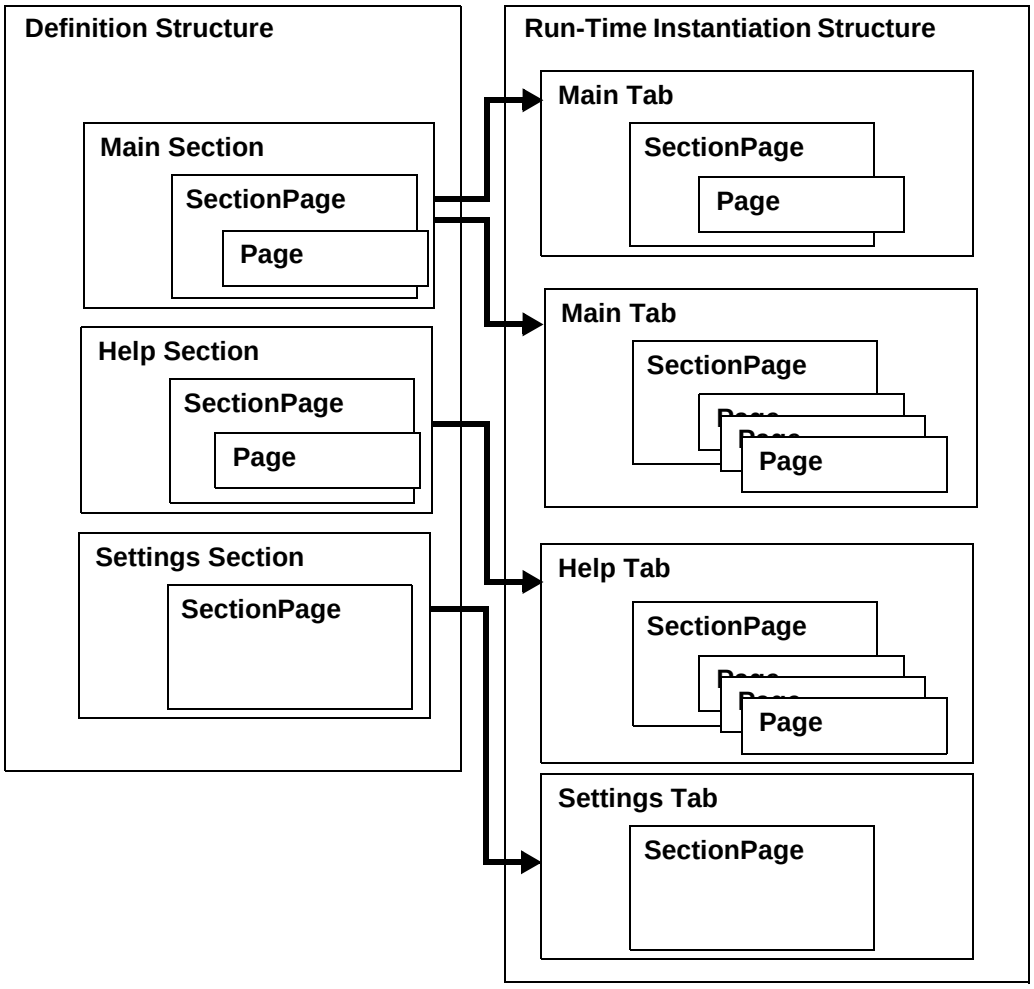
See "Pre-built Pages" on page 4-69.

What all of this amounts to is that the actual structure of Section and Page instances at run-time is dynamic, yet derived from the application's Section and Page definitions.

Note: Every Section requires a Section ID and every Page a Page ID. These indicate the SectionPage and Page XML file names and are used programatically to access them. You set these IDs for Sections created during the build with *FactoryBuild.xml*'s `<section>` element `id` attribute. The framework creates them automatically for run-time created SectionPage and Page instances based on the system time.

To illustrate this point, Figure 4–4 shows how an application's definitions (on the left) could result in a run-time structure, on the right.

Figure 4–4



Application structure from an XML perspective

In this section, we discuss the XML aspects of Sections and Pages.

For clarity, we start with the lowest level (Pages) and work upwards (to Sections):

- Page XML files

  Pages contain useful application and user entered data, as we have seen.

- SectionPage XML files

  SectionPages contain a dynamically maintained list of child Pages, and, as we have seen, optionally data cached upwards from child Pages.

- The application XML file (called *package.xml*)

  package.xml contains a dynamically maintained list of Sections, as we have seen. This material is discussed to better understand the framework, even though it is usually inappropriate to programatically control *package.xml*.

Page XML definition and instance

As noted, Pages have a definition in *PageTemplates.xml* and have XML file instances derived from that definition.

The following example shows several Page definitions taken from the Hello World Tutorial's *PageTemplates.xml* file. Discussion follows the example.

Example 4–1

```
<!-- Define world basePage -->
<basePage type="world">
   <template>design/world.xsl</template>
   <page>
      <worldName />
      <worldRadius />
      <yearLength />
      <dayLength />
      <distanceFromSun />
      <hasWater />
      <hasLife />
      <planVisit />
   </page>
</basePage>
```

Key points:

- Since this is an XML file, comments are XML style, starting with `<!--` and ending with `-->`.
- A non-SectionPage Page definition such as this is defined with a `<basePage>` element.
- The `<basePage>` element's `type` attribute indicates that this definition should be used when creating Pages of the type `world`.
- The `<template>` element specifies that the *world.xsl* XSL transform is used to generate the HTML for the Page.

  Such XSL files must reside in the application's *design* directory, and the `<template>` must specify the d*esign* directory and the particular XSL transform as shown. As noted previously, you could also specify an HTML file, a URL to an HTML file, or a XUL file.
- The `<page>` element wraps the Page data.

  Its child elements specify the XML structure of the data elements needed by the Page for data storage.

  In this case, there are eight child elements, each of which is populated by data entered by the user into an HTML form at run-time.

The following example shows an XML Page file created during application execution that is derived from the previous definition. That is, the following is an *XML instance file* for a dynamically created Page. Discussion follows the example.

Example 4–2

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile>
    <header>
```

```
        <pkgName>helloworld</pkgName>
        <pkgVersion>3.0.3</pkgVersion>
        <template>design/world.xsl</template>
        <noCache>0</noCache>
        <defaultPageType>default</defaultPageType>
        <createDate utc="1158074394101" />
    </header>
    <body>
        <page id="data/1158074394081" name="" type="world">
            <worldName>earth</worldName>
            <worldRadius>not sure</worldRadius>
            <yearLength>365</yearLength>
            <dayLength>24 and a little</dayLength>
            <distanceFromSun>90,000,000 miles or so</distanceFromSun>
            <hasWater>true</hasWater>
            <hasLife>true</hasLife>
            <planVisit>3</planVisit>
        </page>
    </body>
</pageFile>
```

Key points:

- The file is an XML file and therefore starts with this:

  `<?xml version="1.0" encoding="UTF-8"?>`

- The root `<pageFile>` element has two child elements, `<header>` and `<body>`.

  See "Page concept and structure" on page 4-34.

- `<header>` is generated by the framework and contains, among other things, the `<template>` element that indicates how to display the Page.

- `<body>` contains the `<page>` element and all its children as defined in Example 4–1.

- `<page>` has three attributes inserted by the framework when the page was created.

  `id="data/1157662197503"` specifies this Page filename and location. (`data/1157662197503` is the Page ID.) All Page XML files reside in the applications *data* directory (possibly in a subdirectory). The number `1157662197503` is this file's file name without its xml extension. Thus, this Page XML file resides in the following run-time location:

  (application)\data\1157662197503.xml

  As we will see, this Page ID attribute is critical in many respects. You can use it programmatically to access the Page from Java. It is also cached up to the SectionPage so that the SectionPage knows the file name of this child Page, as explained below.

  Note: A SectionPage's `id` attribute is called the *Section ID*.

  The `name` attribute is not used here, but you can use it to store a unique name for the Page.

  The `type="world"` attribute shows that this is a Page derived from the *PageTemplates.xml* definition whose `type` is `world`.

- The `<page>` element's child elements (`<worldName>` through `<planvisit>`) contain data entered by the user when using the Page.

  The first five such elements contain text.

Two contain `true`. These elements are represented as checkboxes in HTML and are `true` when checked and `false` otherwise.

The last, `<planVisit>`, is represented as three radio buttons in HTML. The number 3 in the XML file shows that the third radio button was checked.

The ability to store HTML data into the XML Page file is discussed in "The Section user interface" on page 4-44.

SectionPage XML definition and instance

This section shows a SectionPage definition and the resulting XML Page file.

Example 4–1 shows a SectionPage definition taken from the Hello World Tutorial. Discussion follows the example.

Example 4–3

```
<!-- Define Worlds sectionPage -->
<sectionPage type="worlds">
   <template>design/worlds.xsl</template>
   <defaultPageType>world</defaultPageType>
   <cacheRules match="page">
      <apply>worldName</apply>
   </cacheRules>
   <section />
</sectionPage>
```

Key points:

- A SectionPage definition is wrapped in a `<sectionPage>` element.
- The `<sectionPage>` element's `type` attribute indicates that this definition should be used when creating Pages of the type `worlds`.
- Again, the `<template>` element specifies either the XSL transform used to generate HTML for display or an HTML file (local or URL) to display directly for the Page.
- The `<defaulPageType>` element specifies the type of Page the framework should create when it is triggered to create a Page for the Section.

  In this case, the type is `world`, namely, the Page type we discussed above.
- The `<cacheRules>` element specifies the information the framework should dynamically retrieve from child Pages and cache into this SectionPage.

  In this case, the caching rules indicate the framework caches up `<worldName>` elements from each child Page. The XSL used to generate this SectionPage uses the cached elements to create a table displaying all worlds and showing the name of each.
- The `<section>` element wraps the SectionPage data.

  It is empty in this case. But it could contain an arbitrary structure of XML element to hold user-entered or other data, just as a Page definition uses a `<page>` element to contain such data.

Now, let's take a look at a sample SectionPage XML file created during program execution from this definition. Comments follow.

Example 4–4

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile>
   <header>
      <pkgName>helloworld</pkgName>
      <pkgVersion>3.0.3</pkgVersion>
      <template>design/worlds.xsl</template>
      <noCache>0</noCache>
      <defaultPageType>world</defaultPageType>
      <createDate utc="1157999625450" />
   </header>
   <body>
      <section name="NameKey.Worlds" type="worlds"
                id="data/Worlds" builtin="false">
         <page id="data/1158074394081" name="" type="world">
            <worldName>earth</worldName>
         /page>
         <page id="data/1158075623539" name="" type="world">
            <worldName>jupiter</worldName>
         /page>
      /section>
   </body>
</pageFile>
```

Key points:

- The overall structure is the same as that of the Page XML file.

  There's a root `<pageFile>` element with a child `<header>` that contains key static information derived from the definition file and a child `<body>` that contains dynamic information.

  See "SectionPage and non-SectionPage concept" on page 4-35.

- `<body>` has a `<section>` child, because this is the XML file for a SectionPage.
- `<section>` has the `name="NameKey.Worlds"` attribute.

  This is used by the framework look up a key (`NameKey.Worlds`) in a message catalog file (*PackageStrings.properties*) to obtain the text used to display as the Tab text, as explained below.

- The `<section>` element's `type` and `id` attributes serve the same purpose as for generic Page XML files, as explained previously.
- `<section>` wraps the SectionPage data.

  In this case `<section>` wraps two `<page>` elements, each with three attributes and each with a `<worldname>` child element.

  The `<page>` elements are cached here automatically by the framework as a result of the SectionPage's caching rules in its definition. In this case, the cached elements indicate there are currently two world Pages. The First world's `<worldName>` is `jupiter`, the second, `earth`. Each includes cached `id`, `name` and `type` attributes.

Application XML file

Now, that we see how a Section keeps track of dynamically changing Pages. In this section, we can take a look at how the application itself keeps track of dynamically changing Sections.

As noted above, Sections are defined in the application's *PageTemplates.xml* file with `<sectionPage>` elements. During the build, Section instances are created for each `<section>` element in *FactoryBuild.xml*. The dynamic list of current run-time Sections is maintained by the framework in *package.xml.* This file resides in the application's *data*/*data.zip* file if the current Sections have not changed since the time the application was first launched. If the current Sections have changed, the working *package.xml* file resides in the *data* directory.

> **Note:** This is the same approach used for all data XML files. If they have not been modified since first launch, they reside in the *data.zip* file. Otherwise, the working copy is in the *data* directory.

The following example shows the *FactoryBuild.xml* file of Phase Four of the Hello World Tutorial. Discussion and a sample instance of the corresponding *package.xml* file follows.

> **Note:** Comments and white space have been removed to save space.

Example 4–5

```
<?xml version="1.0" encoding="UTF-8"?>
<factoryBuild>
   <packageList>
      <section name="NameKey.HelloWorld" type="default"
               id="data/helloWorldMain" deletable="true" builtin="false" />
      <section name="NameKey.Worlds" type="worlds"
               id="data/Worlds" deletable="true" builtin="false" />
      <section name="NameKey.LogStatusbarJava" type="java"
               id="data/LogStatusbarJava" deletable="true"
               src="../prebuilt/LogStatusbarJava" />
      <section name="NameKey.WorldJavaSection" type="java"
               id="data/WorldJavaSection" deletable="true"
               src="../prebuilt/WorldJavaSection" />
      <section name="NameKey.WebSection" type="web"
               id="data/web" builtin="true"  >
         <prebuiltPagesDir>../prebuilt/websection</prebuiltPagesDir>
      </section>
      <section name="NameKey.Help" type="help" builtin="true"
               id="data/help/locale(Help)"
               src="../prebuilt/helpsection" />
      <section name="NameKey.Settings" type="settings"
               id="data/settings" builtin="true" />
   </packageList>
</factoryBuild>
```

Key points:

- This is an XML file whose root element is `<factoryBuild>.`
- The `<factoryBuild>` element has a child element, `<packageList>`, that in turn has child `<section>` elements.
- Seven Sections are declared with seven `<section>` elements.

- Each <section> element has attributes.

  Key attributes are as follows:

  - name specifies the key to look up in *PackageStrings.properties* to retrieve localizable text for display in the Section's Tab.
  - type specifies the Page definition in *PageTemplates.xml* to use when creating the Section's SectionPage.

    The type can be one of the pre-set values (default, java, web, help, settings) or a developer-defined type. In either case, there must be a Page with every used type in or included in the *PageTemplates.xml* file. See "<section>" on page B-219.
  - id specifies the SectionPage XML file name that is created for the Section at build time without the XML extension.
  - builtin, deletable and src are explained in "<section>" on page B-219.
- One <section> has a child <prebuiltPagesDir> element.

  This specifies a directory containing pre-built Pages. See "Pre-built Pages" on page 4-69.

The following example shows a run-time sample of the *package.xml* Page file created by the framework. Discussion follows.

Example 4–6

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile deletable="false" backup="true">
   header>
      <pkgName>helloworld</pkgName>
      <pkgVersion>1.0</pkgVersion>
      <template />
      <noCache>0</noCache>
      <defaultPageType>default</defaultPageType>
      <createDate utc="1157736443874" />
   </header>
   <body>
      <packageList>
         <section name="NameKey.HelloWorld" type="default"
                id="data/helloworldMain" builtin="false" />
         section name="NameKey.Worlds" type="worlds"
                id="data/Worlds" builtin="false" />
         <section name="NameKey.LogStatusbarJava" type="java"
                id="data/LogStatusbarJava" deletable="true"
                src="../prebuilt/LogStatusbarJava" />
         <section name="NameKey.WorldJavaSection" type="java"
                id="data/WorldJavaSection" deletable="true"
                src="../prebuilt/WorldJavaSection" />
         <section web="true" name="NameKey.WebSection" type="web"
                id="data/web" builtin="true" />
         <section name="NameKey.Help" type="help" builtin="true"
                id="data/help/locale(Help)" src="../prebuilt/helpsection" />
         <section name="NameKey.Settings" type="settings"
                id="data/settings" builtin="true" />
      /packageList>
   /body>
```

```
</pageFile>
```

Key points:

- Note the same high level XML structure as the other types of Page files, namely a <pageFile> root element containing a child <header> whose contents are static and a child <body> whose contents are dynamic.
- The <body> element contains a <section> element for every current Section instance.
- Each <section> contains attributes necessary to track and display the Section.

Useful Java methods for Sections and Pages

This section lists a few helpful Java methods you can use to access and manipulated Sections, Pages and Page data.

Note: For detailed information, see the framework javadoc.

To get a Section:

```
Section theSection = AbstractPepperProgram.getSection(String sectionId);
```

To get all Sections:

```
SectionList theSections = AbstractPepperProgram.getSections();
```

To get the SectionPage from the Section:

```
SectionPage sectionPage = theSection.getSectionPage();
```

To get all the Pages in a Section:

```
PageList list = theSection.getPages();
```

Three ways to get an individual Page from a Section:

```
Page page = theSection.getPage(int index)
Page page = theSection.getPages().getPage(String pageId);
AbstractPepperProgram.getPage(String pageId, String sectionId);
```

To create a Page:

```
Page page= AbstractPepperProgram.createNewPage();
```

The Section user interface

There are two choices when designing a Section's user interface:

- Default style

The default style uses XSL+HTML/CSS or just HTML/CSS. (XUL is also supported but is beyond the scope of this guide.) Pages are at the root of the default style. Each Page's `<template>` specifies the display mechanism, either an XSL transform that generates HTML, or a local HTML file or URL.

- Java style

  This is a Java Section. Java Sections enable you to create a Section with a Java user interface that optionally saves its data as Pages.

Defining the Section (and therefore the user interface) type

As we have seen, every Section type in an application that is created during the build or during application execution must be defined in *PageTemplates.xml* by wrapping the definition in a `<sectionPage>` element. (The definition may be included from an external XML file with an `<xi:include>` element.)

The Section type is determined by the value of the `<sectionPage>`'s `type` attribute.

- A Default style Section is created by defining your own `type` in the `<sectionPage>` definition.
- A Java style Section is created by specifying the Java type (with `type="java"`) in the `<sectionPage>` definition.

Other Section types (such as Web, Help, and Settings) are also supported.

See "Attribute: type" on page B-220.

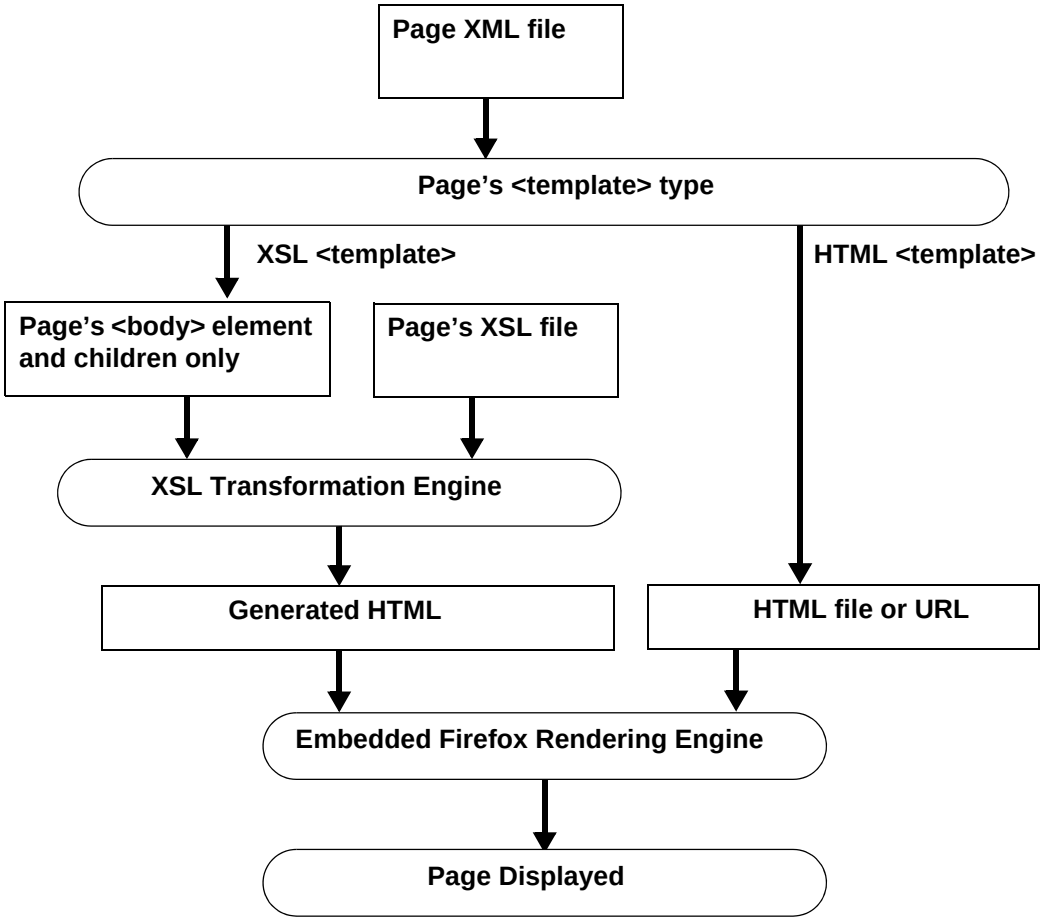
Default-style Section user interface

In a Default-style Section, each Page is the starting point for the user interface. There are two paths through which a Page XML file is displayed in the framework. The path taken depends on the type of file pointed to by the Page's `<template>` element.

- If the Page's `<template>` element specifies an XSL file, the Page's `<body>` element (and all it contains) are passed to the XSL transformation engine, which uses these to generate an HTML page that is rendered in the Section.
- If the Page's `<template>` element specifies another type of file, for example a local HTML file or a URL, the HTML is retrieved and rendered in the Section.

The following figure diagrams this process.

Figure 4–5 How Pages are displayed in the framework



HTML rendering with CSS stylesheets

You can use CSS stylesheets to control the visual styling of HTML, whether it is generated by XSL or not.

You can create your own CSS stylesheets or use the framework's stylesheet: *styles.css*.

> For information about the framework stylesheet, see "Getting started with CSS customization" on page 10-155.
>
> **Note:** If you use your own stylesheet instead of the framework stylesheet, it may be more difficult to implement visual styling customizations that apply to all applications.

As required by HTML, the CSS stylesheet has to be linked inside the HTML page using the HTML `<link>` tag. You can generate this during XSL transformation. Linking the stylesheet is required whether you use your own stylesheet or the framework stylesheet.

If you are using you own stylesheet, it must reside in the application's *design* directory in the SDK.

The framework stylesheet resides in current theme archive (which is *common-resource.zip,* unless you are using a custom archive, in which case you provide the name). Linking to the framework

stylesheet is always done the same way, regardless of whether you are using the standard theme archive or a custom theme archive.

For information about theme archives and customization, see "Customization" on page 10-151

The following example shows the XSL needed to generated HTML that links to an application-specific stylesheet named *myStyles.css* and that links to the framework stylesheet: *styles.css*.

Note: Linking the framework stylesheet involves using the `$platform` framework parameter that is passed to XSL transforms. For more information, see "Framework parameters passed to XSL" on page 4-68.

Example 4–7

```
<!-- add link to the framework main CSS stylesheet -->
<link href="{$platform}/resources/styles/styles.css"
                rel="stylesheet" type="text/css" />
<!-- add link to a user-defined CSS stylesheet -->
<link href="{$design}/design/myStyles.css"
                rel="stylesheet" type="text/css" />
```

Note: This example was taken from the Hello World Tutorial included with the SDK.

Generating Page toolbars

Each Page in a Section of the Default type typically includes a toolbar with one or more buttons.

See "Anatomy of the user interface" on page 3-23.

You can trigger framework and application-specific Java Actions from toolbar button clicks through LiveConnect.

See "Java Actions" on page 4-66.

See "JavaScript and Mozilla LiveConnect" on page 4-62.

Defining a toolbar

The following example shows how a toolbar is defined in Hello World's *worlds.xsl*, a SectionPage.

Example 4–8 Defining a toolbar and buttons in Hello World's worlds.xsl

```
<!--create the page toolbar -->
<pepper:pagebar class="PageBar" id="pb-{$packageId}" package="{$packageId}">
   <!-- create toolbar buttons -->
   <pepper:pagebarentry key="PageBar.New" action="NewPage" param="world" />
   <pepper:pagebarentry key="PageBar.Edit" script="showSelectedPage()" />
   <pepper:pagebarentry key="Label.EditJavaAction" script="editJava()" />
   <pepper:pagebarentry key="PageBar.Delete" script="deleteSelectedWorld()" />
</pepper:pagebar>
```

Key points:

- The `$packageId` framework parameter is used to initialize the toolbar.
- Four buttons are created with four `<pepper:pagebarentry>` elements.

- Each button has a label defined by the `key` attribute, which specifies a key-value pair that must exist in the application's *PackageStrings.properties* file.
- The first button directly calls the NewPage Action using the `action=NewPage` attribute and passes the Action a parameter using the `param=world` attribute.

  Actions are referred to using a String ("`NewPage`" in this case) that uniquely identifies the Action. This is called the *Action name*. This name is set programmatically in Java when you register the Action in the application base class. For information on Action names, see "Java Actions" on page 4-66.
- The last three buttons call JavaScript functions using the `script="function()"` attribute.

  Such JavaScript functions must reside in a file that is included by the HTML or in the *pepper-sdk/lib/common-resources.zip* file.

  See "JavaScript and Mozilla LiveConnect" on page 4-62.

  Phase Two of the Hello World Tutorial shows how to use XSL to include into a generated HTML file an external JavaScript file that contains functions accessed from toolbar buttons. See "Toolbar buttons link to JavaScript and framework Actions" on page 7-113.

A toolbar example

Now that we have seen how to create a toolbar with buttons in XSL, let's see how that toolbar looks in HTML after XSL transformation through the framework.

Example 4–9 An HTML toolbar generated by XSL

```
<body>
   <div class="PageBar" id="pagebar-HelloWorldTutorial-0">
      <div class="PageBarEntry" onmouseup="bridge.action('NewPage', 'world');">New</div>
      <div class="PageBarEntry" onmouseup="showSelectedPage()">Edit</div>
      <div class="PageBarEntry" onmouseup="editJava()">Edit Java</div>
      <div class="PageBarEntry" onmouseup="deleteSelectedWorld()">Delete</div>
   </div>
...
```

Key points:

- The toolbar and the buttons are all `<div>` elements (HTML *tags*).
- Each `<div>` element is visually styled according to its CSS `class`.
- All button clicks are detected with the `onmouseup` attribute.
- `onmouseup` either directly calls an Action (and passes the specified parameter) or the specified JavaScript function.

Toolbar visual styling

After XSL transformation into HTML, the toolbar's visual styling is determined by the classes assigned to its constituent HTML elements in combination with the classes' definitions in the relevant CSS stylesheet.

Example 4–8 shows XSL that creates a toolbar and buttons. The toolbar is defined with a `class="PageBar"` attribute, which refers to a CSS `Pagebar` class for visual styling. As with most CSS framework classes, `PageBar` is defined in *styles.css*, which is in:

pepper-sdk/lib/common-resources.zip/resources/styles

The buttons are automatically generated by the framework with an HTML `class` of `PageBarEntry`, which is also defined in the stated CSS stylesheet.

For information about visual styling and *styles.css*, see "Customization with CSS" on page 10-154.

Java Sections

A Java Section enables you to present a Java user interface in a Section instead of an HTML interface.

This section explains how to create Java Sections. It also covers how to add a toolbar with buttons to the Java Section in a manner that enables its visual styling to be controlled from a single framework CSS stylesheet (*keeper.css*) that also controls other Java Section toolbars, thus facilitating sweeping visual styling customizations.

Note: Two examples of Java Sections are provided in the Hello World Tutorial.

See "Hello World 3: Getting Started with Java" on page 8-123 and "Hello World 4: Advanced Java" on page 9-135.

Java Section overview

Making a Java Section requires the following:

- "Including SectionJava.xml in PageTemplates.xml" on page 4-49
- "Declaring the Java Section instance" on page 4-49
- "Creating the pre-built Page and specifying the Java class" on page 4-50
- "The Java Section class" on page 4-50

Including *SectionJava.xml* in *PageTemplates.xml*

If your application has any Java Sections, you must include *SectionJava.xml* in your *PageTemplates.xml* file, as follows:

```
<xi:include href="../resources/pages/SectionJava.xml" />
```

Note: *SectionJava.xml* (and all other items in *resources*) are in *pepper-sdk/lib/common-resources.zip,* or the current them archive. You do not need to extract the archive. You only have to refer to the item you want to include as demonstrated here.

Declaring the Java Section instance

If the Java Section instance is to be present at first launch (as opposed to being created programatically at run time), it must be declared in *FactoryBuild.xml*:

- As `type="java"`
- With a `src` attribute that specifies the pre-built Page's directory

For example, Phase Three of the Hello World Tutorial application has a Java Section that has buttons that enable the user to write to the framework log and to the framework status bar. This Java Section is declared in *FactoryBuild.xml* as follows:

```
<!--Definition of LogStatusbarJavaSection Section -->

<section name="NameKey.LogStatusbarJava" type="java"
                id="data/LogStatusbarJava" deletable="true"
                src="../prebuilt/LogStatusbarJava" />
```

Creating the pre-built Page and specifying the Java class

Every a Java Section requires pre-built SectionPage.

> **Note:** See "Pre-built Pages" on page 4-69.

The Java Section's pre-built SectionPage must have:

- A `<section>` element whose `type` and `id` attributes are the same as the Section's attribute values declared in *FactoryBuild.xml*.
- A `<java>` element whose `classname` attribute identifies the Java class to execute.

For example, here is the pre-built *LogStatusbarJava.xml* SectionPage from the Hello world tutorial (note the material in **bold**):

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile deletable="false">
   <header>
      <packageName>helloworld</packageName>
      <packageVersion>1.0</packageVersion>
      <template></template>
      <noCache>0</noCache>
      <defaultPageType>default</defaultPageType>
      <createDate utc="1121110270523" />
   </header>
   <body>
      <section name="LogStatusbarJava" type="java" id="data/LogStatusbarJava">
         <java classname="com.pepper.www.HW.LogStatusbarJavaSection" />
      </section>
   </body>
</pageFile>
```

The Java Section class

Every Java Section requires a Java class that follows the guidelines explained in this section.

Extending java.awt.Component

The Java Section's class must extend the java.awt.Component class, or one of its subclasses.

The Java Section's root class must provide a public constructor with no arguments in order to be instantiated by the framework.

In Phase Four of the Hello World Tutorial, a Java Section named WorldJavaSection is added to the application, as follows:

```
public class WorldJavaSection extends JPanel implements JavaSectionComponent {
    ...

    public WorldJavaSection() {
    }
    ...
}
```

Implementing the JavaSectionComponent interface

However, one additional step enables your Java Section to interact with the framework and make use of its rich suite of classes and methods.

This step is implementing the framework com.pepper.platform.program.JavaSectionComponent interface and implementing its `initComponent()` method, as follows:

```
public class WorldJavaSection extends JPanel implements JavaSectionComponent{
    ...

    public void initComponent(AbstractPepperProgram theProg, Properties params) {

        //cast the local reference to the application's base class
        //in order to write to the status bar
        baseClass = (BaseClass) theProg;

        //initialize the section components here, after getting reference
        //to program main class (theProg).
        initMe();
    }

    public void initMe() {
        ...
    }
}
```

In this example, we have created a local handle (`baseClass`) to the application's base class by assigning the AbstractPepperProgram that is passed through the `initComponent()` method to a local variable (defined previously as the type of the actual base class, in this case `BaseClass`). Through `baseClass`, we have access to all methods and fields associated with the AbstractPepperProgram application itself, which enables full integration into the Pepper Application Framework.

This example also shows the `initMe()` method as the last line in the `initComponent()`method. While its contents are not shown here, this `initMe()` method contains all initialization work for the Java Section, for example, creating a toolbar and other user interface work. Most initialization code for Java Sections should be placed either at the end of `initComponent()` or in a method that it is called from `initComponent()`, rather than in the class constructor. This approach ensures initialization occurs after the Java Section is fully integrated into the framework. If you place such initialization code in the constructor, the class may compile, but it may result in run-time null pointer exceptions.

Java ToolBars

The framework provides two Java classes that allow you to add a Java toolbar with buttons to a Java Section:

- com.pepper.guiutils.ToolBar
- com.pepper.guiutils.ToolBarButton

These classes extend the standard javax.swing.JToolBar and javax.swing.JButton to provide the following additional features:

- Their visual styling is consistent with other Java toolbars and buttons in other applications.
- Their colors and fonts are controlled by *keeper.css* styles, thereby enabling comprehensive visual styling changes among all Java Sections by customizing the single *keeper.css* file.

ToolBar coding guidelines

As with all initialization code, the ToolBar should be created and added in the Java Section's `initComponent()` method rather than in its constructor.

The root panel to which the ToolBar is added should have its LayoutManager set to BorderLayout. The ToolBar should be added to the `BorderLayout.NORTH` region.

The ToolBar and ToolBarButton classes have essentially the same API (methods and members) as their Swing equivalents. You can add ToolBarButtons that take: an Action class that extend a ProgramAbstractAction that you have registered in the base class, an icon and text.

The following example shows how to add a ToolBar with ToolBarButtons.

Tip: Ensure you have imported the package containing the classes: com.pepper.guiutils.*;

Example 4–10 Adding a ToolBar with ToolBarButtons to a Java Section

```
//Create the toolbar object
ToolBar toolbar = new ToolBar();

//Create a button associated with GotoMainTabAction class
ToolBarButton b1 = new ToolBarButton(new GotoMainTabAction(), "Go to Main Tab");

//Create a button associated with writeLog class
ToolBarButton b2 = new ToolBarButton(writeLog, "Write To Log");

//Add buttons to toolbar:
toolbar.addButton(b1);
toolbar.addButton(b2);

//Set root JPanel layout manager and add toolbar
this.setLayout(new BorderLayout());
this.add(toolbar, BorderLayout.NORTH);

//Note: Content JPanel is typically added to BorderLayout.WEST
```

Caching

We have seen that the framework automatically tracks the dynamic run-time hierarchy of the application, including: Section instances, SectionPage instances and Page instances.

The framework also maintains a dynamic hierarchy of information inside those files according to *caching rules* you define. These rules control the automatic propagation of elements from Pages into their parent SectionPage. This is useful because there are often cases when you want the SectionPage to display a list of Pages and some, but perhaps not all, of their content.

For example, in the second phase of the Hello World Tutorial, the main SectionPage shows a table containing the names of user-created worlds (but no other information about each world). Yet, each world's data, including its name, is defined and stored in a world Page, not in the SectionPage.

The following figure shows the rendered SectionPage with its table of worlds.

Figure 4–7 on page 4-54 shows a particular world's rendered Page.

Figure 4–6 SectionPage has world names cached to it from Page instances

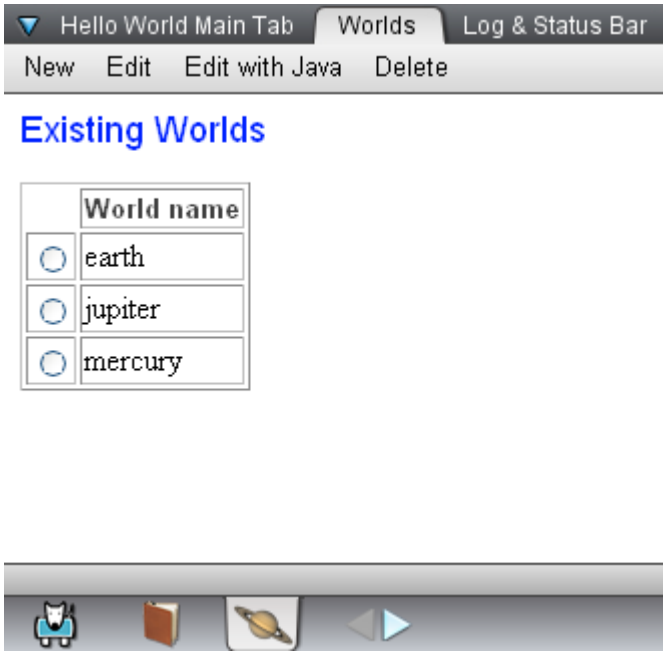
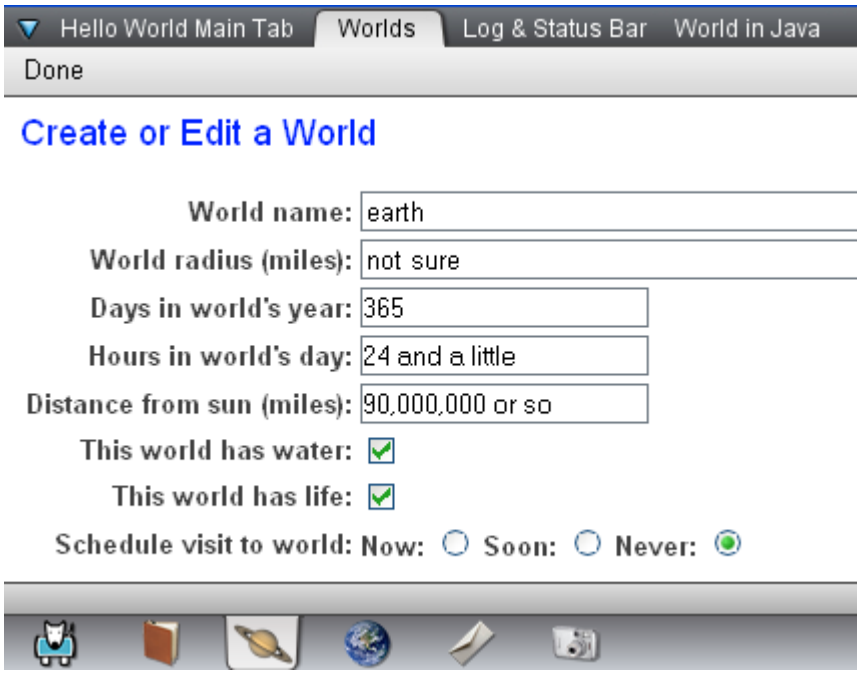


Figure 4–7 Rendered Page instance for a particular world



How does the SectionPage know the names of each world, when these are defined and stored in Pages? Caching, of course.

The caching rule in this case says:

> Let all world names defined in all world Pages automatically propagate into their parent SectionPage.

This is a simple case because the world names are direct child elements in each world's data section and because no attributes are cached. But, you can write caching rules that apply to any element or attribute.

Defining caching rules

Caching rules are defined in the SectionPage's segment of the *PageTemplates.xml* file. The following example is the SectionPage definition for Hello World example shown above with the caching rule in **bold**:

Example 4–11 Simple case of caching rules

```
<!-- Define Worlds sectionPage -->
<sectionPage type="worlds">
   <template>design/worlds.xsl</template>
   <defaultPageType>world</defaultPageType>
   <cacheRules match="page">
      <apply>worldName</apply>
   </cacheRules>
```

```
   <section />
</sectionPage>
```

Key points:

- `<cacheRules match="page">` instructs the cache rule to start the caching by finding the `<page>` element in the world Page XML instance file.
- Each world Page XML instance file has a `<page>` element that has a child element (`<worldName>`) that contains world's name.

  See the actual world Page XML instance file for this example here: "World Page sample" on page A-204.

  Recall that only the contents of the `<body>` element of the XML file (the *data* part of the file) are represented in the Page's JDOM Document. See Figure 4–2 on page 4-34.
- The `<apply>worldName</apply>` element is what defines the caching rule that gets each world's name and puts it into the SectionPage XML instance file.

  See the actual SectionPage XML instance file for this example here: "World Page sample" on page A-204.

  In this file, notice that for each world Page there is a `<page>` element that has a `<worldName>` child element whose test is the world's name that is cached upwards.

Caching rule syntax

Caching rules have two types:

- Selection rules

  Specify movement through the source Page data (this is the Page from which the information is being retrieved)
- Application rules

  Specify which data is to be retrieved from the source Page and cached into the target SectionPage

Selection rules

There are two selection rules:

- `<cacheRules match="page">`

  This is always the wrapper for the caching rule.

  This selects the source Page's `<page>` element, which is always the root of the Page data.
- `<template match=(`*`xpath`*`)>`

  Matches to nested source nodes are made with the `<template match=(`*`xpath`*`)>` element. The (*`xpath`*) value specifies the XML node (an element or attribute) that is to be the source of caching in the Page data.

Application rules

After a source element is matched, its child nodes to cache are specified with the <apply> element.

An matched element's attributes and its child elements are all considered *child nodes*, although they are specified for caching differently, as explained in following sections. To illustrate this point, in the following example, the <foo> element has four child nodes: the name and type attributes and the <bar> and <bat> elements. After matching <foo>, you could specify any of these nodes for caching. However, to cache one of <bar>'s attributes, you would have to match <bar> first, then use an <apply> statement to specify a child attribute.

Example 4–12

```
<page>
   <foo name=″myFooName″ type=″myFooType″>A Foo
      <bar name=″myBarName″ type=″myBarType″>A Bar</ bar>
      <bat>A Bat</ bat>
   /foo>
   <goo>A Goo</ goo>
</page>
```

Caching a single child element

The following rules cache the <foo> element and its text:

Example 4–13

```
<cacheRules match="page">
   <apply>foo</apply>
</cacheRules>
```

Caching multiple elements and attributes

Cache rules can specify multiple elements and attributes to cache by separating them with the pipe symbol: "|".

Assume the Page in Example 4–12 on page 4-56.

The following caches the <foo> and <goo> elements:

Example 4–14

```
<cacheRules match="page">
   <apply>foo|goo</apply>
</cacheRules>
```

Nested caching rules

You can cache data from multiple nested levels from the Page to the SectionPage.

Assume the Page in Example 4–12 on page 4-56.

Now, suppose you want to cache only <foo>'s text contents and <bar>'s text contents. You would do this with nested match rules and <apply> elements, as follows:

Example 4–15

```
<cachRules match="page">
   <apply>foo</ apply>
   <template match="foo">
      <apply>bar</ apply>
   </ template>
</cacheRules>
```

This results in the following data cached into the SectionPage:

Example 4–16

```
<page>
   <foo>A Foo
      <bar>A Bar</ bar>
   </ foo>
</ page>
```

Caching attributes

Specific attributes of the matched element are specified for caching with @(*attribute*), where (*attribute*) is the attribute name.

You can specify all attributes of the matched element with the attribute character ("@") plus the asterix ("*"). For example, <apply>@*</ apply> means cache all attributes of the matched node.

Assume the Page in Example 4–12 on page 4-56.

Suppose you want to cache <foo>'s text contents and all of its attributes. You also want to cache <bar>'s text but only its name attribute. And you don't want to cache any part of <bat>. The following rules accomplish this:

Example 4–17

```
<cachRules match="page">
   <apply>*@|foo</ apply>
   <template match="foo">
      <apply>bar</ apply>
      <template match="bar">
         <apply>@name</ apply>
      </template>
   </ template>
</cacheRules>
```

The cached data is as follows:

Example 4–18

```
<page>
   <foo name="myFooName" type="myFooType">A Foo
      <bar name="myBarName">A Bar</ bar>
   </ foo>
```

```
</ page>
```

When does caching occur?

Caching rules are applied whenever the framework saves the Page's XML instance file. Page saving is automatic. The framework saves a Page into its XML instance file whenever the Page loses focus, for example when a user clicks on another Tab or application.

This means that for strictly non-Java applications, you do not need to take any steps to ensure defined caching rules are carried out. If a Page modifies its data, those modifications are stored to file when the Page loses focus, and, if caching rules apply, they are implemented.

However, if you use a Java Section to modify data in a Page XML instance file, and if that data is subject to a caching rule, you must use a Java framework method (`AbstractPepperProgram.savePage()`) to trigger the Page's save to the XML instance file, which in turn triggers implementation of caching rules.

> **Note:** The first Java phase of the Hello World Tutorial does this. See "Hello World 3: Getting Started with Java" on page 8-123.

Connecting displayed data to Page data

This section covers how data displayed to the user in an HTML page or through a Java user interface is connected to the Page's JDOM Document, and through this, to the Page XML file.

Connecting HTML data to Page data

You can connect HTML `<form>` fields to specific XML elements and attributes in the Page's underlying XML file. The result is that when the HTML is displayed, the form's fields are pre-populated from the XML. And, if the user edits any data, the data is saved into the XML.

Before covering the details of how this is done, it's important to understand the framework's auto-save mechanism. The mechanism is simply that all HTML data that is connected to a Page is automatically saved by the framework into the XML when:

- The user navigates to a different application or Section.
- A different Page is loaded in the current Section.

This automatic mechanism ensures HTML data is always saved.

Connecting HTML data to an underlying Page requires the following:

- The connected HTML fields must be inclosed inside an HTML `<form>` element.

  You can use HTML `<input>` elements for text fields.

  The framework provides additional elements that make checkboxes and radio buttons easier to work with, as explained below.
- To display XML data into a connected HTML field, include a `value` attribute in the field whose value is an absolute XPath expression to the XML field in the Page data.

Tip: Absolute XPath expressions start with a leading forward slash (“/”) to indicate the root node, for example: `/page/wordName`

The XPath expression must be enclosed in curly braces.

Recall that the root of the Page data is the `<page>` element for Pages and `<section>` for SectionPages.

- To save connected field data into the underlying XML, include an `xpath` attribute whose value is an absolute XPath expression to the XML field in the Page data.

  The XPath expression must not be enclosed in curly braces.

The following example shows a fragment of an XSL file (*world.xsl*) used in the Hello World Tutorial. This XSL generates an HTML form whose elements are connected to the underlying Page XML through the Page’s framework JDOM Document. An HTML `<input>`, a `<pepper:checkbox>`, and a radio button group consisting of three `<pepper:radiobutton>`s are connected to the underlying Page XML elements that are specified with the `value` and `xpath` attributes.

Example 4–19

```
<form>
   <input type="text" id="worldName" class="Field" style="width:400px"
             value="{/page/worldName}" xpath="/page/worldName" />
   <pepper:checkbox  type="checkbox" id="hasWater"
             value="{/page/hasWater}" xpath="/page/hasWater" />
<span class="Label">Now: </span>
   <pepper:radiobutton id="planVisit" xpath="/page/planVisit"
             storedvalue="{/page/planVisit}" value="1" class="Text"/>
   <span class="Label">Soon: </span>
   <pepper:radiobutton id="planVisit" xpath="/page/planVisit"
             storedvalue="{/page/planVisit}" value="2" class="Text"/>
   <span class="Label">Never: </span>
   <pepper:radiobutton id="planVisit" xpath="page/planVisit"
             storedvalue="{/page/planVisit}" value="3" class="Text"/>
...
</form>
```

Connecting Java data to Page data

You can use Java that accesses and modifies Page data. This involves using the JDOM Document that represents the Page data to access the XML and framework classes to access the Page.

Note: Phase Four of the Hello World Tutorial provides instructions and complete code to open a Page XML file from Java, modify its contents in a Java Section, and save the modified data into a Page XML file. See “Hello World 4: Advanced Java” on page 9-135.

JDOM classes

The following JDOM classes are a few of the useful classes for manipulating Page XML files and data:

- org.jdom.Document
- org.jdom.Element

- org.jdom.Attribute

  Note: The JDOM jar that includes these classes is provided in the SDK *pepper-sdk/lib* directory.

  For javadoc of the JDOM API, see http://jdom.org/docs/apidocs/index.html.

Loading a Page XML file

To load a Page XML file in Java, you need the Page's Page ID and its Section ID.

- Page ID

  This important framework parameter is a String that consists simply of the `id` attribute of the Page. It specifies the path to the file. It consists of the *data* directory, a forward slash, and the filename (without extension) of the Page XML file. For example, if the Page's filename is *apagefile.xml*, its Page ID is "*data/apagefile*".

  See "Page XML definition and instance" on page 4-38.

- Section ID

  The Section ID is simply the Page ID of the SectionPage XML file.

  See "SectionPage XML definition and instance" on page 4-40.

Now, let's work through an example that shows how to load Page XML data into Java, modify it, and save it back into the Page XML file.

Let's assume the following:

- A Page ID of "data/apage"
- A Section ID of "data/asection"

To return a particular Page object, use the AbstractPepperProgram.getPage() method. (See "Useful Java methods for Sections and Pages" on page 4-44.) To use this method, you need a local handle to the actual AbstractPepperProgram instance. If your code is inside your AbstractPepperProgram class, you can simply used the method. If you are in an Action class that extends com.pepper.platform.program.actions.ProgramAbstractAction, you can use its getProgram() method to return the instance.

In this example, we assume we are in an Action class that extends ProgramAbstractAction. Therefore we can use getProgram(), as follows:

```
Page page = getProgram().getPage("data/apage", "data/asection");
```

Now that you have a handle to the Page object, load it into memory inside an `if` statement and a `try` statement, as follows:

```
if (!page.isLoaded()) {
    try {
        page.load();
    }catch (Exception e) {
        log.error("Page file failed to load into memory");
    }
}
```

Retrieve the framework JDOM Document object from the loaded Page using the com.pepper.platform.page.Page.getPageData() method, as follows:

```
Document document = page.getpageData();
```

You must wrap code that accesses or modifies the document's XML in a Java `synchronized` block in order to prevent data corruption and other problems that can result if the XML data is simultaneously accessed from multiple threads.

Then, you can use JDOM Document, Element, Attribute and other classes to read XML data into local Java fields. The following example encloses the data access in a `synchronized` block, gets the root element (which is the `<body>` element, as discussed previously), and gets the root element's child `worldName` element.

```
synchronized (document) {
   //Load XML element data into java widgets.
   //Note that the root element that is loaded is not the actual XML file's
   //root but is rather the <page> element.
   Element root = document.getRootElement();
   if (root != null) {
      Element worldName = root.getChild("worldName");
...
   }
}
```

So far, we have seen how to load the Page's XML data into Java. To save data into the XML, use the following approach.

Load the Page into Java as described previously.

Set the value of the XML elements or attributes using appropriate JDOM methods (inside a `synchronized` block, as described previously). For example, set the `worldName` as follows:

```
synchronized (document) {
   //Note that the root element that is loaded is not the actual XML file's
   //root but is rather the <page> element.
   Element root = worldData.getRootElement();
   if (root != null) {
      Element worldName = root.getChild("worldName");
      worldName.setText(“new name”);
...
   }
}
```

Finally, save the Page with the AbstractPepperProgram.savePage(Document, boolean) method, where the boolean controls whether to switch the focus to the Page. Put this code inside a `try` statement in order to effectively handle and report error conditions, as follows:

```
try {
   getProgram().savePage(page, false);
}catch (Exception e) {
   log.error("Cannot save Page XMl file to disk");
}
```

JavaScript/Java approach

The first method discussed directly connects HTML form data to Page data. However, you may want to use JavaScript to validate or otherwise process data before allowing it to be stored into the Page. In such cases, you could use JavaScript and Java, as follows:

- Fire a JavaScript method (perhaps from an HTML button) to retrieve the form data and validate it.
- If the data is valid, pass the data to an Action class (through the LiveConnect JavaScript-to-Java bridge) that saves the data to the Page.

JavaScript and Mozilla LiveConnect

Your HTML (and XUL) pages can include JavaScript. The JavaScript can be integrated with key framework Java classes through the framework's support for Mozilla LiveConnect. This integration enables such things as triggering Java Actions from JavaScript and accessing JavaScript objects and functions from Java.

> **Tip:** The Firefox browser integrated in the framework provides a JavaScript console that may be useful to debug JavaScript errors. Launch the JavaScript console by entering "javascript:" into any Web Tab address field in the framework.

Including external JavaScript

The best approach for using JavaScript in your HTML is to place the JavaScript in an external file in the application's *design* directory and use XSL to generated HTML that includes the external file with the HTML `<script>` tag's `src` attribute.

The reason to place the JavaScript in an external file is that if you placed it directly in the XSL you would have to wrap it in a CDATA block. That's because JavaScript is not valid XML and an XSL file must contain only valid XML or text wrapped in a CDATA block.

A few rules apply when including JavaScript from an external file:

- The external file containing the JavaScript must be placed in the application's *design* directory.
- The `<script>` tag's `src` attribute refers to the *design* directory using the `$design` framework parameter enclosed in curly braces and followed by `/design/`*`javascriptFile`*, as follows:

```
<script language="JavaScript" type="text/javascript"
                src="{$design}/design/javascript.js"/>
```

> **Note:** To use the `$design` framework parameter in an XSL transform file, you must first define it as an XSL parameter, as in: `<xsl:param name="design" />`. For information about framework parameters, see "Framework parameters passed to XSL" on page 4-68.

LiveConnect

The framework uses LiveConnect Mozilla technology to connect JavaScript to Java.

For information about LiveConnect, see:

`http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:LiveConnect`

LiveConnect enables the following:

- JavaScript code can use LiveConnect to access Java variables, methods, classes and packages.
- Java code can use LiveConnect to access JavaScript methods and properties.

LiveConnect initialization

LiveConnect must be initialized in each HTML (or XUL) page. Initialize LiveConnect with the following in XSL extension element in the <head> of the HTML:

```
<pepper:initscriptbridge package="{$packageId}" />
```

There are two points to note about this script initialization code:

- It uses the `pepper` namespace prefix.

  Every XSL file requires that this be defined correctly.

  See "Required XSL namespaces and Xalan configuration" on page 4-69.

- The `$packageId` parameter is used.

  Note: To use any framework parameter in an XSL transform file, you must first define it as an XSL parameter, as in: `<xsl:param name="packageId" />`. For information about framework parameters, see "Framework parameters passed to XSL" on page 4-68.

Including this initialization in the XSL inserts the following JavaScript into the HTML/XUL:

```
<script language="JavaScript" type="text/javascript">
var bridge;
var packageId='Your-PackageId';
function initBridge() {
  bridge = Packages.com.pepper.script.Bridge.getPackage(packageId);
}
</script>
```

When the HTML (or XUL) page is loaded, the framework calls the `initBridge()` function as a final step. The JavaScript `bridge` variable is set to the Java com.pepper.script.Bridge class (see javadoc for more information). This initializes LiveConnect. You can use the JavaScript `bridge` variable to access Java com.pepper.script.Bridge methods.

For example, you can execute a registered Java Action associated with the name "myAction" from JavaScript as follows:

```
bridge.action('myAction');
```

Page Initialization with LiveConnect

You can use LiveConnect to access certain framework Java during HTML (and XUL) page initialization.

For example, you might want to access a message in a message catalog during HTML initialization. Do this by passing one or more JavaScript functions to the `<pepper:initscriptbridge>`

element with its `oninit` attribute. These functions are automatically loaded into and called from the `initBridge()` method directly after the `bridge` variable is initialized.

For example, assume:

- There are two JavaScript functions: `myInit1()` and `myInit2()`.
- These functions are defined in *myJavascript.js.*

The following XSL code:

```
<script language="JavaScript" src="{$design}/design/myJavascript.js"/>
<pepper:initscriptbridge package="{$packageId}" oninit="myInit1(); myInit2();"/>
```

Generates the following JavaScript:

```
<script language="JavaScript" type="text/javascript">
var bridge;
var packageId='Your-PackageId';
function initBridge() {
  bridge = Packages.com.pepper.script.Bridge.getPackage(packageId);
  myInit1();
  myInit2();
}
</script>
```

As noted, the framework calls the `initBridge()` function as a final step when the HTML (or XUL) page is loaded. This initializes LiveConnect and then executes `myInit1()` and `myInit2()`. You can use the `bridge` variable in these functions to access framework Java, as explained in the next section.

> **Note:** HTML's `<body onload="">` handler executes before LiveConnect initialization. Therefore it is necessary to use the approach described above when perform initialization that accesses the framework, for example calling a registered Action or retrieving a property from the program.

Accessing Java from JavaScript

As mentioned previously, the com.pepper.script.Bridge class is the gateway through which JavaScript can access certain Java functionality in your application. For example, this class provides:

- The ability to get properties defined in your application.

  See javadoc for Bridge.getProperty()
- The ability to execute registered application Java Actions.

  See javadoc for Bridge.action()
- The ability to get messages defined in your applications Message Catalog.

  See javadoc for Bridge.getMessage()

  See javadoc for com.pepper.script.Bridge for a complete listing of methods.

To access these functions from JavaScript, simply use the global `bridge` variable that is initialized through the `<pepper:initscriptbridge>` element.

For example, the following displays an application property named “foo” when you press an HTML button:

```
<input type="button" onclick="alert(bridge.getProperty('foo'));"/>
```

The following execute an action associated with the name “myAction” when the user clicks an HTML button:

```
<input type="button" onclick="bridge.action('myAction');"/>
```

Note: One important concept when using LiveConnect is that Java objects passed to JavaScript are not automatically converted to JavaScript objects. Therefore, you often need to convert them before using them. For example, Java Strings are passed, not JavaScript Strings. Convert passed Java Strings into JavaScript Strings by appending an empty character before using them, as follows:

```
var javascriptString = javaString + "";
```

Accessing JavaScript from Java

LiveConnect provides the ability to access JavaScript objects and functions from Java. This is done through the netscape.javascript.JSObject class.

In the Keeper Framework, your application can access JSObject through the WebBrowserView interface.

See javadoc of com.pepper.guiutils.WebBrowserView.

The following example shows how to derive a local handle (`jsObj`) to JSObject using the WebBrowserView interface:

```
WebBrowserView view = AbstractPepperProgram.getBrowserContainer().getActiveView();
JSObject jsObj = view.getJSObject();
```

You can then use the JSObject.call() method to execute your JavaScript functions, as follows:

```
jsObj.call('somejavascriptfunction', new String[] {'arg1'});
```

However, the page must be fully loaded before you can access the JSObject to call into JavaScript. An application can override the AbstractPepperProgram.pageDisplayed() method to be notified when a page completes loaded.

This method is part of the com.pepper.platform.program.PageChangeListener interface that AbstractPepperProgram implements.

For example, the following code in your AbstractPepperProgram base class enables you to call the `somejavascriptfunction()` JavaScript function when a Page with this page ID “data/myPage” is loaded:

```
public MyProgram extends AbstractPepperProgram {
...
    public void pageDisplayed(PageChangeEvent event) {
        super.pageDisplayed(event);
        tring pageId = event.getPageId();
        if (pageId.equals("myPage")) {
            // page load is complete for page "myPage"
            try {
```

```
            JSObject jsObj = getBrowserContainer().getActiveView().getJSObject();
            jsObj.call('somejavascriptfunction', new String[] {});
         } catch (Exception e) {
         }
      }
   }
}
```

Java Actions

This section explains how to develop your own Java Actions.

> For an example, see Phase Four of the Hello World Tutorial: "Hello World 4: Advanced Java" on page 9-135.

Such Actions can be triggered from JavaScript through LiveConnect.

> See "JavaScript and Mozilla LiveConnect" on page 4-62.

Action classes

An Action class must implement javax.swing.AbstractAction or extend com.pepper.platform.program.actions.ProgramAbstractAction.

An Action class must also implement the `actionPerformed(ActionEvent ev)` method. This is where you put the code to carry out your Action.

This is also the location where you may obtain a local handle to the base class, which is passed as an argument. This can be helpful if you want to use a method associated with the base class, for example the `getGSP.writeStatus(String)` method, which writes a transient message to the framework status bar.

> **Note:** This step is not required if your Action class is an inner class of the base class.

The following example shows how the EditJavaAction Action class, in `actionPerformed()`, obtains a local handle to the base class (HelloWorld) in Phase Four of the Hello World Tutorial.

```
this.helloWorld = (HelloWorld) getProgram();
```

Retrieving passed parameters

If one or more parameters are passed to the Action, retrieve them using the `ActionEvent` passed into the `actionPerformed()` method.

Inside `actionPerformed()`, cast the `ActionEvent` as an `com.pepper.platform.program.ActionEventWithParams.ActionEventwithParams` object.

Then use `ActionEventwithParams` object's `getParam()` method to return the parameters and assign them to local variables.

The following example shows how the EditJavaAction Action, developed in Phase Four of the Hello World Tutorial, retrieves a passed parameter.

```
String pageId;
pageId = ((ActionEventWithParams) event).getParam();
```

Registering a Java action in the application base class

Actions that are triggered from JavaScript in an HTML page or through AbstractPepperProgram.getAction() have to be instantiated and registered in the application's base class.

Register and instantiate the Action with the `registerAction(String, new Action() )` method inside the base class's required `init()` method, where:

- `String` is the Action name.
- `Action` is the Action class name.

The following example shows the Action name String constants for the EditJavaAction Action and the DoneJavaAction Action being declared and defined in the Hello World application's base class.

```
public static final String EDIT_JAVA = "EditJavaAction";
public static final String DONE_JAVA = "DoneJavaAction";
```

The following example shows the EditJavaAction Action and the DoneJavaAction Action developed in Phase Four of the Hello World Tutorial, being instantiated and registered in the `init()` method of the application's base class.

```
public void init(PepperProgramConfig config) throws PepperProgramException {
   super.init(config);
   //Register application's Actions with framework
   registerAction(EDIT_JAVA, new EditJavaAction());
   registerAction(DONE_JAVA, new DoneJavaAction());
}
```

Calling Java Actions from Java

When an Action has been properly instantiated and registered, it can be fired from anywhere in Java. You can use AbstractPepperProgram.getAction(String ActionName) to retrieve an action registered by your application, or simply use the Action directly if it is stored as a member in your application.

To fire an Action from a com.pepper.guiutils.ToolBarButton, instantiate the button using the constructor that takes an new instance of an Action class as a parameter, and label text, as follows.

```
ToolBarButton b1 = new ToolBarButton(new DoneJavaAction(), “Done”);
```

XSL

XSL can be used to display Pages.

Figure 4–5 on page 4-46 is a diagram that illustrates the process through which Pages are displayed.

The particular XSL file used to display a Page is specified with the `<template>` element in the Page's definition in *PageTemplates.xml*.

The XSL file can generate HTML or XUL.

Note: XUL is not explained in this guide.

Framework parameters passed to XSL

The framework passes a set of parameters to each XSL transform when the transform executes. The parameters define useful aspects of the application's run-time environment that are not known until the application is installed in the framework. There are parameters for the application's unique ID, its *data.zip* and *design.zip* files, and the framework's root installation directory.

For example, use the application's ID to initialize its LiveConnect JavaScript-to-Java bridge. Use the parameter that defines the application's *design*.zip file to link your own CSS stylesheet or external JavaScript file (residing in the *design.zip/design* directory) into the generated HTML.

Each parameter passed into the XSL transform has a name and a value, as described next.

Table 4–2 shows the framework parameters and for each provides its name and a description.

Table 4–2 Framework parameters passed to XSL

Parameter name	Description
packageId	Uniquely identifies this application (package) in the framework.
platform	The path to the root directory of the framework installation.
data	The path to the application's installed *data.zip* file
design	The path to the application's installed *design.zip* file.

Using framework parameters in XSL

To use a passed parameter, define it in the XSL transform as an XSL parameter with its exact name. The new XSL parameter is assigned the value of the parameter with the same name that is passed to the XSL transform by the framework. After defining the parameter, refer to it in XSL by preceding it with a dollar sign ("$").

Example 4–20 shows how to define framework parameters in XSL.

Example 4–20 Defining framework parameters in XSL

```
<!-- define parameters passed by the framework into this XSL stylesheet -->
<xsl:param name="packageId" />
<xsl:param name="platform" />
<xsl:param name="design" />
<xsl:param name="data" />
```

You use `design` parameter in XSL to include an external JavaScript file.

See "Including external JavaScript" on page 4-62.

You use `packageId` parameter in XSL to create a Page toolbar.

See Example 4–8 on page 4-47.

Required XSL namespaces and Xalan configuration

XSL files used to generate Page HTML must conform to a few rules in order to integrate into the framework.

What this amounts to is simply that every XSL file should contain certain code shown in "Required namespace and Xalan configurations in XSL transforms" on page 4-69.

Every XSL transform requires the following:

- A series of namespace definitions including `xsl`, `xalan` and `pepper`
- Configuration of the Xalan XSL processor to enable it to access Java methods associated with the framework com.pepper.script.Elements class for elements using the `pepper` prefix

  This is accomplished by declaring the `pepper` prefix as an extension element prefix and then using the `<xalan:component>` and `<xalan:script>` elements to point Xalan to the framework `com.pepper.script.Elements` class.

Example 4–21 shows the required namespace and xalan configurations that must be in every XSL transform, typically at the top.

Example 4–21 Required namespace and Xalan configurations in XSL transforms

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xalan="http://xml.apache.org/xalan"
   xmlns:pepper="http://www.pepper.com/elements"
   extension-element-prefixes="pepper">
   <xalan:component prefix="pepper">
      <xalan:script lang="javaclass" src="com.pepper.script.Elements"/>
   </xalan:component>
...
</xsl:stylesheet>
```

Pre-built Pages

As we have seen, there are two kinds of Sections: Java and Default. We have also seen that Default Sections require a SectionPage and may optionally have one or more Pages.

To be displayed, every Page needs an XML instance file.

(That is because the XSL processor receives the contents of the `<body>` element of the Page XML instance file and transforms it into HTML according to the Page's XSL transform file.)

Page XML instance files come into being in one of the following ways:

- The Page is created from its definition in *PageTemplates.xml* during the build process.
- The Page is created from its definition in *PageTemplates.xml* when needed by the framework during application execution.
- The developer makes a *pre-built* Page that is included in the application's distribution by the build process.

This section explains why and how to create and use pre-built Pages.

Why use pre-built Pages?

Sometimes, pre-built Pages are required. Sometimes they are useful and convenient.

A Java Section requires a pre-built SectionPage.

See "Java Sections" on page 4-49.

Pre-built non-Java Sections are useful and convenient when you want to pre-populate data in a SectionPage or a Page. For example:

- Web bookmarks are pre-built Pages.
- Help Sections use pre-built SectionPages and Pages.
- Games use pre-built Pages to contain their Flash programming.

You can think of pre-built Pages as enabling the developer to pre-populate information into a Section's SectionPage and/or its Pages so that it is present at application first launch.

Pre-populating by copying-pasting-editing

You can pre-populate an application in the SDK with Pages you've generated in a different running application. For example, you can use a web Section to generate a set of bookmarks and then give them to an application in the SDK. You simply copy and paste the Page XML files and make minor edits (as explained below).

Such Pages are considered *pre-built* because they are not created by the build process or by the application at run time but instead are provided to the application in the SDK.

Pre-populating without copying-pasting-editing

Pre-built Pages are also useful to pre-populate an application with Pages of a type that the user cannot create by using the application. For example, you might use pre-built Pages for games, where each Page provides information about a game, such as the file containing the Flash programming for the game. Yet, the application doesn't support creating new game Pages.

Why not pre-populate the data into the XSL transform?

Since XSL can include into its generated HTML whatever you want it to, it is possible to put your data (like web bookmarks, help information and Flash programming commands) in an XSL transform rather than in a pre-built Page.

So you might wonder, are pre-built Pages really necessary for these cases? Strictly speaking, no, they are not. However, there are reasons for using pre-built Pages instead of XSL.

From the point of good programming practice, it is a good idea to separate data from presentation. Placing the data in XML instance files (the pre-built Pages) rather than in XSL transform files (which is used to generate presentation HTML) is therefore the better approach.

But, pre-built Pages are often easier to use for this purpose.

Pre-built Pages are often *easier* than XSL

In some cases it is easier to use a pre-built Page than XSL.

Consider the example of pre-populating a Web Section with a set of pre-built bookmark Pages.

- You can use any existing Web Section to browse the internet and create and save bookmarks.
- Each bookmark is saved by the framework as a Page XML instance file.
- You can copy these bookmark Page XML instance files and include them as pre-built Pages in your own Web Section.
- You need to put the bookmark files in the right directory, modify the `id` attribute of each bookmark, and modify the Section definition to point to the bookmark directory (discussed further below).
- Then, after the application is built, the bookmarks are present.

This approach is far easier than using XSL to generate bookmark Pages.

While this example deals with bookmarks, it applies generally. If an application creates Page XML instance files, you can copy the instance files and use them as pre-built Pages, as long as you follow the required steps.

Can pre-built Pages be modified during application use?

Yes.

Pre-built Pages are the starting point for the Page when the application first executes. As the application is used, the Pages can be modified, if the application is designed to support this. For example, cache rules execute as specified and modify the XML instance files, whether pre-built or not. As with all Pages packaged with an application, when a pre-built Page is modified, its new version is saved in the *data* directory, and that is used from then on instead of the Page XML instance file provided with the application build (in *data.zip*).

Pre-built Pages

Pre-built Pages (non-SectionPages) are the most typical type (although pre-built SectionPages are also supported — and required for Java Sections and Help Sections).

Let's take another look at the Web Section bookmarks example:

- Some Page bookmark files are pre-built
- The SectionPage is not pre-built

To work, a number of requirements must be met. The following sections demonstrate these requirements with respect to pre-built bookmark Pages for the ready-made Web Section that is added to the Hello World application in the first phase of the Hello World Tutorial. (See "Hello World 1: Application Creation" on page 6-89.)

However, the requirements explained here apply generally.

Defining the Section's pre-built Page directory

For a Section to use pre-built Pages (when there is no pre-built SectionPage), the SDK build system has to know where the pre-built Page XML instance files reside. You provide this information in the Section's definition in *FactoryBuild.xml* with the `<prebuiltPagesDir>` element.

Example 4–22 shows how the Hello World's ready-made Web Section is defined in *FactoryBuild.xml.*

Example 4–22 Defining a Web Section's pre-built Page location

```
<section name="NameKey.WebSection" type="web" id="data/web" builtin="true"  >
   <prebuiltPagesDir>../prebuilt/websection</prebuiltPagesDir>
</section>
```

The directory path specified by `<prebuiltPagesDir>` is relative to the application's *design* directory. It therefore implies the following directory structure in the application's SDK directory:

Note: Only the applicable directories are shown.

- *Phase2*
  - *design*
  - *prebuilt*
    - *websection*

If you look in the Hello World Tutorial source files, you will see that *websection* contains the following subdirectory:

- *bookmarks*

The *bookmarks* directory contains the pre-built Page XML instance files.

Why didn't the *FactoryBuild.xml* Section definition include this final *bookmarks* directory? Because `<prebuildPagesDir>` specifies what to copy, during the build, into the *data* directory for distribution with the application. What is found in this instance, and therefore copied, is the *bookmarks* subdirectory, including its pre-built bookmarks. Your application may have other types of pre-built Pages. This mechanism enables them to be sorted into different run-time directories.

How then are they found at run-time if their complete directory location is not specified? The all-important `id` attribute. Recall that a Page `id` attribute occurs twice:

- Each SectionPage contains a `<page>` element for each child Page, and that `<page>` element has an `id` attribute that specifies the path to and filename of the Page XML instance file.

  For information, see Figure 4–3 on page 4-35.

- Every Page XML instance file's `<page>` element contains its own `id` attribute, which is, by definition, identical to the one in its parent SectionPage that points to the Page.

When a SectionPage is not pre-built but has pre-built Pages, the SectionPage XML instance file is created from scratch by the build system and contains the required `<page>` elements with their `id` attributes.

> **Note:** If there are no pre-built Pages, and if the SectionPage is not pre-built, the SectionPage XML instance file is not actually created until it is needed, which is when the Section is first viewed.

So:

- Because the contents of the directory pointed to by the `<prebuiltPagesDir>` element are copied to the *data* directory during the build,
- And because those contents included the *bookmarks* directory, which in turn included pre-built bookmark files,
- The SectionPage's `id` attribute for each Page takes the following form: `data/bookmarks/(`*`bookmarkFilename`*`)`.

This shows that the Web SectionPage is built to "know" where to find the pre-built bookmark Page XML instance files.

There is one more requirement for creating pre-built Pages: the pre-built Pages must have an element structure that adheres to the Page's definition in *PageTemplates.xml*.

> **Tip:** Every Page, whether pre-built or not, must be defined in *PageTemplates.xml*.

The next section covers how to create pre-built Page XML instance files.

Element structure of pre-built Pages

A pre-built Page XML instance file is identical to non-pre-built Page XML instance files. Their content and structure do not depend on whether they are pre-built or created by the framework at run-time.

- If it is a pre-built SectionPage, it must have the required element structure of a SectionPage.
- If it is a pre-built Page, it must have the required element structure of a Page.

  For information about Page XML instance file structure and content, see Figure 4–2 on page 4-34.

In addition, the Page must have the element structure specifically defined for it in *PageTemplates.xml*.

Defining a pre-built Page

For example, in the first phase of the Hello World Tutorial, a Web Section is added. The Web Section is pre-populated with pre-built Page bookmark XML instance files.

Example 4–23 shows the XML element structure of the bookmark Page as defined in *PageTemplates.xml*.

> **Note:** The definition for Bookmark Pages is included in *PageTemplates.xml* by including the framework *Bookmark.xml* file.

Example 4–23 Bookmark element definition included in PageTemplates.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- definition for Web bookmarks -->
<basePage type="web-bookmark">
```

```
<template>design/bookmark.xsl</template>
   <page>
      <title/>
      <url/>
      <host/>
      <description />
      <thumbImage />
      <date/>
      <visited/>
      <clipping/>
      <history/>
   </page>
</basePage>
```

So, the pre-built bookmark file's <page> element must contain a <tile>, a <url> and the rest of the defined elements.

Creating pre-built Pages

Continuing with the ongoing example, Example 4–24 shows a run-time bookmark Page XML instance file.

> **Note:** The SDK version of the file is slightly different, which makes the whole process easier for the developer, as discussed in the next section.

Example 4–24 A pre-built bookmark XML instance file

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile>
   <header>
      <pkgName>MusicPlayer</pkgName>
      <pkgVersion>1.0</pkgVersion>
      <template>design/bookmark.xsl</template>
      <createDate utc="1080057939081" />
   </header>
   <body>
      <page id="data/bookmarks/cnetmusic" name="Music Center - CNET.com"
                type="web-bookmark">
         <title>Music Center - CNET.com</title>
         <url>http://reviews.cnet.com/
                              Music/4520-7899_7-5105287-1.html?tag=mc</url>
         <host>reviews.cnet.com</host>
         <description>reviews.cnet.com</description>
         <thumbImage>design/images/webpage.png</thumbImage>
         <created>Mar 23, 2004 11:05:39 AM</created>
         <visited>Mar 23, 2004 11:05:39 AM</visited>
         <clipping />
         <history />
      </page>
   </body>
</pageFile>
```

Take note of the following:

- The <page> element's id attribute is the run-time path to and name of this file (without the *xml* file extension).

  In other words, the distribution (built) application has a *data/bookmarks* directory that contains a *cnetmusic.xml* file.

- The element structure of <page> and its child elements is consistent with the definition in the *FactoryBuild.xml*.

Delete the id attributes — the build provides them

The build process provides correct id attributes. All you have to do is delete them from the pre-built Page XML files in the SDK.

For example, you might copy a Page XML file named *afile.xml* from the following run-time directory: *data/collection*. Such a file would have an id attribute as follows:

```
id="data/collection/afile"
```

Let's suppose your application's Section stores its files in *data/holder.* So, in your case, the id attribute would have to be:

```
id="data/holder/afile"
```

And, let's suppose your section is declared as follows:

Example 4–25 A Section's instance declaration in FactoryBuild.xml

```
<section name="NameKey.MySection" type="files" id="data/sec1" builtin="false"  >
   <prebuiltPagesDir>../prebuilt/sec1</prebuiltPagesDir>
</section>
```

The Section instance declaration implies the following SDK directory for the application:

(application)/prebuilt/sec1

Since you have decided to store the pre-built Page XML instance files in a directory named *holder*, the following SDK directory is implied:

(application)/prebuilt/sec1/holder

This is inconsistent with the id attribute in the file you copied from the other application and want to use as a pre-built file because as we have seen its id attribute is:

```
id="data/collection/afile"
```

but needs to be:

```
id="data/holder/afile"
```

Fixing this is straightforward:

- Simply put the copied XML files with the incorrect id attributes into the correct SDK directory.
- Delete the id attributes from the files.

The build process adds correct id attributes to each built XML file that is packaged with the application based on the `<prebuiltPagesDir>` contents in Section definition, the file directory location in the SDK and the file name.

One final point: XSL transform required

Before wrapping up the pre-built Page discussion, let's hit one final point: pre-built Pages require an XSL transform.

The specific XSL transform to use for each Page is indicated in its `<template>` element in *PageTemplates.xml*. (Unless the `<template>` element refers to an HTML page, local or network.) The developer must ensures the `<template>` element points to the appropriate XSL transform.

So far, we've covered pre-built Pages in detail. How about pre-built SectionPages? That's covered next.

Pre-Built SectionPages

Pre-built SectionPages are supported, although not used as often as pre-built Pages.

> **Note:** As mentioned, all Java Sections require a pre-built SectionPage. Help Sections also use pre-built SectionPages.

Here's an example of when pre-built SectionPages might be appropriate: a Games application.

Suppose you have a bunch of Flash games, each of which requires a handful of parameters (screen size, quality, etc.) to be set at launch time. You could create a Section for each game. Each Section's SectionPage is pre-built, and the XML instance file contains the parameter settings appropriate for each game. In this application, no Pages are used at all.

Creating a pre-built SectionPage

As with pre-built Pages, a pre-built SectionPage's element structure must be consistent with its element structure as defined in *PageTemplates.xml*.

Also as with pre-built Pages, when using a pre-built SectionPage, you identify a directory that contains one or more files. In this case, naturally, the pre-built SectionPage XML file must be present. All files present are copied by the build into the *data* directory and included with the application for distribution.

However, the method for identifying the pre-built directory differs between pre-built Pages (without a pre-built SectionPage) and a pre-built SectionPage (with or without pre-built Pages), although in both cases, the directory is specified for the Section in *FactoryBuild.xml*.

- Identifying the directory for pre-built Pages without a pre-built SectionPage

  Use the `<prebuiltPagesDir>` element, as discussed previously.

- Identifying the directory for a pre-built SectionPage, with or without pre-built Pages

  Use the `<section>`'s `src` attribute to specify the directory. The path is relative to the SDK *(application)/design* directory. The `src` attribute triggers a subset of the `<prebuiltPagesDir>`'s functionality. All files contained are copied to *data* (which is also `<prebuiltPagesDir>`'s functionality), but the build does not construct a SectionPage. Instead, the build looks for (and requires) a pre-built SectionPage.

Use the <section>'s id attribute (in *FactoryBuild.xml*) to specify the pre-built SectionPage XML instance filename. As we have seen, the id attribute specifies the run-time path to an XML instance file and its filename (without its "xml" extension). That path is always in the *data* directory, therefore the id attribute starts with *data*. The final token in the id attribute specifies the XML filename of the pre-built SectionPage (without its "xml" extension).

Putting these two rules together, the build system discovers the pre-built SectionPage's location and filename. So, the developer must:

- Put the pre-built SectionPage XML file in the directory specified by the src attribute.
- Ensure it has the filename specified by the id attribute, excluding the path.

Example 4–26 shows how a Section is defined in *FactoryBuild.xml* with a pre-built SectionPage that resides in *prebuilt/DirName* that is named *Test.xml*.

Example 4–26 Specifying the directory and filename of a pre-built SectionPage

```
<section name="Test" type="test" id="data/Test" deletable="true" builtin="false"
src="../prebuilt/DirName" />
```

Pre-Built SectionPage *and* Pages

We've seen that many applications use pre-built Pages. And how to create a pre-built SectionPage. Which leaves an obvious question: is it possible to have a pre-built SectionPage that in turn has pre-built Pages? Yes, although it is not as common.

Bear in mind that if you just have a bunch of pre-built Pages, the SectionPage is created at build time with the Pages cached into it. The hierarchy of inter-pointing XML instance files is intact; the SectionPage has a <page> element and id attribute for each pre-built Page. And, if the SectionPage had cache rules to bring specific data from each Page into it, this occurs as well at build time. And, of course, XSL generates the HTML. This meets the needs of most applications.

If an application needs a pre-built SectionPage *and* pre-built Pages, this is also supported. Simply create the pre-built Pages and the SectionPage exactly as described in the previous sections. The SectionPage and the directory containing the pre-built Pages is identified with the src attribute. When the src attribute is used, the <prebuiltPagesDir> attribute is not required. All pre-built Pages are copied into *data* for distribution.

In this case you need to take special care to ensure the SectionPage contains a <page> element for each pre-built Page, which in turn has the appropriate id attribute to identify the Page XML instance file. If the SectionPage lacks these, they are not displayed.

Also, if the SectionPage has cache rules defined for it in *PageTemplates.xml*, you need to manually cache the appropriate information into the pre-built SectionPage XML file.

Event notification

The framework includes a mechanism to notify interested listeners of certain events that occur during runtime. The framework uses the standard Java event listener paradigm in which interested parties register themselves as listeners and are notified at the appropriate time.

As a convenience, the AbstractPepperProgram base class registers itself as a listener for many typical events. This allows subclasses to simply override appropriate listener methods and prevents the need to fully implement the listener interface.

The class definition for AbstractPepperProgram shows the interfaces that it implements, as follows:

```
public abstract class AbstractPepperProgram
    implements SectionChangeListener,
    ActionListener,
    PageChangeListener,
    ProgramChangeListener,
    PackageInstance {
}
```

> **Note:** The javadoc for AbstractPepperProgram also lists EventListener as an implemented interface. This is because ActionListener extends EventListener.

com.pepper.platform.program.PageChangeListener

The PageChangeListener interface enables reception of event notifications that are fired when an application Page is created, displayed, modified or deleted. You can use this to respond programmatically to such events.

> See javadoc of com.pepper.platform.program.PageChangeListener for methods and details.

As noted, the first step is to override the appropriate method.

For example, suppose you want to respond to changed Page data. In this case you would override pageModified(PageChangeEvent).

You can determine which Page was modified because PageChangeListener methods pass a com.pepper.platform.program.PageChangeEvent. You can use the PageChangeEvent.getPageId() method to return the ID of the Page whose data changed. If the modified Page is the right one, you can execute your own code. For example, you might execute a local method that includes the appropriate code, optionally defining it to pass the PageModifiedEvent, as follows:

```
public class MyProgram extends AbstractPepperProgram {
    ...
    public void pageModified(PageChangeEvent event) {
        // always call super class first as AbstractPepperProgram
        // performs default implementation
        super.pageModified(event);

        //determine whether this is the right Page
        if (event.getPageId().equals("idOfInterest")) {
            // app specific processing here
            handlePageModified(event.getPageId());
        }
    }
    ...
}
```

com.pepper.platform.program.SectionChangeListener

This interface is used to listen for events that occur when a Section (a tab in your application's user interface) is changed. That is, the listener is notified when the user interface changes its current tab. It may be appropriate to listen for these events if your application wishes to perform additional logic when a specific tab is hidden or shown.

As with the PageChangeListener, simply override the specific listener method of interest in your AbstractPepperProgram subclass to listen for these events.

com.pepper.platform.program.ProgramChangeListener

Similar to SectionChangeListener, this interface is used to listen for events that occur when a Keeper Application is hidden or shown.

Other listener interfaces

See the javadoc for com.pepper.platform.program for information about other listener interfaces.

Writing to the framework log

The framework uses the Apache open source LogFactory logging mechanism. This mechanism enables logging events or information in the framework log file as directed by Java code.

> **Tip:** You can create any number of logging engines, but nothing is gained with more than one. Instead, it is recommended to create one logging engine and use it for all Java associated with a particular application.

To log:

- Include the following import statements in the classes writing log events:

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

- Create a logging engine instance, as follows:

```
static Log log = LogFactory.getLog("com.pepper.yourProgram");
```

  To ensure your application creates only a single logging engine, always place your application's package path inside the double quotes.

- Write events to the framework log using the following Log methods:

  `log.info(String)`

  `log.warning(String)`

  `log.error(String)`

  `log.fatal(String)`

  `log.debug(String)`

GUI services

The com.pepper.guiutils.GuiServicesProvider interface provides methods that enable you to programatically access many aspects of the framework's user interface.

The object that implements this interface is typically retrieved using AbstractPepperProgram.getGSP().

Once you have retrieved a handle, you can access its methods. A few useful methods are listed here.

> **Note:** See the javadoc for complete API documentation.

To write to the Status Bar:

```
AbstractPepperProgram.getGSP().writeStatus();
```

For more information, see "Writing messages to the framework status bar" on page 4-80.

To retrieve the System Tray:

```
AbstractPepperProgram.getGSP().getSystemTray();
```

For more information, see "Using the System Tray" on page 4-81.

To play a sound:

```
AbstractPepperProgram.getGSP().playSound();
```

To show the Progress Bar and launch an object that implements runnable:

```
AbstractPepperProgram.getGSP().showProgressBar();
```

This section also contains information about Tab control.

See "Tab control" on page 4-81.

Writing messages to the framework status bar

You can write messages to the framework Status Bar that either disappear after five seconds or that last until the next message replaces them.

> **Note:** Status bar message queueing is not implemented in the current release. This means that it is possible for a displayed message to be immediately replaced by another message.

Writing to the Status Bar requires access to the methods provided by the `GuiServicesProvider` interface. Any class that implements this interface or that is derived from another class that implements this interface can write to the Status Bar.

> **Note:** See the API documentation of the `GuiServicesProvider` interface in the SDK javadoc in *pepper-sdk/docs*.

Two examples of writing to the Status Bar.

- Write a Status Bar message that disappears after five seconds:

```
AbstractPepperProgram.getGSP.writeStatus("Here is my message");
```

- Write a Status Bar message that persists until replaced by another message:

```
AbstractPepperProgram.getGSP.writeStatus(“Here is my message, true, false);
```

The first boolean parameter is not used and can be either true or false. The second controls whether the message is transient. When `true`, the message disappears in five seconds. When `false`, the message persists.

Using the System Tray

The System Tray (represented by the class com.pepper.guiutils.SystemTray) is the area in the bottom right-hand corner of the Keeper. The System Tray contains objects (com.pepper.guiutils.SystemTrayObject) that represent System-level resources, such as Wi-Fi status, the battery gauge, and the system clock.

Applications can access the System Tray and add application-specific objects to it.

To access the System Tray:

```
SystemTray systemTray = AbstractPepperProgram.getGSP().getSystemTray();
```

To add an object to the System Tray:

```
SystemTrayObject stObject = new SystemTrayObject(String uniqueId,
                                                 String displayName,
                                                 ImageIcon image);
systemTray.add(stObject);
```

To retrieve a System Tray object by id:

```
SystemTrayObject myObj = systemTray.getSystemTrayObject(String objectId);
```

Tab control

An application can allow (or prevent) the user from creating new Tab (Section) instances, deleting Tab instances, moving Tabs and renaming tabs. This is called *Tab control*.

Note: *Tab* is the user interface term that is equivalent to a Section.

By default, user Tab control is enabled.

Unless specified programatically, new Tabs are of the Section type declared as `type=”default”` in *FactoryBuild.xml* and use the SectionPage type defined as `type=”default”` in *PageTemplates.xml.*

Tab control is set at the application level. The Tab control setting applies to all Tabs in the application.

Tab control is configured in the application’s base class (the class that extends `AbstractPepperProgram`).

To disable Tab control, insert the following method in the base class:

```
public boolean enableTabControls(){
   return false;
}
```

Mime type handling

Your application can register to handle files downloaded from Web Sections that are of a particular mime type.

This involves the following:

Ensuring no other application has registered for the mime type

You need to review all registered mime types and ensure the one(s) you want to register is not already claimed by another application.

All currently registered mime types from all applications are listed in the Keeper's own *package.xml* file. This file resides in the Keeper's run-time *data* directory.

Data for each application is wrapped in a `<program>` element. The application's registered mime types are each indicated with a `<mimeType>` element.

The following example shows that PhotoLibrary has registered for three mime types.

Example 4–27

```
<program id="file://C:/Documents and Settings/kyle.nitzsche/My Documents/
                Pepper/suite/photolibrary-0/package.ppld"
                packageId="PhotoLibrary-0"
                name="Photo Library" type="photolibrary" updateStatus="new">
   <description />
   <thumbImage>data/thumbnails/PhotoLibrary-0/photo-64.png</thumbImage>
   <mimeType name="image/jpeg" />
   <mimeType name="image/gif" />
   <mimeType name="image/png" />
</program>
```

Registering mime type in *package.ppld*

To register the mime type for your application, include the `<mimeType name="(`*`mime type`*`)"/>` element inside the `<information>` element in the application's *package.ppld* file, where the (*`mime type`*) is the a mime type, as follows:

```
<information>
   <title>Hello World Phase 4</title>
   <packageType>PepperHelloWorldPhase4</packageType>
   <packageGUID>0</packageGUID>
   <mimeType name="image/jpeg"/>
   <pages>100000</pages>
   <vendor>Pepper</vendor>
   <homepage href="http://www.pepper.com"/>
   <description>description</description>
   <icon>design/images/saturn.24w.png</icon>
```

```
    <thumbnail>design/images/saturn.64w.png</thumbnail>
    <deletable>true</deletable>
</information>
```

Handling the new Page in createPage()

When a file is downloaded in a Web Section (for example by a user right-clicking on an image or URL and selecting **Keep Image**), the file is downloaded and the AbstractPepperProgram.createPage(Object, List, PageChangeListener) method of application that has registered for the mime type is called.

You therefore override createPage() in your base class and inside it handle Page creation and display.

See javadoc for AbstractPepperProgram.createPage(Object, List, PageChangeListener).

5

Hello World: Getting Started

Overview

This chapter is an introduction to the Hello World Tutorial. The Hello World Tutorial takes you through four phases of developing a Pepper application. The application is designed to cover many important aspects of the framework.

Prerequisites

Familiarity with the following is required:

- HTML and CSS
- JavaScript
- Java
- XML and XSL

Helpful information

Information required to program Pepper applications is included throughout this guide. For an overview of many important parts of developing Pepper applications, see “Framework and Application Architecture” on page 4-27.

During the course of the tutorial, you build the application and add it to the framework numerous times. This sort of information is covered in the following chapter: “Building Applications” on page 11-189.

Tutorial structure

The Hello World Tutorial has four phases.

- In Phase One, you copy source files from the *SDK*'s *applicationTemplate* directory and then follow detailed steps to edit critical files.

  Every application requires editing certain critical application definition files.

- In Phases Two, Three and Four, the completed source files are provided.

  You don't actually have to edit the files, although you do have to copy the applications' source files from a resource directory into the working area of the SDK.

  The text explains the important aspects of each phase with examples and references to the provided source files.

The final result of each phase is an application you add to the framework. Although the applications are cumulative, with each phase only adding to the functionality of the previous phase, each phase is also a completely independent application that you add to the framework. At the end of the tutorial, you have added four applications, one for each phase.

> **Tip:** Experienced programmers can start by adding the Phase Four application and then reading the textual material for all phases.

What's next

Now, let's move on to Phase One of the Hello World Tutorial. See "Hello World 1: Application Creation" on page 6-89.

6

Hello World 1: Application Creation

This chapter is the first in a series that explains how to develop the Hello World Pepper application. These chapters demonstrate how to make use of many critical aspects of the Pepper Application Framework architecture.

> **Note:** Directory paths use the Linux convention of forward slashes (‘/’) to separate directories and files.

Creating project directory tree

Every project requires a dedicated application directory with a specific structure of subfolders. In this case, our application directory is named “Phase1”.

The SDK includes a template directory for new applications. In this procedure, you copy the template directory to the correct location and rename it to your project name: *Phase1*.

> For information about SDK directories, see “SDK directories” on page 2-5.

Procedure:

1. Copy the template directory named:

   pepper-sdk/applications/applicationTemplate.
2. Paste the copied directory into:

   pepper-sdk/applications/
3. Rename the new directory *Phase1.*

After completing these steps, you should have a directory named:

pepper-sdk/applications/Phase1

Procedure complete

Creating the required base Java class

Every Pepper application requires a base Java class that extends the AbstractPepperProgram class. The base class must have a specific constructor and `init()` method.

For information, see "AbstractPepperProgram life cycle" on page 4-29.

The SDK includes a *BaseClass.java* template file that extends AbstractPepperProgram and has the required constructor and `init()` method.

However, the Java source file for this class must be in a subdirectory of the application's *src* directory that is consistent with its Java package path. The package path in the application template is incorrect and must be fixed.

In this procedure, you create the required *src* directory and subdirectories and modify *BaseClass.java*'s `package` path accordingly. You also change the class name to `HelloWorld` and change the file name accordingly.

Procedure:

1. Create the base class's package path directories, as follows:
   a. Take note of the subfolders of the following directory:

      pepper-sdk/applications/Phase1/src

      The subfolders are:

      com/placeholder/appName

   b. Change the *pepper-sdk/applications/Phase1/src/com/placeholder* directory to *pepper-sdk/applications/Phase1/src/com/pepper*
   c. Change the final subfolder name to *HW*.

      Note: The final subfolder (*appName*) is generally used to differentiate applications that share the rest of the package path.

      At this point your *src* directory structure appears as follows:

      src/com/pepper/HW

      In accordance with Java conventions, this determines the package path required by all Java files in *src/com/pepper/HW*.

      Note: For a non-tutorial application, you would probably change the names of all subfolders to create a package path appropriate to your requirements. Many such package paths encode (in reverse order) a unique DNS name in order to ensure a globally unique class identifier. For example, a company with the DNS name yourcompany.com would typically name the first two folders: *com/yourcompany*.
2. Modify the `package` path in the *BaseClass.java* file to reflect the directory structure you just created, as follows:
   a. Open *src/com/pepper/HW/BaseClass.java* for editing.

      The file is as follows:

```
/*
 * Copyright (c) 2006 Pepper Computer, Inc. All Rights Reserved.
```

```
 * PEPPER PROPRIETARY/CONFIDENTIAL.  Use is subject to license terms.
 */

package com.placeholder.appName;

import com.pepper.platform.program.AbstractPepperProgram;
import com.pepper.platform.program.ActionEventWithParams;
import com.pepper.platform.program.PepperProgramConfig;
import com.pepper.platform.program.PepperProgramException;
import com.pepper.platform.program.actions.ProgramAbstractAction;

/**
 * Base class template for Pepper SDK
 */
public class BaseClass extends AbstractPepperProgram {

   /**
    * Constructor
    */
   public BaseClass(Integer pid) {
      super(pid);
   }

   /**
    * First pass config to the superclass with super.init(config)
    * and then perform any required initializations, such
    * as registering Actions.
    */
   public void init(PepperProgramConfig config)
             throws PepperProgramException {
      super.init(config);

   }
}
```

b. Modify the `package` path statement to be the path you created previously.

For example, since your *src* directory structure is:

com/pepper/HW

Then, your `package` path statement must be:

```
package com.pepper.HW;
```

3. Change the class name to `HelloWorld`.

After making the edit, the class declaration is as follows:

```
public class HelloWorld extends AbstractPepperProgram {
```

4. Change the class constructor to `HelloWorld`.

The class constructor should appear as follows:

```
public HelloWorld(Integer pid) {
        super(pid);
    }
```

5. Save and close the file.
6. Rename *BaseClass.java* to *HelloWorld.java*.

Procedure complete.

Modifying *build.xml*

Each application has a *build.xml* file that must be modified to configure the build system to build your application.

The *build.xml* file has a `<project>` element whose `name` attribute has to be set to equal the application's root directory name, in this case *Phase1*.

The file is:

pepper-sdk/applications/Phase1/build/build.xml

Procedure:

1. Modify *pepper-sdk/applications/Phase1/build/build.xml*, as follows:
   a. Open *pepper-sdk/applications/Phase1/build/build.xml.*

      The file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE project [
   <!ENTITY common SYSTEM "file:../../../bootstrap/build/commonbuild.xml">
   <!ENTITY package SYSTEM "file:../../../bootstrap/build/packagebuild.xml">

]>

<project name="TODO:applicationDirectoryName" default="all" basedir=".">

   &common;
   &package;

</project>
```

2. Modify the `project` element's `name` attribute to be `Phase1`, as follows:

```
<project name="Phase1" default="app" basedir=".">
```

3. Save and close the file.

Procedure complete.

Modifying *package.ppld*

The *(application)/design/package.ppld* file provides critical information used to install the application in the framework and is used by the framework at run time. Some customizations are required, others are optional.

For reference information about this file, see "package.ppld" on page B-208.

In this procedure, you customize this file for Phase One of the Hello World application.

Procedure:

1. Open *pepper-sdk/applications/Phase1/design/package.ppld* in an appropriate text editor.

   The file is as follows:

```
<jnlp spec="1.0+">

   <information>
      <title>TODO:displayedApplicationTitle</title>
      <packageType>TODO:com.yourCompany.uniqueApplicationType</packageType>
      <packageGUID>0</packageGUID>
      <vendor>TODO-optional:yourName</vendor>
      <homepage href="TODO-optional:yourHomePage"/>
      <description>TODO-optional:applicationDescription</description>
      <icon>TODO:design/images/filename</icon>
      <thumbnail>TODO:design/images/filename</thumbnail>
      <deletable>true</deletable>
   </information>

   <security>
      <all-permissions/>
   </security>

   <resources>
      <jar href="TODO:add application jar file"/>
      <jar href="data.zip"/>
      <jar href="design.zip"/>
   </resources>

   <application-desc main-class="TODO:add package path to required base class">
      <packageVersion>TODO:set to equal application version</packageVersion>
      <requiredKeeperVersion>TODO:set to the minimum required
         Keeper version (3.0.3 or higher)</requiredKeeperVersion>
   </application-desc>

</jnlp>
```

2. Set the application's title, as follows:

```
<title>Hello World Phase 1</title>
```

   The `<title>` element is used to derive the application's run-time directory name. It also provides the text displayed beneath the application's icon in the framework's **Applications** Tab.

3. Set the application's type as follows:

```
<packageType>com.pepper.HelloWorldPhase1</packageType>
```

The <packagetype> uniquely and globally identifies the application type. This is used, among other things, when the application identifies itself to an update server. If an application of that type exists on the server, it is used to update the local application from the server. Therefore, in order to ensure that an application is not accidentally updated from a *different* application on the update server that happens to have the same <packageType>, it is necessary to ensure every application has a globally unique type.

The following type naming convention achieves globally unique application typing for every application:

- Start with the company DNS name in reverse order, delimited with periods, following the Java convention for package names.

  For example, Pepper Computer, Inc. has the following DNS name: pepper.com.

  Reversing this yields: com.pepper.

- Finish with a description of the application type that is unique in your company (without any spaces).

  For example, the application we are developing now is Hello World Phase One.

  Removing the spaces yields a type of: HelloWorldPhase1

- Putting the two together yields: com.pepper.HelloWorldPhase1.

Note: Even though the <packageType> and the *package.ppld's* <main-class> both use a reverse encoded DNS name, they are unrelated.

4. Optionally identify the application developer, as follows:

```
<vendor>YourName</vendor>
```

Note: This is not displayed in the application unless you take programmatic steps to do so.

5. Optionally identify a URL to a web page (starting with `http://`), as follows:

```
<homepage href="http://www.yourHomePage.com"/>
```

Note: You must provide a complete URL, including "http://".

Note: The homepage is not displayed in the application unless you take programmatic steps to do so.

6. Provide a path to a 24 x 24 pixel icon image file used to represent the application on the framework Flag Panel, as follows:

```
<icon>design/images/saturn.24w.png</icon>
```

Icons and other resource files must be stored in the *design* directory or one of its subdirectories. Typically, images are stored in *design/images*. This particular file, *saturn.24w.png*, is included with the SDK in the *applicationTemplate/design/images* folder and was copied to the *Phase1/design/images* chapter previously.

7. Provide a path to a 64 x 64 pixel thumbnail image file used to represent the application on the framework **Application** tab, as follows:

```
<thumbnail>design/images/saturn.64w.png</thumbnail>
```

8. Set whether the application is deletable by the user.

```
<deletable>true</deletable>
```

Note: Most applications should be deletable by the user.

9. Set the application's jar file name, as follows: (**bold**):

```
<jar href="Phase1.jar" />
```

Note: The jar file must be named the same as the application's root directory, also know as the *project name*.

10. Indicate the class that is the entry point into the application.

Use the package path plus the Java class that extends AbstractPepperProgram, delimited with periods (“.”), as follows (**bold**):

```
<application-desc main-class="com.pepper.HW.HelloWorld">
```

11. Set the application's version, as follows.

```
<packageVersion>1.0</packageVersion>
```

The format supports two to four numbers delimited by periods. This format is consistent with the following application revision numbering scheme:

major.minor.patch.build

For applications distributed from an update server, the version is used to determine whether the installed application needs an update.

12. Set the minimum version of the framework required for this application, as follows.

```
<requiredKeeperVersion>3.0.3</requiredKeeperVersion>
```

Note: Applications developed in the SDK require a framework of at least revision 3.0.3.

13. Save and close the file.

Procedure complete.

Customizing *FactoryBuild.xml*

An application's initial Sections are declared in:

(application)/design/FactoryBuild.xml.

For information about Sections, see “Framework and Application Architecture” on page 4-27.

In this procedure, you customize *FactoryBuild.xml* to create the simplest possible application: one with a single Section. This Section has a SectionPage that displays “Hello World!”

Procedure:

1. Open *pepper-sdk/applications/Phase1/design/FactoryBuild.xml* in an appropriate text editor.

The file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<factoryBuild>

   <!-- Definition of Sections -->

   <packageList>

      <!--Template for a new Section -->
      <!--
      <section name="NameKey.TODO:YourKey" type="default" id="data/TODO:yourSectionName"
         deletable="false" />
        -->

      <!-- Definition of Web Browsing Section with bookmarks -->
      <!--
      <section name="NameKey.WebSection" type="web" id="data/web" builtin="false">
         <prebuiltPagesDir>../prebuilt/websection</prebuiltPagesDir>
      </section>
      -->

      <!-- Help section -->
      <!--
      <section name="NameKey.Help" type="help" builtin="true" id="data/help/locale(Help)"
         src="../prebuilt/helpsection" />
      -->

      <!-- Settings Section -->
      <!--
      <section name="NameKey.Settings" type="settings" id="data/settings"
         builtin="true" />
      -->

   </packageList>
</factoryBuild>
```

Notice that:

- All <section> elements are commented out, which means this default template does not produce any Sections (Tabs, from the user interface perspective).
- The first <section> element is a template for a new Section.
- The remaining <section> elements, although commented out, define three ready-made Section types easily includable in any application: a web browser Section, a help Section, and a Settings Section. These sections are added to Hello World later.

2. Modify the template to declare a new Section, as follows.

   Note: The modifications are displayed in **bold** text.

```
   <!-- Definition of the Hello World Section -->

   <!--
   <section name="NameKey.HelloWorld" type="default"
                  id="data/helloWorldMain" deletable="false" />
     -->
```

3. Remove the comment markers around the modified `<section>` element, so it appears as follows:

```
<!--Defintion of the Hello World Section -->

<section name="NameKey.HelloWorld" type="default"
                    id="data/helloworldMain" deletable="false" />
```

The `<section>` element's `name` attribute specifies a key for a key-value pair stored in the *pepper-sdk/applications/Phase1/design/PackageStrings.properties* file. The framework references the specified key (`NameKey.HelloWorld`) in that file and uses its value as the display text for the Section's Tab in the framework. Editing this file to add the key-value pair is covered in the next procedure.

4. Save and close the file.

Procedure complete.

Customizing *PackageStrings.properties*

As noted in the previous procedure, *PackageStrings.properties* enables you to set the text displayed on a Tab.

> **Note:** Also use this file for setting other displayed text and for creating different versions of an application for different language. For information, see "How to localize for different languages" on page 10-177.

In this procedure, you edit the file to add a key-value pair for the Section definition created in the previous procedure.

Procedure:

1. Open *design/PackageStrings.properties* for editing.

   The file is as follows:

```
#Labels on new Section Tab
#TODO: Create labels for any new Section Tabs

#TODO: Uncomment these when adding ready-made Web, Help and Settings Tabs
#NameKey.WebSection=Web
#NameKey.Help=Help
#NameKey.Settings=Settings
```

2. Replace the first commented out TODO line with a new key-value pair that corresponds to the entry you just made in the *FactoryBuild.xml* file, as shown in the following **bold** text:

```
#Labels on new Section Tab
NameKey.HelloWorld=Hello World Main Tab
```

3. Save and close the file.

Procedure complete.

Defining the SectionPage in *PageTemplates.xml*

Each Section must have one SectionPage. The SectionPage displays in the Section's Tab. The SectionPage is defined with the `<sectionPage>` element in: *design/PageTemplates.xml*

> **Note:** For reference information about elements in *PageTemplates.xml*, see "PageTemplates.xml" on page B-222.

> **Note:** For information about Sections and Pages, see "Framework and Application Architecture" on page 4-27.

In this procedure, you set two required general values and then define a SectionPage that displays by default in the Hello World Application's Hello World Tab.

> **Note:** Even after editing this file is complete, the application is not ready to build and run because it is still necessary to create an XSL file that transforms the SectionPage XML into renderable HTML at run time. (See "The Section user interface" on page 4-44.) You do need to name this XSL file here.

Procedure:

1. Open *pepper-sdk/applications/Phase1/design/PageTemplates.xml* for editing.

   The file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<pageTemplates xmlns:xi="http://www.w3.org/2003/XInclude">
   <packageName>TODO:set to equal package.ppld's packageType
      element</packageName>
   <packageVersion>TODO:set to equal package.ppld's requiredKeeperVersion
      element</packageVersion>

   <!-- Define default page for new application -->
   <sectionPage type="default">
      <template>design/TODO:xslFileName</template>
   </sectionPage>

   <!-- Include ready-made page definitions  -->
   <!-- Required for Web Sections -->
   <xi:include href="../resources/pages/SectionWeb.xml" />
   <xi:include href="../resources/pages/Bookmark.xml" />
   <xi:include href="../resources/pages/Clipping.xml" />

   <!-- Required for Settings Section -->
   <xi:include href="../resources/pages/SectionSettings.xml" />

   <!-- Required for Help Sections -->
   <xi:include href="../resources/pages/SectionHelp.xml" />

   <!-- Required for Java Sections -->
   <xi:include href="../resources/pages/SectionJava.xml" />

</pageTemplates>
```

> **Note:** The file includes XML files with `<xi:include>` elements, each of which defines ready-made SectionPage types that are provided with the SDK.

2. Enter `com.pepper.HelloWorldPhase1` as the application name into the `<packageName>` element, as shown in **bold** text:

```
<packageName>com.pepper.HelloWorldPhase1</packageName>
```

   The package name must be set to equal the `<packageType>` in *package.ppld*.

3. Enter the package version into the `packageVersion` element, as shown in **bold** text:

```
<packageVersion>1.0</packageVersion>
```

4. Enter `design/helloWorldMain.xsl` into the `template` element.

   This is the name of the XSL file that transforms this SectionPage into renderable HTML.

```
<template>design/helloWorldMain.xsl</template>
```

   Note: All application-specific XSL files must reside in the *design* directory. This XSL file is created in the next procedure.

5. Save and close the file.

Procedure complete.

Creating the main XSL file from *sample.xsl*

Most Pages require an XSL file that is used by the framework to transform the XML into HTML for rendering and display in the framework.

For information about the role of XSL files, see "The Section user interface" on page 4-44.

Note: Some Pages, such as those in help Sections, name an HTML file in their `<template>` element. In such cases, the specified HTML is rendered directly and no XSL is used. For information, see "The Section user interface" on page 4-44.

In this procedure, you rename the *sample.xsl* file to *helloWorldMain.xsl*, as you specified for the SectionPage in the previous procedure. You also open and examine the new XSL file.

Procedure:

1. In the *design* directory, rename *sample.xsl helloWorldMain.xsl*.
2. Open *helloWorldMain.xsl* in an appropriate text editor for viewing.

   The file is as follows:

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xalan="http://xml.apache.org/xalan"
   xmlns:pepper="http://www.pepper.com/elements"
   extension-element-prefixes="pepper">

   <xalan:component prefix="pepper">
      <xalan:script lang="javaclass" src="com.pepper.script.Elements"/>
   </xalan:component>

   <xsl:output method="html" />
```

```
   <!-- Catch passed framework parameters in XSL -->
   <xsl:param name="packageId" />
   <xsl:param name="platform" />
   <xsl:param name="design" />

   <!--Create XSL variable for use as HTML <title> -->
   <xsl:variable name="sectionName" select="pepper:nameString($packageId,section/@name)"/>

   <!-- The following template matches the root element of the Page data
          to begin the transformation -->
   <xsl:template match="/">
      <html xmlns="http://www.w3.org/1999/xhtml">
         <head>

            <!-- use XSL sectionName variable to set HTML <title> -->
            <title><xsl:value-of select="$sectionName"/></title>

            <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

            <!-- The following can be used to link the Pepper CSS stylesheet into
                  the generated HTML:
            <link href="{$platform}/resources/styles/styles.css" rel="stylesheet"
                  type="text/css" />
            -->

            <!-- The following initializes the JavaScript bridge and is required: -->
            <pepper:initscriptbridge package="{$packageId}" />

            <!-- The following can be used to include ready-made javascript code
            <script type="text/javascript" src="{$platform}/resources/commonsection.js"/>
            -->

         </head>
         <body>
            <h1>Hello World!</h1>
         </body>
      </html>
   </xsl:template>
</xsl:stylesheet>
```

Note: Some knowledge of XSL is a prerequisite for programming Pepper applications.

3. Examine the file, observing the following:
   - The output is set to HTML with the `<xsl:output method="html"/>` element.
   - Three framework parameters are passed to the transform and are available for programmatic use during the transform:

```
<xsl:param name="packageId" />
<xsl:param name="platform" />
<xsl:param name="design" />
```

Only `platform` is used at this early stage of Hello World. It is used to initialize the LiveConnect JavaScript-to-Java bridge with the `<pepper:initscriptbridge>` element.

- There's a single XSL template that matches the root element of the XML file.

  Upon matching the root element, the transform outputs a simple HTML page whose meaningful content is simply "Hello World!" enclosed in an HTML `<h1>` tag.

  Note: The HTML character encoding is set to the utf-8 in a `<meta>` tag. All generated HTML should use UTF-8 character encoding.

4. Close *helloWorldMain.xsl*.

Procedure complete.

Building Hello World

Now that you have created all files necessary for this initial version of the Hello World application, it's time to build the application. Building the application creates (or if it has already been created, updates) the *dist* directory, which contains all files needed to add the application to a framework.

For information about the *dist* directory and the application distribution package, see "Application distribution package" on page 4-31.

Three steps are required to build an application.

1. Customizing the application's *build.xml* file.

   You have already completed this step.

2. Opening a command line window to use for building, and setting its environment variables.

   You need to complete this step.

   See "Setting the Build Environment Variables" on page 11-190.

3. Building the application.

   You need to complete this step.

   See "Building an Application" on page 11-191.

Adding the application to the framework

Now that the application is built, you can add it to a framework.

Because the framework and its applications can run on a Windows system and on Pepper devices, and because you can develop your applications on Windows or on Linux, different procedures are followed to add your application to the framework.

- If you are developing your application on a Windows system, it is probably most convenient to try out applications on the Pepper Desktop before adding them to a Pepper device.

  For the procedure to add the application to the Pepper Desktop, see "Adding a local application to the framework" on page 12-201.

  If you want to add the application to a Pepper device, you need to:

  1. Make the built files available to the Pepper device.

See “Making an application’s files accessible to the framework” on page 12-197

2. Add the application to the Pepper device.

   See “Adding a local application to the framework” on page 12-201.

- If you are developing your application on a Linux system, you probably need to try out the application on the Pepper device itself, which requires making the application’s built files accessible to the Pepper device.

  For the options and procedures for making the built files available to the Pepper device, see “Making an application’s files accessible to the framework” on page 12-197.

  Then, follow the procedure to add the application to the Pepper device: “Adding a local application to the framework” on page 12-201.

Adding ready-made Sections

So far, you have created an application that offers a bare minimum of functionality: a single Section named “Hello World Main Tab” with a SectionPage that displays the text “Hello World!”

This procedure shows how to add additional tabs of ready-made types, including:

- A Web Tab

  Provides a fully functional browser with four pre-built bookmarks.

  Note: The set of bookmarks that are provided by default with a Web section can be customized. See “How to customize web bookmarks” on page 10-172.

- A Help Tab

  Provides default help information that you can customize for your application.

  Note: Help can be customized. See “How to customize an application’s help” on page 10-175.

- A Settings Tab

  Provides a framework for user modification of the application’s settings.

Adding these three Sections is simple because they are ready-made and included with the SDK. However, a few actions are required:

- *FactoryBuild.xml* needs to be modified to add the new Section definitions.
- *PageTemplates.xml* needs to include the new Page definitions.

  These are included by default. The procedure below instructs you to open the file and take note of which includes correspond to which Section types.

- *PackageStrings.properties* needs to be modified to add key-value pairs that provide display text for the Section Tabs.

  Note: You do not need to provide XSL files to transform the ready-made Pages’ XML because they are provided with the SDK.

These Sections are defined in *FactoryBuild.xml* as `builtin` by setting their `builtin` attribute to `true`. This affects the position of their Tabs, particularly with respect to their position relative to any new Tabs that are created. See Figure 3–1 on page 3-24 and "Attribute: builtin" on page B-220.

Procedure:

1. Modify *FactoryBuild.xml* to add (by uncommenting) the three new Sections, as follows:
   a. Open *pepper-sdk/applications/Phase1/design/FactoryBuild.xml* for editing.
   b. Find the three commented out `<section>` elements that define the Web, Help and Settings Sections.
   c. Uncomment these three `<section>` elements.

      At this point, the `<section>` elements should be as follows:

```
<!-- Definition of the Hello World Section -->

<section name="NameKey.HelloWorld" type="default"
       id="data/helloworldMain" deletable="false" builtin="false" />

<!-- Definition of Web Browsing Section with bookmarks -->

<section name="NameKey.WebSection" type="web" id="data/web" builtin="false">
   <prebuiltPagesDir>../prebuilt/websection</prebuiltPagesDir>
</section>

<!-- Definition of Help section -->

<section name="NameKey.Help" type="help" builtin="true"
       id="data/help/locale(Help)" src="../prebuilt/helpsection" />

<!-- Definition of Settings Section -->

<section name="NameKey.Settings" type="settings"
       id="data/settings" builtin="true" />
```

      Note: Web and Help Sections use *pre-built* Pages for bookmarks and help material. For information, see "Pre-built Pages" on page 4-69.

   d. Save and close *FactoryBuild.xml*.
2. Modify *PageTemplates.xml* to include ready-made Page definitions for the new Sections, as follows:
   a. Open *pepper-sdk/applications/Phase1/design/PageTemplates.xml* in an appropriate text editor.
   b. Observe the `<xi:include>` elements near the bottom of the file, which are as follows:

```
<!-- Include ready-made page definitions  -->
   <!-- Required for Web Sections -->
   <xi:include href="../resources/pages/SectionWeb.xml" />
   <xi:include href="../resources/pages/Bookmark.xml" />
   <xi:include href="../resources/pages/Clipping.xml" />

   <!-- Required for Settings Section -->
   <xi:include href="../resources/pages/SectionSettings.xml" />
```

```
<!-- Required for Help Sections -->
<xi:include href="../resources/pages/SectionHelp.xml" />

<!-- Required for Java Sections -->
<xi:include href="../resources/pages/SectionJava.xml" />
```

These `<xi:include>` elements include the contents of the files specified with the `href` attributes into this *PageTemplates.xml* file.

The comments indicate the includes that are required for each ready-made Section type.

The paths in the SDK to the referenced files are not as they appear here because the framework assumes the paths start in the run-time *common-resources.zip* file. To view the Page definitions, open the appropriate files here:

pepper-sdk/lib/common-resources-zip/resources/pages/

These included files contain Page definitions that are needed by the ready-made Sections. For example, you have added a `<section>` to *FactoryBuild.xml* with the following attribute: `type=″web″`. This means there must be a Page definition in *PageTemplates.xml* whose type is web. *SectionWeb.xml*, the first of the included files, provides the web Page definition.

Note: `<xi:include>` elements are only used for ready-made objects that are used across multiple applications and that are included with the SDK. `<xi:include>` elements are not recommended for your applications.

c. Save and close *PageTemplates.xml*.

3. Modify *PackageStrings.properties* to add key-value pairs that provide display text for the Tabs, as follows:

   a. Open *PackageStrings.properties* in an appropriate editor.

   The file is as follows:

```
#Labels on new Section Tab
Namekey.HelloWorld=Hello World Main Tab

#TODO: Uncomment these when adding ready-made Web, Help and Settings Tabs
#NameKey.WebSection=Web
#NameKey.Help=Help
#NameKey.Settings=Settings
```

   b. Uncomment the three lines with the key-value pairs for the Web, Help, and Settings tabs.

   c. Save and close *PageTemplates.xml*.

4. Rebuild Hello World.

   The build procedure was previously discussed.

   See “Building an Application” on page 11-191.

5. Depending on your development model, you may have to take steps to provide the framework with the new build files, as previously discussed.

   See “Refreshing an application in the framework” on page 12-202.

6. Relaunch the Hello World application, and check out the new Web, Help, and Settings Tabs.

Procedure complete.

Customizing the display with CSS

Now that you have generated an HTML page and displayed it, you may want to customize its appearance. CSS stylesheets are used for this. You can link the HTML page to a ready-made stylesheet that is included with the SDK, or link to your own. In either case, the link in the generated HTML is made by using the XSL transform to generate a `<link>` element in the HTML that uses the standard CSS `href`, `rel` and `type` attributes.

Link to the ready-made CSS stylesheet as follows:

```
<link href="{$platform}/resources/styles/styles.css"
      rel="stylesheet" type="text/css" />
```

> **Note:** `$platform` is a framework parameter that provides the path to the *common-resources.zip* file in the framework installation, which includes the ready-made CSS stylesheet. The curly braces are standard XSL syntax for referencing attribue value templates. See "Framework parameters passed to XSL" on page 4-68.
>
> **Tip:** You can familiarize yourself with ready-made CSS styles by opening `styles.css`, which resides here:
>
> *pepper-sdk/lib/common-resources.zip/resources/styles/styles.css*

You can link to your own CSS stylesheet by putting the stylesheet in the *Phase1/design* directory and then linking to it as follows:

```
<link href="{$design}/design/(yourStylesheet).css" rel="stylesheet"
      type="text/css" />
```

> **Note:** `$design` is a framework parameter that provides the path to the run-time *design.zip* file for the application. This file is created by the build and replicates the application's *design* directory in the SDK. See "Framework parameters passed to XSL" on page 4-68.

This procedure modifies Hello World's generated HTML to link to a ready-made style sheet and to link to a stylesheet you create.

Procedure.

1. Open *helloWorldMain.xsl* for editing, and add the `<link>` to the ready-made CSS stylesheet as follows:
   a. Uncomment the following element:

```
<link href="{$platform}/resources/styles/styles.css"
      rel="stylesheet" type="text/css" />
```

   > **Note:** This makes use of the `$platform` parameter discussed previously.

   b. Modify `<h1>Hello World!</h1>` to be `<h1 class="Heading">Hello World!</h1>`.

   > **Note:** `Heading` is a class in `styles.css`

   c. Save the file.

d. Observe the different display style of “Hello World!” by rebuilding the application, if necessary, making the new build accessible to the framework, and relaunching Hello World.

Note: From now on it is assumed you can build applications and update the framework with them. These topics are covered in: “Building Applications” on page 11-189.

2. Create your own CSS stylesheet and add a link to it and an HTML element to use it, as follows:

a. Create and open for editing a file named *myStyles.css* in the application’s *design* directory.

b. Add the following text:

```
/* My styles */

Div.myPage {
/* Contain all page HTML and position it below the ToolBar */
    position:fixed;
    top: 32px;
}

.myFirstStyle {
    font-family:arial, helvetica, sans-serif;
    font-size: 24px;
    line-height: 32px;
    color: 00aaff;
}
```

c. Save the file.

a. In *helloWorldMain.xsl*, insert the following `<link>` element right after the `<link>` element you uncommented previously:

```
<link href="{$design}/design/myStyles.css" rel="stylesheet" type="text/css" />
```

b. Also in *helloWorldMain.xsl*, modify the `<body>` element to be as follows:

```
<body>
    <!--create a div with class myPage to contain and position all page contents -->
    <div class="myPage">

    <!-- Display Hello World messages -->
        <h1 class="Heading">Hello World!</h1>
        <div class="myFirstStyle">Hello World!</div>
    </div>
</body>
```

The `<div class=”myPage”>` element encloses all the page’s HTML and positions it using the `myPage` class in *myStyles.css*.

The `<h1 class="myFirstStyle">` element display’s “Hello World!” using the `myFirstStyle` class in *myStyles.css*.

c. Save the file.

d. Rebuild the application and relaunch Hello World.

Procedure complete.

What's next

This completes the first phase of the Hello World application.

The next phase adds Pages and toolbars.

See "Hello World 2: Pages and ToolBars" on page 7-109.

7

Hello World 2: Pages and ToolBars

This chapter is Phase Two of the Hello World application development tutorial.

Overview

By the end of the last chapter, you created a Hello World application that had a Section with a single Page (its SectionPage) that, when transformed by XSL, resulted in HTML whose display styling you controlled with CSS stylesheets. In this chapter, you develop the application further.

However, first note that this chapter is organized differently from the first Hello World chapter. In the first Hello World chapter, the procedures provided detailed steps for making each and every edit to each and every file.

This chapter takes a different approach:

- This chapter starts with a description of the revised Hello World application, including a look at its user interface.
- Then, key topics necessary for developing the revised Hello World are discussed and explained.
- The revised source files are provided with the SDK in the following directory: *pepper-sdk/applications/HelloWorldResources/Phase2/design/*.

  Note: The discussions assume you examine the relevant source files.
- Finally, we provide a few simple procedures for creating the necessary revised source files in your SDK directory and building and launching the revised application.

Description of the revised Hello World application

There's a new Section that displays a table of "worlds". (The first time the program runs, there are no worlds, but users can add them.)

All worlds are listed in the table by their names. But, there is more to a world than just its name. Each world has a number of other characteristics, such as days in a year, hours in the day, whether

it has life, and so on. All world characteristics are defined on another Page, of which there is one instance for each world.

- Figure 7–1 on page 7-110 shows the “worlds” SectionPage that lists all worlds.
- Figure 7–2 on page 7-111 shows the “world” Page for a single world instance.

The user can select a world from the table in the main Page (the SectionPage), then:

- Click a toolbar **Edit** button to replace the worlds SectionPage with a world Page that displays the selected world’s characteristics and allows the user to edit them.
- Click a toolbar **Delete** button to delete the selected world’s Page.

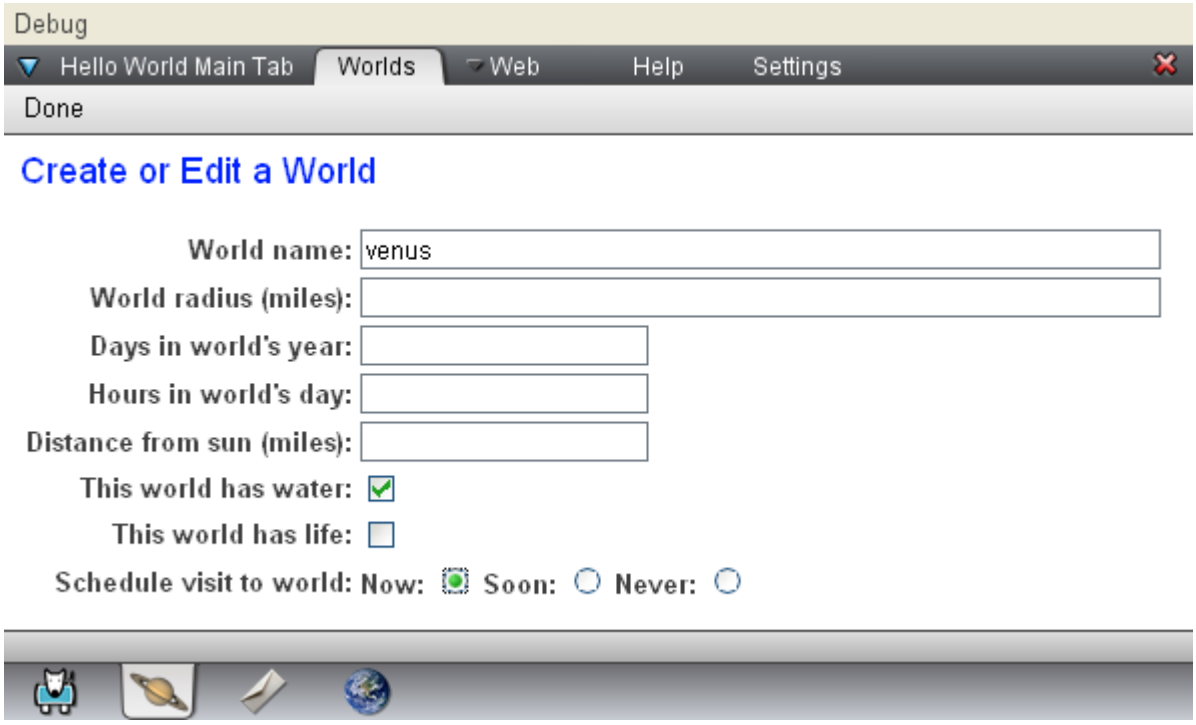
The user selects a world on the worlds SectionPage with a radio button. So, for each world in the table there is a dedicated radio button.

The user can create a world with a toolbar **New** button, which displays an empty world Page they use to define the world.

When the user is done editing (or creating) a world, they click a toolbar **Done** button to bring them back to the worlds SectionPage, which displays an updated table of all worlds.

Figure 7–1 Worlds SectionPage

Figure 7–2 World Page



Now let's drill into the programmatic details to scope out the work that needs to be done to develop this application.

Hello World's programmatic structure

Let's start with the Section declaration and Page definitions, then show how these result in a run-time instantiation of Pages.

> **Note:** For information about Sections, Pages, their XML files and their dynamic run-time structure, see "Sections and Pages" on page 4-32.

Section declarations, Page definitions and caching rules

- There's a new Section.

  A new Section declaration is added to *FactoryBuild.xml*. Its displayed Tab text derives from a new key-value pair (the key is `NameKey.Worlds`) added to *PackageStrings.properites.*

  See *FactoryBuild.xml* and *PackageStrings.properties.xml* in:

  pepper-sdk/applications/HelloWorldResources/Phase2/design/

- There's a new worlds SectionPage for the new Section.

See *PageTemplates.xml* in:

pepper-sdk/applications/HelloWorldResources/Phase2/design/

The SectionPage definition is enclosed in a `<sectionPage type="worlds">` element.

The SectionPage's new XSL stylesheet is indicated with the `<template>design/worlds.xsl</template>` element.

The SectionPage definition specifies its default Page type with the `<defaultPageType>world</defaultPageType>` element. Notice that `world` is the exact `type` defined in the Page explained below. This determines the type of new Pages added to the Section at run-time when the type is not specified.

The SectionPage has to implement cache rules to acquire each Page's `<worldName>` element in order to display world names in the table. This is specified with the `<cacheRules>` element.

For information about caching, see "Caching" on page 4-53.

The SectionPage has a table created from the cached `<worldName>` elements. This is implemented through the SectionPage's XSL stylesheet, not the Page definition. This is discussed later.

Observe also that the main SectionPage has no elements defined for itself, other than the required, empty `<section />` element.

- The Page definition for the world Page specifies a set of elements, each of which holds a specific piece of user-editable data about the world, such as its name, the number of days in its year, and so on.

  See *PageTemplates.xml* in:

  pepper-sdk/applications/HelloWorldResources/Phase2/design/

  As covered previously, the Page's `<worldName>` element is the source of the caching into the parent SectionPage for use in the displayed table of worlds.

  As with all Pages, the new Page needs an XSL stylesheet whose name is specified as *world.xsl* in its `<template>design/world.xsl</template>` element.

A run-time instantiation

Now, let's take a look at a possible run-time instantiation of these declarations and definitions.

Suppose a user has created two worlds. The result would be a run-time structure with a single SectionPage XML instance file (for the table of worlds) and two Page XML instance files (one for each created world).

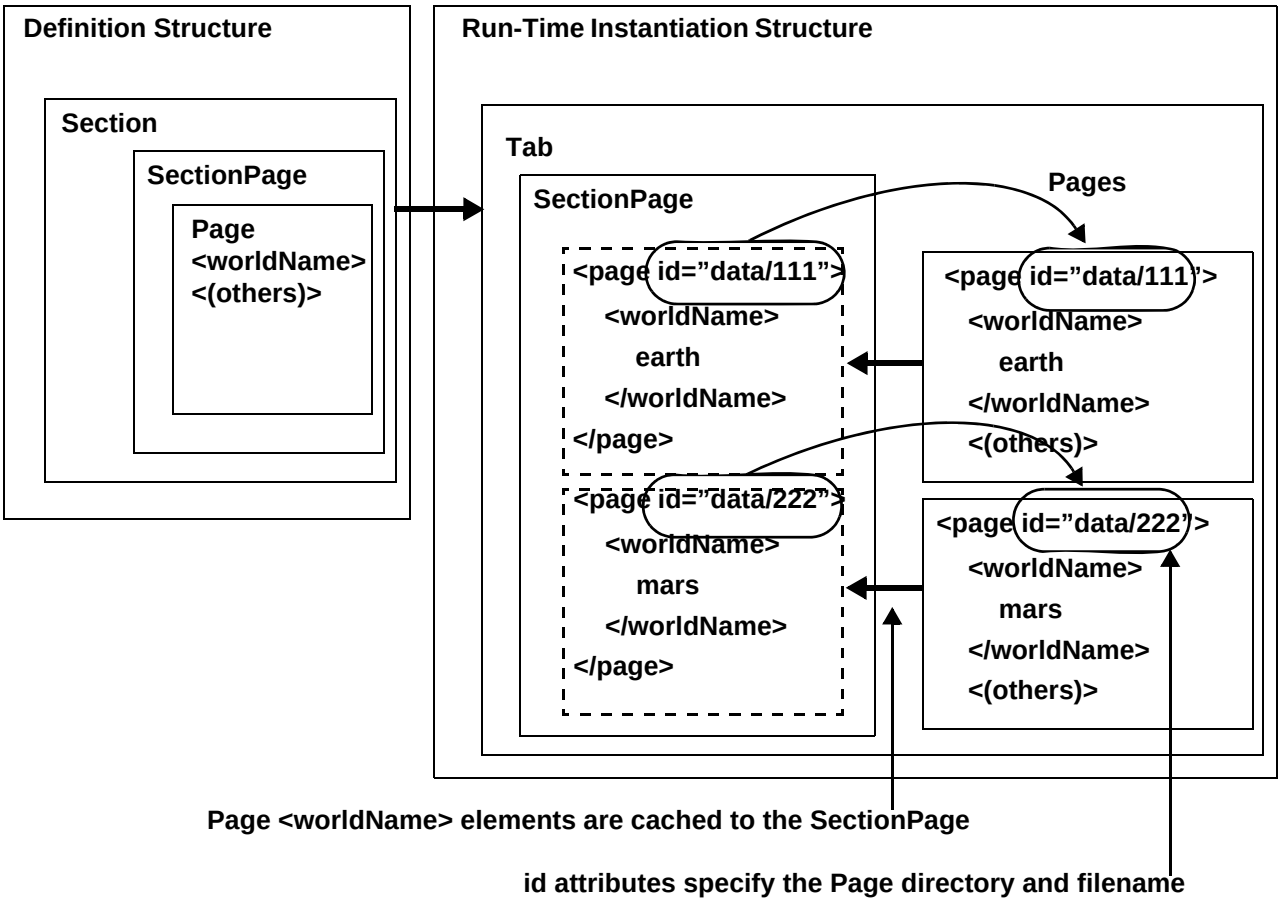
The SectionPage XML instance file has a `<page>` element for each Page instance. Each `<page>` element's `id` attribute specifies the Page's XML instance file. And, each of these `<page>` elements includes the cached `<worldname>` element.

This is shown in Figure 7–3 on page 7-113.

See the complete XML instance files here:

- "Worlds SectionPage sample" on page A-203
- "World Page sample" on page A-204

Figure 7–3 A sample run-time structure with caching



Toolbar buttons link to JavaScript and framework Actions

Before discussing the XSL stylesheets that actually create Pages (including each Page's toolbar and buttons), let's consider the two new toolbars.

> **Note:** For additional information, see "Generating Page toolbars" on page 4-47.

The worlds SectionPage toolbar has the following three buttons:

- **New**

  This directly calls the NewPage framework Action and passes it the type of Page to create. In this case, we pass `world`, which results in replacing the current SectionPage with a new instance of a Page of type `world`.

  For information about Actions, see "Java Actions" on page 4-66.

- **Edit**

  This first calls a JavaScript function whose purpose is to determine which radio button is selected, then to pass its value to the framework ShowPage Action.

As noted, there's a radio button for each world Page. The radio buttons are in a group named `worldPicker`. Each radio button has a `value` that is set to the `id` of the Page for the particular world.

After the JavaScript determines the `id` of the Page for the selected world, it calls the ShowPage framework Action and passes it the `id`, which it has first bundled into a framework URL as required by the ShowPage Action.

For information about using JavaScript to trigger Java Actions, see "JavaScript and Mozilla LiveConnect" on page 4-62.

- **Delete**

  Like Edit, this first calls a JavaScript function to determine the `id` of the selected world. The function then calls the DeletePage framework Action and passes it the `id`, which it has first converted it into array (with one element), as required by the Show Page Action.

The world Page toolbar has one button:

- **Done**

  This directly calls the ShowTabContents framework Action, which replaces the current Page with the Section's SectionPage.

  The world Page toolbar requires no JavaScript.

Now that we have an understanding of the way the toolbar buttons work, let's consider implementation details.

Creating the toolbars: a closer look

Toolbars in the SectionPage and Page HTML are generated by each Page's XSL stylesheet. We'll look at each separately next.

Creating toolbars and buttons in *worlds.xsl*

The following snippet from the SectionPage's *worlds.xsl* file creates the world SectionPage's toolbar and buttons.

Example 7–1 Creating toolbars and buttons in worlds.xsl

```
<pepper:pagebar class="PageBar" id="pb-{$packageId}" package="{$packageId}">
<!-- create three toolbar buttons -->
   <pepper:pagebarentry key="PageBar.New" action="NewPage" param="world" />
   <pepper:pagebarentry key="PageBar.Edit" script="showSelectedPage()" />
   <pepper:pagebarentry key="PageBar.Delete" script="deleteSelectedWorld()" />
</pepper:pagebar>
```

- The toolbar itself is the `<pepper:pagebar>` element, with required attributes as shown.

  The `$packageId` XSL parameter, which equals the packageId parameter that is passed to all XSL transforms by the framework, is required to set up the toolbar.

  For information about framework parameters, see "Framework parameters passed to XSL" on page 4-68.

- Each button is a child `<pepper:pagebarentry>` element.

The displayed button text is determined by the `key` attribute value, which is a key-value pair.

The particular keys used here (`Pagebar.New`, `Pagebar.Edit`, and `Pagebar.Delete`) are defined in the framework's *CommonStrings.properties* file, which resides in the *pepper-sdk/lib/common-resources.zip* file, in its *resources/catalogue* subdirectory.

To define application-specific key-value pairs, use the *pepper-sdk/applications/(application)/design/PackageStrings.properties* file.

Each button calls a JavaScript function through with the `script` attribute or a framework Action with the `action` attribute.

Now, let's take a look at how the buttons call framework Actions and JavaScript functions.

- **New** button

  With `action="NewPage"`, the button directly calls the NewPage framework Action. The NewPage Action requires a `param` attribute whose value specifies the type of Page to create. That Page type must be (and is) defined in *PageTemplates.xml*.

  Note: Many framework Actions require parameter(s) of a specific type. Check the SDK API javadoc class documentation for the com.pepper.platform.program.actions package. See "Javadoc" on page 2-20.

- **Edit** button

  With `script="showSelectedPage()"`, the button calls the JavaScript function `showSelectedPage()`, which resides in the *helloWorld.js* file that is included in the generated HTML with a `<script>` element. The `showSelectedPage()` JavaScript function calls the framework ShowPage Action. See "Drilling into helloWorld.js" on page 7-120.

- **Delete** button

  With `script="deleteSelectedWorld()"`, the Delete button calls the JavaScript function `deleteSelectedWorld()`, which also resides in the *helloWorld.js* file. The `deleteSelectedWorld()` JavaScript function calls the framework DeletePage Action. See "Drilling into helloWorld.js" on page 7-120.

Creating toolbars and buttons in *world.xsl*

The following snippet from the Page's *world.xsl* file creates the world Page's toolbar and button.

Example 7–2 Creating toolbars and buttons in world.xsl

```
<pepper:pagebar class="PageBar" id="pb-{$packageId}" package="{$packageId}">
   <!-- create one toolbar button -->
   <pepper:pagebarentry key="PageBar.Done" action="ShowTabContents" />
</pepper:pagebar>
```

The **Done** button directly triggers the ShowTabContents framework Action, which does not require any parameters.

Drilling into *world.xsl*

We've seen how the toolbars are created by the XSL stylesheet source files.

Now let's review the *world.xsl* file.

> **Note:** We start with *world.xsl*, even though it is the Page, and not with *worlds.xsl*, the parent SectionPage, to build up the picture from the bottom up so that we can then more easily see how the SectionPage's *worlds.xsl* is able to create a table of worlds from world XML instance files that were previously created by the user.
>
> **Tip:** It is recommended that you open the *world.xsl* source file included with the SDK in the following location:
>
> *pepper-sdk/applications/HelloWorldResources/Phase2/design/*

- First comes the required `<xsl:stylesheet>` element with required namespace declarations and Xalan XSL processor configuration.

  For information, see "Required XSL namespaces and Xalan configuration" on page 4-69.

- `<xsl:output method="html" />` sets the output file type to HTML.
- Three framework parameters are accepted into the stylesheet with the `<xsl:param>` elements.

  For information, see "Framework parameters passed to XSL" on page 4-68.

- The first (and only) `<xsl:template>` matches the root element (`match="/"`) in the XML instance file.

  The root element is `<page>`. For information, see "SectionPage and non-SectionPage concept" on page 4-35.

- The `<title>` is set from the Page's `<worldName>` element using the framework Elements class `nameString()` method.

  > See javadoc for com.pepper.script.Elements.

  Elements class methods (such as Elements.nameString()) are available because `pepper` is defined as an `extension-element-prefix` and the Xalan XSL processor is configured to access the Elements class for `pepper` prefixed elements.

  > See "Required XSL namespaces and Xalan configuration" on page 4-69.

  The `nameString()` method takes the `$packageId` framework parameter and an xpath that identifies the element from which to retrieve the text. These parameters must be passed using the syntax shown in this example.

  This xpath is an absolute xpath that starts from the root element of the JDOM document object that represent the Page data. The current position in the Page XML with respect to the current XSL transform is irrelevant. That is, that we have already matched the root element in our XSL transform (with the `<xsl:template match="/">` statement) is meaningless to the `nameString()` method, which requires an absolute path to the target element.

  Since our goal here is to obtain the `<page>` element's child `<worldName>` element and use its text to set the HTML `<title>`, the absolute path is `/page/worldName`.

  > **Note:** The `<title>` is not actually displayed in the framework. However, this demonstrates key points required to develop full-featured applications.

- A `<meta>` tag is generated that indicates the mime type ("text/html") and that the character set is (UTF-8).

Note: In the current release of these tutorial files, the XSL processor is not explicitly told to generate HTML files using the UTF-8 character set.

- Two CSS stylesheets are linked in: one is the framework stylesheet, the other is the user-defined stylesheet you created in the last chapter.
- The LiveConnect bridge is initialized with the `<pepper:initscriptbridge package="{$packageId}" />` element.

  See "JavaScript and Mozilla LiveConnect" on page 4-62.
- The Page's toolbar and buttons are created.

  We have already discussed this in detail. See "Creating toolbars and buttons in world.xsl" on page 7-115.
- A `<div class="myPage">` element is added to hold all subsequent HTML and position it below the toolbar.
- A `<div>` element holding a header message is provided, followed by a `<p>` tag that provides vertical white space through the application-specific `verticalSpacer20` CSS style.
- A `<form>` element is provided to enclose a table that includes input elements.

  The `<form>` element must wrap input elements whose data is to be stored in the XML instance file. See "Connecting displayed data to Page data" on page 4-58.
- The `<table>` is created.

  The table has two columns, the first for a text label derived from the *PackageStrings.properties* file and the second for a text input field or checkbox.
- The first cell of each row is created with a `<pepper:textString>` element.

  As we saw previously, the `pepper` prefix connects this element to the Elements.textString() Java method. (See javadoc.) This method takes two parameters that must be passed as attributes, as in this example.

  The `<pepper:textString>` element is used to set the HTML display text from the *PackageStrings.properties* file in order to support *localization*, instead of simply entering the text directly into the XSL transform.

  For information about localization, see "How to localize for different languages" on page 10-177.

  Each key (for example `key="Label.Name"`) is (and must be) unique in *PackageStrings.properties*. The key corresponds to a value, in this case to `World name`. Here is the relevant line in *PackageStrings.properties*:

```
Label.Name=World name
```

  When generating the HTML for display, the framework looks up the key and retrieves the corresponding value.

  See the *PackageStrings.properties* source file at:

  pepper-sdk/applications/HelloWorldResources/Phase2/design/
- The second cell of each row is populated either with an HTML `<input type="text"...>` element, a `<pepper:checkbox ...>` element, or a series of `<pepper:radiobutton ...>` elements.

Let's take a closer look at these input fields to understand how user-entered data is saved into the XML instance file.

`<input type="text"...>` and `<pepper:checkbox ...>` elements have a `value` attribute and an `xpath` attribute. For example:

```
<input type="text" id="distanceFromSun" value="{/page/distanceFromSun}"
                xpath="/page/distanceFromSun" />
```

The `value` attribute's value must be in curly braces and is an absolute XPath expression that locates the item in the Page data XML to display.

For example, `value="{/page/distanceFromSun}"` selects the root node's (`<page>`) `distanceFromSun` child and displays its contents in the generated HTML. In this case, the content is text and is displayed in a text input element. In the case of a `<pepper:checkbox>` element, the value is `true` or `false` in the XML data and is displayed as a checked or non-checked box accordingly (`true` displays as checked, `false` as unchecked).

The `xpath` attribute enables a framework capability to store the user-entered value into the Page XML data. The `xpath` attribute's value is also an absolute XPath location expression, but it is not enclosed in curly braces.

For example, `xpath="/page/hasLife"` selects (in the Page XML data) the `hasLife` child of the root `page` node and saves into it the user entered value, in this case a checkbox's true|false setting.

In the case of the `<pepper:radiobutton...>` element, the `storedvalue` attribute is used instead of the `value` attribute to specify the source of the information to display. This is because the `value` attribute is already used for a different purpose. It is the `value` attribute that differentiates radio buttons with the same `id`. What is stored in the Page XML data is the `value` attribute of the currently selected radio button. It is stored in the element specified with the `xpath` attribute.

> **Note:** For information about the `<pepper:checkbox>` element, see the SDK API javadoc and look at the `checkbox()` method of the Elements class. For information about the `<pepper:radiobutton>` element, look at the Elements class's `radiobutton()` method.

- The rest of the XSL file simply completes the table, the HTML and the stylesheet.

Drilling into *worlds.xsl*

worlds.xsl has the task of generating an HTML page with a table. The number of rows in the table varies depending on the number of worlds the user has created. As we have seen, for each world, there is both:

- An XML instance file for the world Page, and
- A cached `<page>` element (with a child `<worldName>` element) in the *Worlds.xml* instance file.

> **Note:** Each `<page>` element has an `id` attribute whose value specifies the XML instance file for the world Page, which is created automatically through cache rules. For information, see "Caching" on page 4-53.

Let's take a look at the distinctive events in *worlds.xsl*.

- *worlds.xsl* creates an HTML header, links CSS stylesheets, initializes the LiveConnect bridge, and includes an external JavaScript file.
- The HTML body is created, and the toolbar is added.

  The toolbar was previously discussed in "Creating toolbars and buttons in worlds.xsl" on page 7-114.
- An HTML table is created.
- The first row displays text derived from *PackageStrings.properties* by referencing the `Label.Name` key using the `<pepper:textstring>` element, as previously described.
- For the remaining table rows, *worlds.xsl* uses an XSL `for-each` loop to find each `<page>` of `type="world"`.

  For each `<page>` of `type="world"` found, an XSL template is executed that matches `<page>`'s child `<wordName>` element. This template creates a radio button in a group named `worldPicker` whose value is the `id` attribute of the parent `<page>` element, which, as we have seen, specifies the world XML instance Page. Because of how HTML radio button groups work, when the user selects a particular radio button, the value of the group becomes the value of the selected button, which specifies the world XML instance file.

  Note: We are not using the `<pepper:radiobutton>` element here because its purpose is to store the user-selected buttons value in the XML instance file. In this case, we are merely using the radio button to select the row of the table with the desired world, so the standard HTML radio button suffices.

  Note: XUL is supported by Firefox and therefore by the Framework. XUL has facilities for determining a table's selected row. This approach could be followed instead of the radio button approach used here. For an example, see the Remote Desktop application bundled with the SDK.

Let's take a closer look at the XSL `for-each` loop that creates a new table row for each cached world.

The `<xsl:for-each select="section/page[@type='world']"> ... </xsl:for-each>` statement uses the `section/page[@type='world']` XPath expression to find every world Page. The square brackets are used in XSL to filter what is being selected.

Recall that the *worlds.xsl* stylesheet is operating on the data portion (everything inside the `<body>` element) of the SectionPage XML Page file. See an example of this XML instance file here: "Worlds SectionPage sample" on page A-203.

The XPath expression `section/page[@type='world']` says: find every `<section>` element that has a `<page>` child element that has a `type` attribute whose value is `world`. Looking at the XML instance file indicated above, you can see there are four matching cases.

The `<xsl:for-each>` statement iterates with each of these matching cases. With each it applies all matching XSL templates, of which there is one: `<xsl:template match="worldName">`

- The `<xsl:template match="worldName">` template applies sequentially to each the previously selected node and continues building the table.

  First, it creates a radio button whose `value` attribute is set to the parent `<page>` element's `id` attribute. Refer again to the XML instance file, and bear in mind that our current XSL node is the `<worldName>` element. It's parent `<page>` element's `id` attribute specifies the actual XML

instance file of the world Page for example: `data/1148582938729`. This specifies a file named *1148582938729.xml* that resides in the Hello World installation subdirectory named *data*.

Note: The framework adds the *xml* extension to the `id` attribute.

The radio button's `value` attribute (which identifies the Page) is not displayed in the HTML, but is passed by a JavaScript function to a framework Action to edit or delete the selected Page.

- After setting up the radio button, *worlds.xsl* creates the second cell in each table row, simply getting the text saved in the `<worldName>` element and displaying it.
- After this, the table and HTML are completed.

Drilling into *helloWorld.js*

This tutorial assumes knowledge of JavaScript. Our main focus here is here is how the JavaScript functions call framework Actions. However, each function must first derive the value of the selected radio button, which is described next.

Getting the selected radio button value

As noted above, the JavaScript functions `showSelectedPage()`and `deleteSelectedWorld()` both extract the radio button group's value before calling their respective framework Actions. Both JavaScript functions must handle two cases:

- When there is a single radio button, in which case the radio button group is treated syntactically as a scalar data type (not an array), and
- When there are two or more radio buttons, in which case they are treated as an array.

Both functions do this by testing for the length of the radio button group object. If it is undefined, then the current radio button value is accessed as a scalar data type. Otherwise, it is accessed as an array, in which case a for loop is used to iterate through the array to find the particular radio button that is selected, get its array index, and then use that to get the selected button's value.

In all cases, the value of the selected radio button is then loaded into a variable named `xmlId`.

See the *helloWorld.js* source code in the following directory:

pepper-sdk/applications/HelloWorldResources/Phase2/design/

Once the `xmlId` variable has the selected radio button value, it is passed to the appropriate framework Action, as discussed next.

Framework Actions from JavaScript

Framework Actions are called from JavaScript as follows:

```
bridge.action('(ActionName)', (comma-separated parameter list) );
```

The LiveConnect bridge is initialized by the framework as a result of the `<pepper:initscriptbridge>` element earlier in the XSL transform. This initialization occurs after the HTML page has been entirely loaded.

Calling the ShowPage framework Action

The ShowPage Action takes a parameter that identifies a single XML instance Page to display. The parameter must first be packaged into a string that provides a framework URL to the desired XML instance file in a specific form:

```
pepper://HelloWorldPhase2-0/data/1149?isPage=true
```

Where:

- `pepper://HelloWorldPhase2-0` is defined with the $packageID framework parameter,
- `data/1149` is the XML instance file specification derived from the selected radio button, and
- `?isPage=true` is required to complete the URL.

The `showSelectedPage()` function creates the required parameter and calls the ShowPage framework Action, as follows:

```
var url = "pepper://" + packageId + "/" + xmlId + "?isPage=true";
bridge.action('ShowPage', url);
```

As explained above, the `xmlId` variable already specifies the directory and name of the XML instance file (without its *xml* extension, which is added by the framework).

Calling the DeletePage framework Action

The DeletePage framework Action takes a parameter that identifies a single XML Page to delete. The parameter must first be packaged into an array, even if there is only a single Page to delete, as is the case here.

The `deleteSelectedWorld()` function first creates an array with a single element (the Page to delete) and then passes it to the DeletePage framework Action as follows:

```
idArray[0]=xmlId
bridge.action('DeletePage', idArray);
```

As explained above, the `xmlId` variable specifies the directory and name of the XML instance file (without its *xml* extension, which is added by the framework).

Setting up the new source files

This procedure covers creating the working version of the current phase of the Hello World application from source files provided with the SDK.

Procedure:

1. Copy the following directory:

   pepper-sdk/applications/HelloWorldResources/Phase2

2. Paste the copied directory into:

   pepper-sdk/applications/

   You should now have the following directory:

pepper-sdk/applications/Phase2

Procedure complete.

Build, launch and use the revised application

This procedure covers launching the Phase Two of Hello World and observing the changes in XML instance files as worlds are created and deleted.

Procedure:

1. Build and launch the application.

   Note: It is now on it is assumed you can build applications and update the framework with them. For information, see “Building Applications” on page 11-189.

2. Make new worlds and observe how the contents of the XML instance files change by opening the files residing here:

   (KeeperInstallDirectory)/Phase2-0/data

Procedure complete

What’s next

This completes the second phase of the Hello World Tutorial.

The next phase adds Java.

See “Hello World 3: Getting Started with Java” on page 8-123.

8

Hello World 3: Getting Started with Java

This chapter is Phase Three of the Hello World application development tutorial.

This chapter explains how to add a Java Section that contains a Java ToolBar with ToolBarButtons that write user-entered messages to the framework log and to the framework Status Bar.

Overview

This section provides introductory information regarding Phase Three of the Hello World Tutorial.

Prerequisites

- Completion of or familiarity with the previous Hello World application development stages
- A working knowledge of Java
- Framework concepts and requirements explained in “Framework and Application Architecture” on page 4-27
- Java Section concepts and requirements explained in “Java Sections” on page 4-49

Hello World’s new functionality

Phase Three of the Hello World Tutorial adds a Java Section with the following user interface widgets:

- A ToolBar with two ToolBarButtons

  These are instances of framework classes, not Swing classes.

  See javadoc for com.pepper.guiutils.ToolBar and com.pepper.guiutils.ToolBarButton.

  Although you can use Swing JToolbar and JButton classes, using the framework classes enables you to control the ToolBar's colors and fonts (along with those in other Sections and applications) from a single CSS stylesheet.

  See “Customization with CSS” on page 10-154.

  One button retrieves text from a text field and writes it to the framework log.

The other button retrieves text from a text field and writes it to the framework Status bar.

For information about writing to the framework log, see “Event notification” on page 4-77.

For information about viewing the framework log file, see “Keeper event log” on page 2-19.

For information about writing to the framework Status Bar, see “Event notification” on page 4-77.

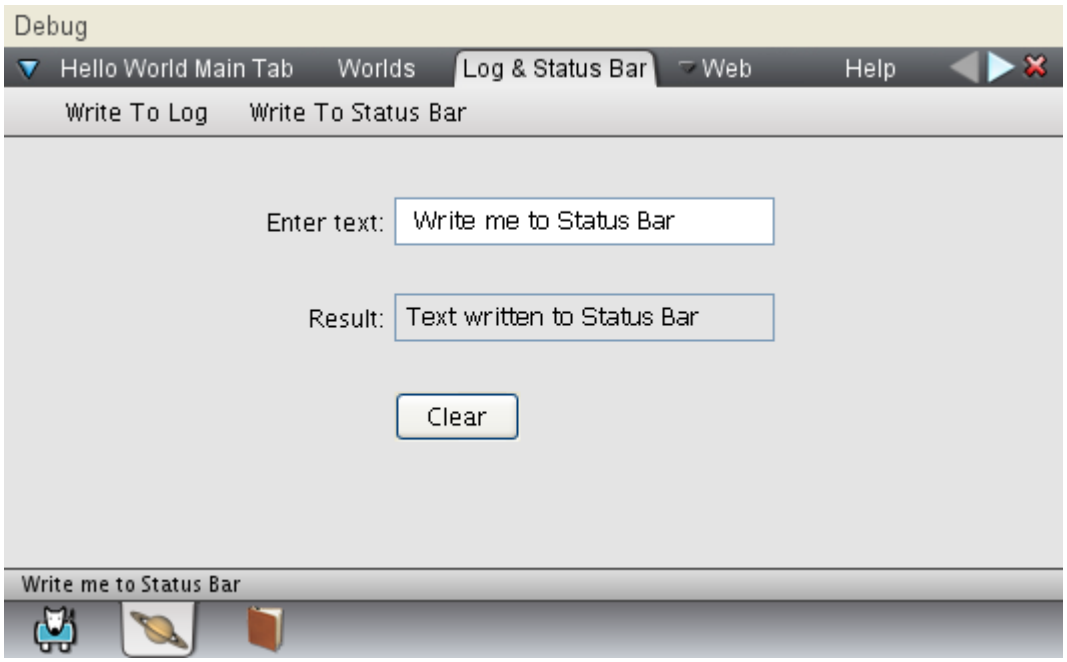
- A JTextField

  The text entered into this field is either written to the log or Status Bar.
- A JTextField that displays messages indicating success of ToolBarButton actions
- A JButton

  Clears the text fields

The following figure shows the revised Hello World application’s new Java Section.

Figure 8–1 Hello World’s new Java Section



Source files

The revised source files for this phase of the Hello World project are here:

pepper-sdk/applications/HelloWorldResources/Phase3/

After the new functionality’s conceptual basis is explained through code examples and discussion, there’s a procedure in which you copy this directory’s source files to a working directory to enable building and running the revised application.

Understanding the code

This section explains the new functionality through code snippets and discussion.

Note: It may be helpful to examine the source files. See "Source files" on page 8-124.

Declaration and definition files

This section explains the modifications required to Hello World's declaration and definition files:

- *FactoryBuild.xml*
- *PackageStrings.properties*
- *PageTemplates.xml*

This section also discusses the *prebuilt* directory. Every Java Section requires a pre-built SectionPage.

After this section, the modifications required to Hello World's Java files are discussed.

FactoryBuild.xml

As explained previously, all Sections must be declared in *FactoryBuild.xml*.

The new Java Section is declared as follows:

Example 8–1 Defining the new Java Section in FactoryBuild.xml

```
<!--Definition of LogStatusbarJavaSection Section -->
<section name="NameKey.LogStatusbarJava" type="java"
                 id="data/LogStatusbarJava" deletable="true"
                 src="../prebuilt/LogStatusbarJava" />
```

Key points:

- The `name` attribute obtains its value from a new key in *PackageStrings.properties.*
- The `type` attribute value is and must be `java` when declaring a Java Section.

  This informs the framework that this is a Java Section.

  For information on Section types, see "<section>" on page B-219.
- The `id` attribute value specifies the filename (without any file extension) of the Java Section's pre-built SectionPage XML file.

  Since Java Sections are pre-built by definition, the pre-built file named by the `id` attribute must be in the application's SDK directory specified by the `src` attribute, as explained next.
- The `src` attribute refers to a directory in which the Java Section's pre-built SectionPage XML file resides.

  For information, see "Pre-built Pages" on page 4-69.

PackageStrings.properties

A new key-value pair is added to *PackageStrings.properties* to contain the text displayed on the Java Section's Tab, as follows (**bold**):

Example 8–2 Defining a new key-value property to provide the Section Tab text

```
#Section text labels
NameKey.HelloWorld=Hello World Main Tab
NameKey.Worlds=Worlds
NameKey.LogStatusbarJava=Log & Status Bar
NameKey.WebSection=Web
```

PageTemplates.xml

An application with a Java Section must have a *PageTemplates.xml* file that includes the ready-made *SectionJava.xml* file. This file includes contains a SectionPage definition for Java Section SectionPages. The line to include *SectionJava.xml is as* follows (in **bold**):

Example 8–3 Including SectionJava.xml into PageTemplates.xml

```
<!-- include shared page and section definitions in platform -->
<xi:include href="../resources/pages/SectionWeb.xml" />
<xi:include href="../resources/pages/Bookmark.xml" />
<xi:include href="../resources/pages/Clipping.xml" />
<xi:include href="../resources/pages/SectionSettings.xml" />
<xi:include href="../resources/pages/SectionHelp.xml" />
<xi:include href="../resources/pages/SectionJava.xml" />
```

The pre-built directory and pre-built Page

Each Java Section requires an *(application)/prebuilt/(sectionDirectory)* directory and Page, where:

- *(sectionDirectory)* is the directory defined in the `src` attribute of the `<section>` element in *FactoryBuild.xml*, and
- The pre-built Page (residing in that directory) has the proper format.

  See "Creating the pre-built Page and specifying the Java class" on page 4-50.

  Note: Usually one just copies an existing pre-built Page for a working Java Section and modifies its filename and its various contained attributes appropriately.

  Note: The filename of the pre-built Java Section XML page can be anything.
- The pre-built Page must have a `<section>` element whose `type` is `java` and whose `id` attribute is that same as the Section's `<section> id` attribute value defined in *FactoryBuild.xml*.
- The pre-built Page must have a `<java>` element whose `classname` attribute identifies the Java class to execute in the Java Section.

The following example shows the pre-built Page.

Example 8–4 Java Section's pre-built Page

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile deletable="false">
   <header>
      <packageName>Hello World Phase 3</packageName>
      <packageVersion>1.0</packageVersion>
      <template></template>
      <noCache>0</noCache>
      <defaultPageType>default</defaultPageType>
      <createDate utc="1121110270523" />
   </header>
   <body>
      <section name="LogStatusbarJava" type="java" id="data/LogStatusbarJava">
         <java classname="com.pepper.HW.LogStatusbarJavaSection" />
      </section>
   </body>
</pageFile>
```

Java Section's Java

This section discusses the Java aspects of the new Java Section.

One new Java source file is required

There is a single Java source file that must be created:

LogStatusbarJavaSection.java

LogStatusbarJavaSection.java provides the Java code that executes in the new Java Section. As such, it implements the rules explained in "Java Sections" on page 4-49, as covered in the following sections.

LogStatusbarJavaSection.java location

LogStatusbarJavaSection.java resides in the application's SDK *src/* directory in a subdirectory consistent with its package path. In this case the package path is:

`com.pepper.HW`

Therefore, *LogStatusbarJavaSection.java* resides here:

pepper-sdk/applications/Phase3/src/com/pepper/HW/

LogStatusbarJavaSection.java therefore must use the following package statement:

```
package com.pepper.HW;
```

Extending java.awt.Component and implementing JavaSectionComponent

Java Section classes:

- Must extend java.awt.Component or a class that extends java.awt.Component, such as JPanel.
- Should implement the com.pepper.platform.program.JavaSectionComponent interface.

  For information, see "The Java Section class" on page 4-50.

  Note: Implementing the JavaSectionComponent interface is only necessary if the class is to interact with the framework. If the Section merely runs Java and does not need to interact with the framework in any way, it does not need to implement this interface. However, such code cannot write to the framework log or to the Status Bar.

The following example shows how the LogStatusbarJavaSection class is declared.

Example 8–5 Declaring the Java Section LogStatusBarJavaSection class

```
public class LogStatusbarJavaSection extends JPanel implements JavaSectionComponent{
    ...
}
```

Implementing initComponent()

Classes that implement the JavaSectionComponent interface must implement initComponent() method. It is not required that the method contain any working code. However, all Java Section initialization should be placed in this method instead of in the constructor to ensure it occurs after the framework has completed its initialization work.

For information, see "Java Sections" on page 4-49.

`initComponent()` passes:

- A reference to the class (AbstractPepperProgram) from which the base class (HelloWorld, in this case) is subclassed.
- A handle to a Properties object.

You can use the reference to the AbstractPepperProgram application base class to gain access to base class methods, such as:

AbstractPepperProgram.getGSP()

See "GUI services" on page 4-80.

getGSP() provides access to the writeStatus() method, which is used to write to the Status Bar, as demonstrated below.

To use the writeStatus() method, use the passed AbstractPepperProgram as follows:

- Declare a local variable of the same type as the base class (HelloWorld, in this case).

  In this code, the `helloWorld` variable is declared.
- Cast the passed AbstractPepperProgram to the base class type (HelloWorld).
- Assign the passed reference to your local variable.

The following snippet declares the `helloWorld` variable.

Example 8–6 Declaring a local variable for the base class

```
/** Handle for acessing <code>HelloWorld</code>, the application's base class. */
public HelloWorld helloWorld;
```

The following snippet shows how to implement `initComponent()`, cast the passed reference to AbstractPepperProgram (`theProg`) to the base class (HelloWorld), and assign it to the local `helloWorld` variable.

Example 8–7 Getting a reference to the base class

```
public void initComponent(AbstractPepperProgram theProg, Properties params) {
   //get local reference to HelloWorld in order to write to the status bar
   helloWorld = (HelloWorld) theProg;
}
```

The `helloWorld` object can now access methods used to write to the Status Bar:

AbstractPepperProgram.getGSP().writeStatus(String)

This is covered further below.

ToolBar and ToolBarButtons

This section covers use of the framework ToolBar and ToolBarButton classes.

For more information on these classes, see “Java ToolBars” on page 4-52.

Using these classes instead of Swing classes ensures that:

- Their look and feel is consistent with other Java toolbars and toolbar buttons.
- Their visual styles (colors and fonts) are derived from the customizable *keeper.css* file.

  For more information on customizing Java visual styling with the *keeper.css* file, see “Keeper.css-based customizations” on page 10-159.

First, let’s discuss creating the ToolBarButtons and their associated actions, then show how to create the ToolBar and add the buttons to it.

Buttons and actions

Each of the two ToolBarButtons and the JButton used to clear the text fields has an action associated with it. Each action is an inner class that extends AbstractAction and overrides actionPerformed(ActionEvent).

The following snippet shows one of the inner Action classes.

Example 8–8

```
protected class WriteToLog extends AbstractAction{
   public void actionPerformed(ActionEvent ev) {
      try{
         log.info(txt_getThis.getText());
         txt_result.setText("Text written to log");
```

```
        }catch (Exception e){
            txt_result.setText("Write to log failed");
        }
    }
}
```

The following snippet shows creation of the three Action objects.

Example 8–9

```
WriteToLog writeLog = new WriteToLog();
WriteToStatusBar writeStatus = new WriteToStatusBar();
ClearButtonAction clearFields = new ClearButtonAction();
```

The following snippet shows how the two ToolBarButtons are created using constructors that receive an Action object and that set the button display text.

Example 8–10

```
ToolBarButton b1 = new ToolBarButton(writeLog, "Write To Log");
ToolBarButton b2 = new ToolBarButton(writeStatus, "Write To Status Bar");
```

Creating the ToolBar and adding ToolBarButtons to it

The ToolBar is created as a class level member, as shown in the following:

Example 8–11

```
/** ToolBar for Section*/
protected ToolBar toolbar = new ToolBar();
```

ToolBarButtons are added to the ToolBar using the `ToolBar.addButton(ToolBarButton)` method, as shown in the following:

Example 8–12

```
toolbar.addButton(b1);
toolbar.addButton(b2);
```

Then, the ToolBar is added to the JPanel. The JPanel is set to use BorderLayout and the ToolBar is added to the BorderLayout.NORTH region, as shown in the following:

Example 8–13

```
this.setLayout(new BorderLayout());
this.add(toolbar, BorderLayout.NORTH);
```

> **Note:** The JButton used to clear the text fields is created with its action and added to the JPanel in a standard Java manner and is not discussed here.

Writing to the log

Writing to the framework log is important for most applications. Let's take a closer look at how this is done.

For more information about writing to the framework log, see “Event notification” on page 4-77.

Writing to the log involves three coding steps:

- Two `import` statements in the class file

  See Example 8–14.
- Creation of a log instance

  See Example 8–15.
- Actually writing to the log with the `log.info(String)` or `log.error(String)` methods

  See Example 8–16.

Example 8–14 Log imports

```
//imports required for writing to the Keeper log
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

Example 8–15 Creating a log instance

```
/** Log instance for writing to Keeper log */
static Log log = LogFactory.getLog("com.pepper.HW");
```

> **Note:** The package path in double quotes should be the same for all log instances in your application, regardless of class. See “Event notification” on page 4-77.

Example 8–16 Writing to the log, see bold text

```
public void actionPerformed (ActionEvent ev){
    //the ActionCommand indicates which button was clicked and therefore
    //whether to write to the Log or the status bar.
    if ("toLog".equals(ev.getActionCommand())){
        log.info(txt_writeToLog.getText());
    }
    if ("toStatusBar".equals(ev.getActionCommand())){
        helloWorld.getGSP().writeStatus(txt_writeToStatusBar.getText());
    }
}
```

Writing to the Status Bar

Writing to the framework Status Bar is also important for most applications. Let’s take a closer look at how that is done.

For more information about writing to the framework Status Bar, see “Event notification” on page 4-77.

Writing to the Status Bar requires access to the AbstractPepperProgram.getGSP().writeStatus(String) method, as discussed previously. This method is accessible from the application’s base class (`helloWorld` in this example).

> **Note:** It was explained above how the this Java Section class obtains a reference named `helloWorld` to the base class.

The following snippet shows how to write to the Status Bar. In this case, the String message that is written is the text entered into the `txt_writeToStatusBar` JTextField, therefore the `txt_writeToStatusBar.getText()` method is used instead of a simple `String` message.

Example 8–17 Writing to the status bar

```
public void actionPerformed (ActionEvent ev){

    //the ActionCommand indicates which button was clicked and therefore
    //whether to write to the Log or the status bar.
    if ("toLog".equals(ev.getActionCommand())){
        log.info(txt_writeToLog.getText());
    }
    if ("toStatusBar".equals(ev.getActionCommand())){
        helloWorld.getGSP().writeStatus(txt_writeToStatusBar.getText());
    }
}
```

Creating the revised Hello World

This procedure covers creating the Phase Three of the Hello World application from files source provided with the SDK.

Procedure:

1. Copy the following directory:

   pepper-sdk/applications/HelloWorldResources/Phase3/

2. Past the copied directory into:

   pepper-sdk/applications/

   You should now have the following directory:

   pepper-sdk/applications/Phase3

Procedure complete.

Using the revised Hello World

Build Hello World and make the source files available to the framework, as explained in “Building Applications” on page 11-189.

Use the new Java Section to write messages to the Status Bar and the framework log.

For information about viewing the framework log, see “Keeper event log” on page 2-19.

What's next

This completes the Phase Three of the Hello World Tutorial.

In the next phase, a new Java Section is added that provides a Java user interface for creating and editing a world.

See "Hello World 4: Advanced Java" on page 135.

9

Hello World 4: Advanced Java

This chapter is Phase four, the final phase, of the Hello World application development tutorial.

This chapter explains how to develop and register Java framework Actions that are triggered from HTML buttons and Java ToolBarButtons and how to read from and write to Page XML files from Java. It also shows how to derive text displayed in Java from a properties file in order to support region and language localization.

Overview

This section provides introductory information regarding Phase Four of the Hello World Tutorial.

Prerequisites

- Completion of or familiarity with the previous Hello World application development stages
- A working knowledge of Java
- Familiarity with Java Actions

  See "Java Actions" on page 4-66.

Hello World's new functionality

In this phase, the following are added to Hello World:

- A new Java Section called WorldJavaSection that displays a world Page and allows the user to edit the world's data items

  WorldJavaSection's functionality is equivalent to that of the world Page developed previously in this tutorial. This Section demonstrates how to access Page XML files from Java.

  Figure 9–1 on page 9-136 shows the new WorldJavaSection.
- The WorldJavaSection Section has a framework ToolBar and ToolBarButton
- The WorldJavaSection populates labels in the user interface from a properties file using the framework MessageCatalog class to demonstrate localization of displayed text in Java

For information about localization, see “How to localize for different languages” on page 10-177.

- A new Java Action named EditJavaAction

  EditJavaAction is linked to an HTML button (**Edit with Java**) on the worlds SectionPage. When triggered by the button, EditJavaAction receives the Page ID of the user-selected world, switches focus to the new WorldJavaSection, and populates its widgets from the world’s data items from its Page data.

  Figure 9–2 on page 9-137 shows the revised worlds SectionPage with the new **Edit with Java** button.

- A new Java Action named DoneJavaAction

  DoneJavaAction is linked to a **Done** ToolBarButton in the new WorldJavaSection. When clicked, it saves the current world values from the WorldJavaSection’s user interface to the world’s Page XML file and then returns the focus to the worlds SectionPage.

  Note: The user can click directly on the new WorldJavaSection Tab at any time. However, when focus switches to the WorldJavaSection Section in this way, the WorldJavaSection is not loaded with the data for a particular world but is empty. The application could be developed further to hide the WorldJavaSection and its Tab unless the user clicks the new **Edit with Java** button on the worlds SectionPage, and then to hide the WorldJavaSection when the user clicks the WorldJavaSection’s “Done” button.

Figure 9–1 WorldJavaSection

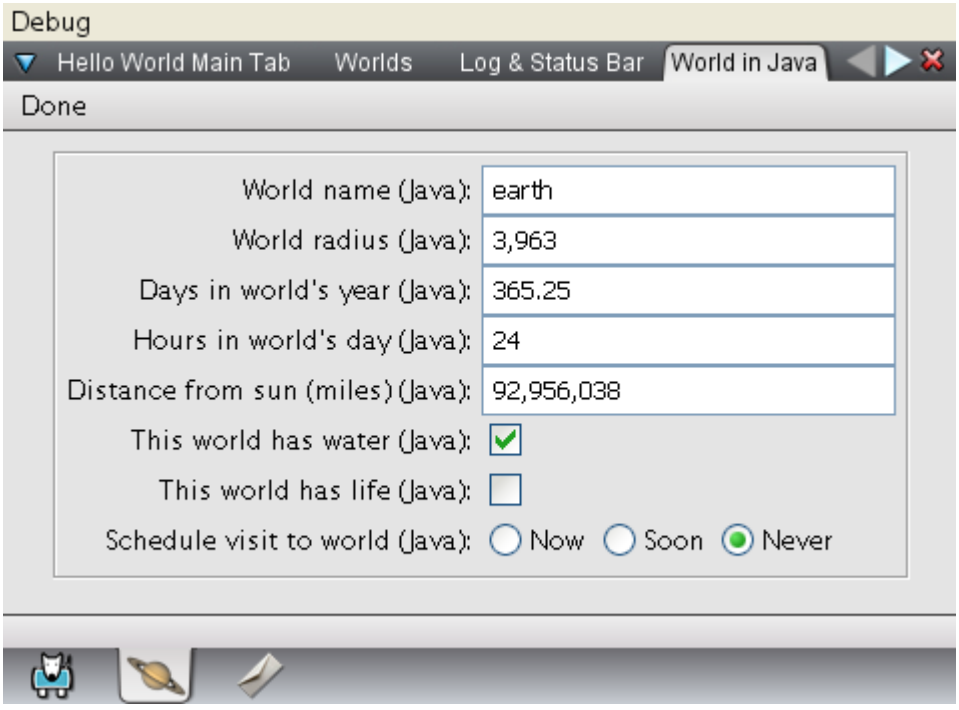
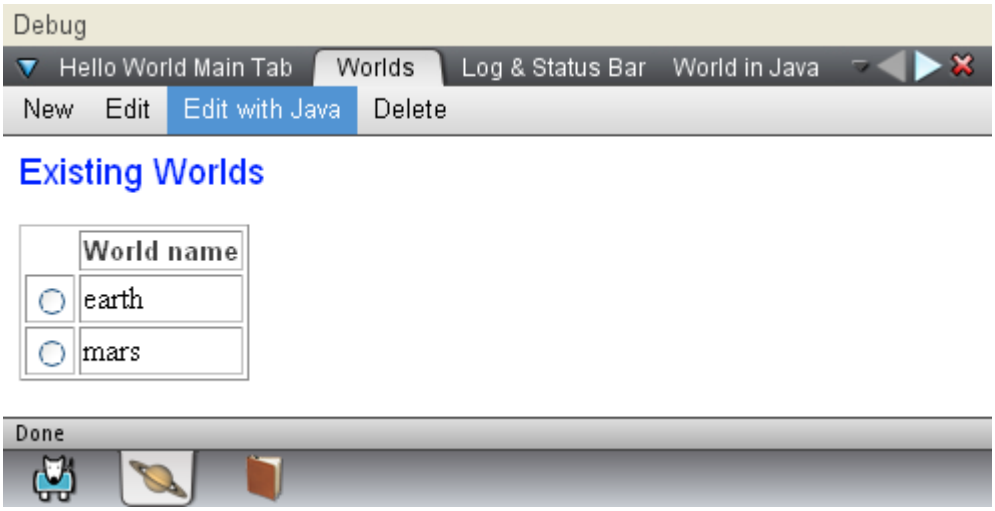


Figure 9–2 New Edit with Java button on worlds SectionPage



Source files

The revised source files for this phase of the Hello World project are here:

pepper-sdk/applications/HelloWorldResources/Phase4/

After the new functionality's conceptual basis is explained through code examples and discussion in the following sections, there's a procedure in which you copy this directory's source files to a working SDK directory to enable building and running the revised application.

Understanding the code

This section explains the new functionality through code snippets and discussion.

> **Note:** It may be helpful to examine the complete source files as you read this discussion. See "Source files" on page 9-137.

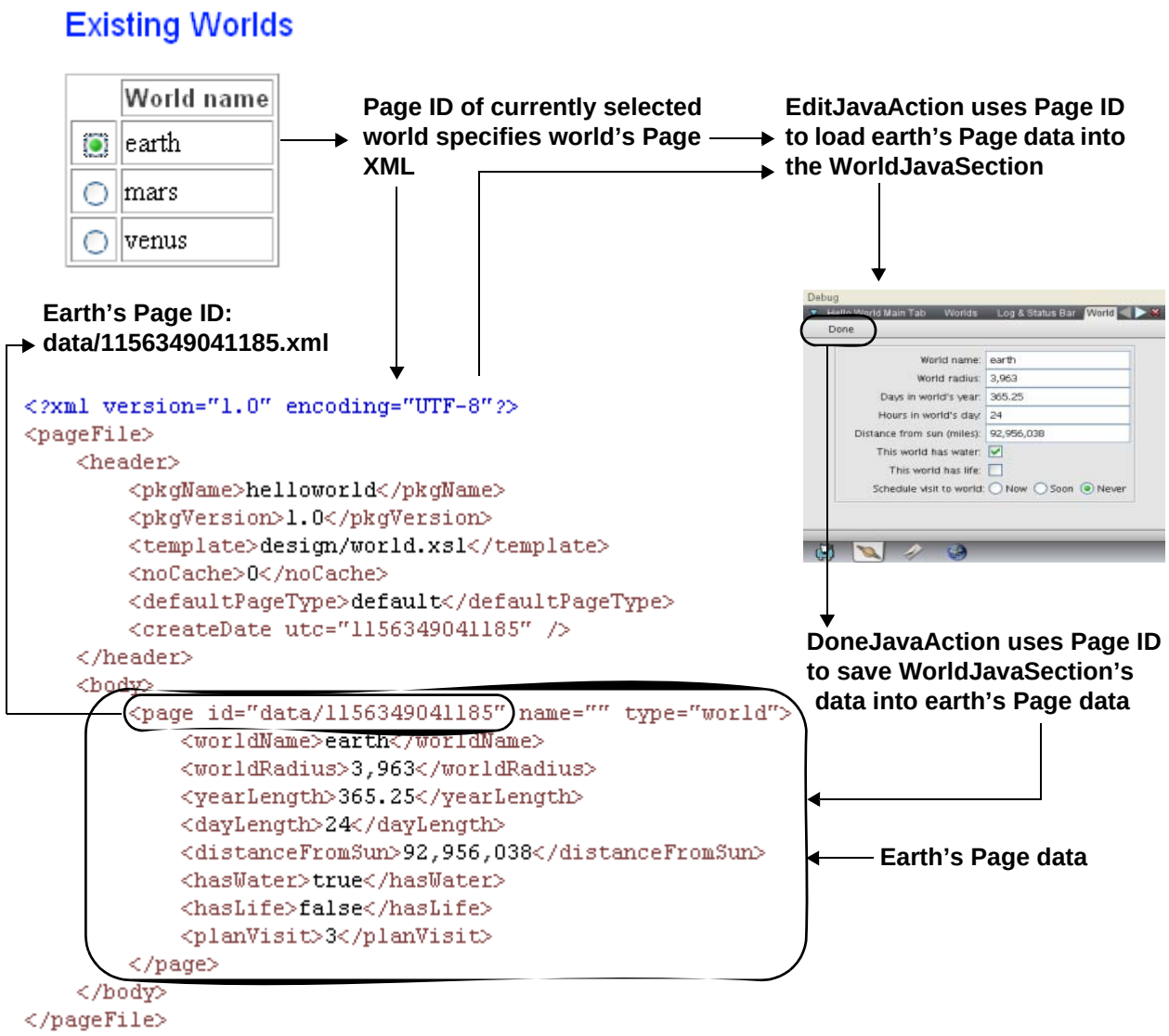
Using the world's Page ID

In this phase, we use the Page ID of the selected world XML Page file for various purposes. It's a bit confusing so here's a summary followed by a graphic that should help to clarify things:

- The Page ID of a world XML Page file is retrieved from the currently selected radio button on the worlds SectionPage and handed to the EditJavaAction.
- The EditJavaAction uses the Page ID to open access world's Page XML data and load it into the WorldJavaSection for display and user-editing.
- The DoneJavaAction uses the Page ID to save data (that the user may have edited) from the WorldJavaSection into the correct world Page XML data.

This is shown graphically in the following figure.

Figure 9–3 Using the Page ID



Application declaration and definition files and pre-built directory

This section explains the modifications required to Hello World's application declaration and definition files and the new pre-built directory:

- *FactoryBuild.xml*
- *PackageStrings.properties*
- *PageTemplates.xml*
- *prebuilt/WorldJavaSection*

FactoryBuild.xml

The new WorldJavaSection is declared in *FactoryBuild.xml* as follows:

Example 9–1 Declaring the new WorldJavaSection in FactoryBuild.xml

```
<!--Definition of WorldJavaSection Section -->
<section name="NameKey.WorldJavaSection" type="java"
                id="data/WorldJavaSection" deletable="true"
                src="../prebuilt/WorldJavaSection" />
```

As discussed previously:

- The type of all Java Sections is `type="java".`
- The `id` attribute value (`id="data/WorldJavaSection"`) identifies the run-time location and name of the Section's SectionPage file (without a file extension).

  As with all Java Sections, it is pre-built and resides in the SDK in the location specified by the `src` attribute.

  Note: The WorldJavaSection Section ID is used programmatically for the first time in this phase of the tutorial. It is used in EditJavaAction to get a handle to the WorldJavaSection Page file (*data/WorldJavaSection.xml*) in order to switch focus to it.

PackageStrings.properties

Two key-value pairs are added to *PackageStrings.properties*:

- One provides the text for the new **Edit with Java** HTML button on the worlds SectionPage.
- The other provides the text displayed on the WorldJavaSection Tab.

The following snippet shows the new key-value pairs.

Example 9–2 Defining two new key-value pairs

```
NameKey.WorldJavaSection=World in Java
Label.EditJava= Edit with Java
```

PageTemplates.xml

As covered in Phase Three of the Hello World Tutorial, an application with one or more Java Sections must have a *PageTemplates.xml* file that includes the ready-made *SectionJava.xml* file. This is done by including the ready-made *SectionJava.xml* file, as shown in the following snippet.

Example 9–3 Including SectionJava.xml in PageTemplates.xml

```
<xi:include href="../resources/pages/SectionJava.xml" />
```

The pre-built directory and pre-built Page

As explained in Phase Three of the Hello World Tutorial, each Java Section requires its own directory inside the *(application)/prebuilt* directory. This directory is specified to in the

FactoryBuild.xml `<section>` declaration. Therefore, there's a new directory: *Phase4/prebuilt/WorldJavaSection*.

This new directory contains the pre-built SectionPage XML file for the new WorldJavaSection, as shown in the following example. Note that the `<section>` and `<java>` elements have their attributes completed as appropriate for this new Java Section.

See "The pre-built directory and pre-built Page" on page 8-126 for information about setting these attributes.

Example 9–4 The WorldJavaSection.xml pre-built SectionPage XML instance file

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile deletable="false">
   <header>
      <packageName>HelloWorldTutorial4</packageName>
      <packageVersion>1.0</packageVersion>
      <template></template>
      <noCache>0</noCache>
      <defaultPageType>default</defaultPageType>
      <createDate utc="1121110270523" />
   </header>
   <body>
      <section name="WorldJavaSection" type="java" id="data/WorldJavaSection">
         <java classname="com.pepper.HW.WorldJavaSection" />
      </section>
   </body>
</pageFile>
```

Adding "Edit with Java" HTML button to worlds SectionPage

The worlds SectionPage developed previously has a toolbar that is given a new button (**Edit with Java**) to triggers the new EditJavaAction. As with the **Edit** button developed in Phase Two of this tutorial, the **Edit with Java** button passes the Page ID of the world Page that is associated with the currently selected radio button, as described above. A new JavaScript function (`editJava()`) is used to extract the Page ID from the selected radio button, call the new EditJavaAction, and pass it the Page ID as a parameter. The `editJava()` function is added to the *helloWorld.js* file, which is included in the worlds SectionPage generated HTML (as it was in previous phases).

The following snippet shows the modified portion of *worlds.xsl* that generates the toolbar with the new button. The button text is set with a new key-value pair (`Label.EditJava`). And, the button is connected to the `editJava()` function.

Example 9–5 Adding the "Edit with Java" button to the worlds SectionPage toolbar

```
<!--create the page toolbar -->
<pepper:pagebar class="PageBar" id="pb-{$packageId}" package="{$packageId}">
   <!-- create three toolbar buttons -->
   <pepper:pagebarentry key="PageBar.New" action="NewPage" param="world" />
   <pepper:pagebarentry key="PageBar.Edit" script="showSelectedPage()" />
   <pepper:pagebarentry key="Label.EditJava" script="editJava()" />
   <pepper:pagebarentry key="PageBar.Delete" script="deleteSelectedWorld()" />
</pepper:pagebar>
```

The following snippet shows the new JavaScript `editJava()` function that is added to *helloWorld.js*. This function retrieves the selected world's Page ID and calls the EditJavaAction with its registered name `EditJavaAction` (as discussed below).

Example 9–6 New editJava() JavaScript function that calls EditJavaAction

```
function editJava(){
    var numWorlds = document.worlds.worldPicker.length;
    var xmlId;
    if (!numWorlds){
        xmlId=document.worlds.worldPicker.value;
    }else{
        for (i=0; i < numWorlds; i++) {
            if (document.worlds.worldPicker[i].checked==true){
                theOne=i;
                break;
            }
        }
        xmlId=document.worlds.worldPicker[i].value;
    }
    bridge.action('EditJavaAction', xmlId);
}
```

New Java code

The base class, HelloWorld, is modified.

In addition, three new classes are added, each with its own source file.

- WorldJavaSection — for the new Java Section
- EditJavaAction — for the new EditJavaAction
- DoneJavaAction — for the new DoneJavaAction

Discussions of the important points for each are provided next.

HelloWorld.java

Note: The source file is included in the SDK in *pepper-sdk/applications/HelloWorldResources/Phase4*

Two important modifications are made to *HelloWorld.java*, as explained next.

New Actions

The two new Actions (EditJavaAction and DoneJavaAction) are instantiated and registered using Action names defined for them as String constants in the HelloWorld base class.

For information about registering Java Actions, see "Registering a Java action in the application base class" on page 4-67.

Action registration in the application's base class must occur in the init() method using the AbstractPepperProgram.registerAction(String, AbstractAction) method.

The String is registered with the framework and provides the *name* of the Action that is used when calling the Action from HTML buttons or from JavaScript. The AbstractAction passed during registration is a new instance of the Action class.

In accordance with good programming practice, the Strings for each Action are defined first as static final class members, as shown in the following.

Example 9–7

```
/**
* <p>Constant used to register EditJavaAction.</p>
*/
public static final String EDIT_JAVA = "EditJavaAction";

/**
* <p>Constant used to register DoneJavaAction.</p>
*/

public static final String DONE_JAVA = "DoneJavaAction";
```

Then, in the init() method, the two Actions are registered, as shown in the following.

Example 9–8

```
public void init(PepperProgramConfig config) throws PepperProgramException {
   super.init(config);
   //Register application's Actions with framework
   registerAction(EDIT_JAVA, new EditJavaAction());
   log.info("completed EditJavaAction action registration");
   registerAction(DONE_JAVA, new DoneJavaAction());
   log.info("completed DoneJavaAction action registration");
}
```

WorldJavaSection registers with HelloWorld

The DoneJavaAction class needs to access WorldJavaSection's user interface widgets in order to save user-entered data from them into the particular world's Page XML data.

To accomplish this, the following steps are taken:

- A WorldJavaSection member variable (`worldJavaSection`) is added to the HelloWorld base class.
- A method is created in HelloWorld (`HelloWorld.setJavaSection(WorldJavaSection)`) that allows the WorldJavaSection class, during its initialization, to pass itself to HelloWorld.

  In the `HelloWorld.setJavaSection(WorldJavaSection)` method, HelloWorld sets the passed WorldJavaSection to equal its local variable (`worldJavaSection`). This gives HelloWorld, and all classes that have access to HelloWorld, a handle to the WorldJavaSection.
- Since DoneJavaAction is a registered Action class, it has access to the HelloWorld base class.

  DoneJavaAction therefore can access WorldJavaSection's data input widgets through the base class's reference (`worldJavaSection`) to WorldJavaSection.

First, in HelloWorld, the `worldJavaSection` member is declared, as follows:

Example 9–9

```
public WorldJavaSection worldJavaSection;
```

Then, the HelloWorld method is added that receives the passed WorldJavaSection and sets it to equal the `HelloWorld.worldJavaSection` member, as follows:

Example 9–10

```
protected void setJavaSection(WorldJavaSection s) {
   this.worldJavaSection = s;
}
```

With these steps in place, HelloWorld is complete. Let's now take a look at the new WorldJavaSection class.

WorldJavaSection.java

Note: The source file is included in the SDK in *pepper-sdk/applications/HelloWorldResources/Phase4*

Most of the code in this file is associated with creating the layout and user interface widgets through which the particular world's data items are displayed and edited. We do not discuss this typical Java code here.

Other points of interest are described in the following sections.

Registering with HelloWorld

In its initComponent() method, which is required because the class implements JavaSectionComponent to hook into the framework, WorldJavaSection registers itself with the HelloWorld base class using HelloWorld's new setJavaSection(WorldJavaSection) method, as described previously. Then, the initWorldJavaSection() method is called to perform Section user interface initialization, as shown in Example 9–11.

Note: It is good programming practice to initialize the Section from the `initComponent()` method, not from the class's constructor. This ensures such initialization occurs after the Section class is integrated with the framework.

Example 9–11

```
public void initComponent(AbstractPepperProgram theProg, Properties params) {
   //give HelloWorld (and therefore my Action classes) a handle to this class
   helloWorld = (HelloWorld) theProg;
   helloWorld.setJavaSection(this);
   initWorldJavaSection();
}
```

Using localizable label text

User interface label text in is derived from the application's *PackageStrings.properties* file. This approach enables Java support for multiple different languages (*localization*) in the same application build.

> For additional information about localization, see "How to localize for different languages" on page 10-177.

As discussed in previous phases of this tutorial, the application's *PackageStrings.properties* file consists of any number of key-value pairs. You can add key-value pairs and then programmatically obtain a value by referencing the key.

In the case of Java, this is done with the following framework class:

> com.pepper.platform.i18n.MessageCatalog
>
> Check the javadoc for detailed information.

Each application has a MessageCatalog object that contains all key-value pairs contained in the application's *PackageStrings.properties* file and in the framework *CommonStrings.properties* file.

You can return a handle to the application's MessageCatalog object using the AbstractPepperProgram.getMessageCatalog() method. Since WorldJavaSection has a handle (`helloWorld`) to the base AbstractPepperProgram object, this can be used to create a handle (named `catalog` in this case) to the application's MessageCatalog object.

Then, the MessageCatalog.getString(String *key*) method is used to return the value associated with the specific key. This value is used to set label text in the user interface.

The first step is creating the key-value pairs in *PackageStrings.properties*. The following key-value pairs are added.

Example 9–12

```
JavaLabel.Name=World name (Java):
JavaLabel.Radius=World radius (Java):
JavaLabel.YearLength=Days in world's year (Java):
JavaLabel.DayLength=Hours in world's day (Java):
JavaLabel.DistanceFromSun=Distance from sun (miles) (Java):
JavaLabel.HasWater=This world has water (Java):
JavaLabel.HasLife=This world has life (Java):
JavaLabel.PlanVisit=Schedule visit to world (Java):
```

The key is the text on a line to the left of the equals sign ("="). the corresponding value is the text to the right of the equals sign. For example:

- `JavaLabel.Name` is a key.
- `World name (Java):` is the corrsponding value.

Then, in the WorldPanel constructor inside WorldJavaSection, the user interface is set up, including setting labels. Only the first label object (`worldName`) is shown in the following example. The

example shows how the label is set from the MessageCatalog (and therefore ultimately from the properties file) using the approach described above.

Example 9–13

```
public WorldPanel() {
   //world name. This label derived from PackageStrings.properties
   //through the MessageCatalog, an approach that supports localization
   MessageCatalog catalog = helloWorld.getMessageCatalog();
   worldName = new JLabel(catalog.getString("JavaLabel.Name")+": ",
      JLabel.RIGHT);
   worldNameTextField = new JTextField();

...
```

Adding the ToolBar and ToolBarButton

The ToolBar object is declared and created as a class member, as follows:

Example 9–14

```
/** ToolBar for the Section*/
protected ToolBar toolbar = new ToolBar();
```

Then, in the initWorldJavaSection() method, the ToolBar is added to the BorderLayout.NORTH region of WorldJavaSection (which is a JPanel set to use BorderLayout), the ToolBarButton is created with DoneJavaAction and with “Done” display text, and the ToolBarButton is added to the ToolBar, as shown in the following.

Example 9–15

```
public void initWorldJavaSection() {
   this.setLayout(new BorderLayout());
   this.add(toolbar, BorderLayout.NORTH);
   this.add(panel_content, BorderLayout.CENTER);
   ToolBarButton b1 = new ToolBarButton(new DoneJavaAction(), "Done");
   toolbar.addButton(b1);
...
```

Page ID helper methods

As we have seen, DoneJavaAction needs to know the world Page the WorldJavaSection is displaying so that it can save WorldJavaSection’s data into the correct Page XML.

A high-level view of the process through which the Page ID is retrieved from the worlds SectionPage, passed through EditJavaAction to the WorldJavaSection, and then passed to DoneJavaAction was explained previously.

See “Using the world’s Page ID” on page 9-137.

WorldJavaSection contains two methods that enable passage of the Page ID.

- WorldJavaSection.setPageId(String)is used by the EditJavaAction class to inform WorldJavaSection of the Page ID of the world Page it is displaying.

This is necessary because EditJavaAction, which is associated with an HTML button on the worlds SectionPage, is passed the Page ID of the particular world the user has selected and wants to edit (the *target* world). EditJavaAction uses the Page ID to load the correct world's data from its XML instance file into the WorldJavaSection.

- WorldJavaSection.getPageId() is used by DoneJavaAction to retrieve the Page ID from WorldJavaSection so that it can save the data to the correct XML instance file.

The following shows both methods:

Example 9–16 Interface methods to get and set the Page ID

```
protected void setPageId(String pageId) {
    this.s_pageId = pageId;
}
protected String getPageId() {
    return s_pageId;
}
```

EditJavaAction.java

Note: The source file is included in the SDK in *pepper-sdk/applications/HelloWorldResources/Phase4*

Note: The EditJavaAction class follows the rules for Action classes explained in "Retrieving passed parameters" on page 4-66.

EditJavaAction performs several key operations:

- Switches focus to the WorldJavaSection
- Retrieves the Page ID of the target world Page from the parameter passed to it from the worlds SectionPage **Edit with Java** button
- Informs the WorldJavaSection object of the Page ID of the target world to be displayed using the WorldJavaSection.setPageId(PageId) method
- Makes a handle to a particular world Page named `worldPage` using the Page ID of the target world and the ID of the worlds SectionPage (the parent SectionPage of the target world Page)
- Loads the target `worldPage` Page
- Creates a JDOM Document object (`worldData`) from the worldPage

  This provides access to the target world Page data elements.
- Loads `worldData`'s data elements into WorldJavaSection's widgets

These operations are executed each time a user clicks the **Edit with Java** button. To accomplish this, the code for these operations is placed in the `public void actionPerformed(ActionEvent event)` method.

These operations are explained in the following sections.

Switching focus to the WorldJavaSection

EditJavaAction switches the focus to the WorldJavaSection.

This is done using the handle to the base class (`helloworld`), which is an instance of an AbstractPepperProgram. It uses the handle to access two AbstractPepperProgram methods:

- AbstractPepperProgram.getSection(sectionID)

  This returns the section with the specified ID.

- AbstractPepperProgram.showSection(Section)

  This displays the passed Section.

Example 9–17 Displaying the WorldJavaSection Section from the EditJavaAction

```
//try to display the WorldJavaSection:
try{
   //get a handle to the WorldJavaSection
   worldJavaSection = helloWorld.getSection(WORLD_JAVA_SECTION);

   //switch focus to the WorldJavaSection
   helloWorld.showSection(worldJavaSection);
}catch (Exception e) {
   log.error("Could not display Java Section from EditJavaAction Action");
}
```

Retrieving the passed world Page ID

As explained previously, EditJavaAction is passed the Page ID of the currently selected world from the editJava() JavaScript function that is associated with the **Edit with Java** button on the worlds SectionPage, as shown in Example 9–18.

For more information, see "Retrieving passed parameters" on page 4-66.

Example 9–18 Retrieving the world Page ID

```
//get the pageId of the world page to display from the parameter passed to this Action
this.pageId = ((ActionEventWithParams) event).getParam();
```

Setting the target world Page in WorldJavaSection

EditJavaAction informs the WorldJavaSection of the Page ID of the target world Page using the `WorldJavaSection.setPageId(PageId)` method, as follows:

Example 9–19 Setting the target world Page ID in WorldJavaSection

```
//set JavaSection's pageId so it knows the id of the page it is to display
helloWorld.worldJavaSection.setPageId(pageId);
```

Making a document for the target World Page

EditJavaAction makes a handle (named `woldPage`) to the target world, loads the world XML file into the Page object, and then makes a JDOM Document object (`worldData`) from the Page data to enable easy access to the XML elements, as follows:

Example 9–20 Setting the target world Page ID in WorldJavaSection

```
//Make a Page object for the world Page to be edited
//method's first param is the Page ID to get, handed here from worlds SectionPage
//method's second param is the id of the section (the worlds SectionPage) the page is in
Page worldPage = getProgram().getPage(pageId, "data/Worlds");
// load the page from disk into memory if it is not yet loaded
if (!worldPage.isLoaded()) {
   try{
      worldPage.load();
   }catch (Exception e){
      log.error("World's XML instance file failed to load into memory");
   }
}
//create a Document object for the target world base page
Document worldData = worldPage.getPageData();
```

Reading data into WorldJavaSection

All that remains for EditJavaAction is to:

- Find the world `<page>` element in the JDOM Document (`worldData`) that represents the Page Data,

  This is returned by the framework `worldPage.getRoot()` method.
- Read each world data element value into a JDOM Element object, and
- Write the value from each Element object into the appropriate WorldJavaSection user interface widget.

  Tip: The XML data is accessed in a `synchronized` block to prevent collisions from potential simultaneous operations on the same file.

Example 9–21 Reading data from the target world Page data and writing to Java

```
synchronized (worldData) {
   Element root = worldData.getRootElement();
   if (root != null) {
      Element worldNameElement = root.getChild("worldName");
      helloWorld.worldJavaSection.worldPanel.worldNameTextField.
         setText(worldNameElement.getText());
   ...
   }
}
```

DoneJavaAction.java

Note: The source file is included in the SDK in *pepper-sdk/applications/HelloWorldResources/Phase4*

DoneJavaAction is very similar to EditJavaAction, with the following exceptions:

- Instead of setting the focus to WorldJavaSection as EditJavaAction does, DoneJavaAction switches focus to the worlds SectionPage using the `WORLDS_XML` constant, which is defined as equal to “data/Worlds”, the worlds SectionPage Section ID.
- To identify the particular world Page that had just been edited by WorldJavaSection, DoneJavaAction retrieves its Page ID using WorldJavaSection’s getPageId() method, described previously.
- Instead of reading data from the world Page data and writing it to the WorldJavaSection as EditJavaAction does, DoneJavaAction reads data from WorldJavaSection widgets and writes it to the target world Page using HelloWorld’s registered reference to WorldJavaSection, as described previously and shown in the following example.

Example 9–22 Writing WorldJavaSection’s data to the world Page

```
//load worldPage XML data into JDOM Document
Document worldData = worldPage.getPageData();

synchronized (worldData) {
   //Load XML element data into Java widgets.
   //Note that the root element that is loaded is not the actual XML file's
   //root but is rather the <page> element.
   Element root = worldData.getRootElement();

   //get the current value of each UI widget and save it in an appropriate
   //format into the corresponding element in the JDOM Document
   if (root != null) {
      Element worldNameElement = root.getChild("worldName");
      worldNameElement.setText(helloWorld.worldJavaSection.
         worldPanel.worldNameTextField.getText());
      ...
```

Creating the revised Hello World

This procedure covers creating the Phase Four of the Hello World Tutorial from files provided with the SDK.

Procedure:

1. Copy the following directory:

   pepper-sdk/applications/HelloWorldResources/Phase4/

2. Past the copied directory into:

   pepper-sdk/applications/

   You should now have the following directory:

   pepper-sdk/applications/Phase4

Procedure complete.

Using the revised Hello World

Rebuild Hello World as explained in Building Applications and launch it.

What’s next

This completes the Hello World Tutorial application.

10

Customization

Customization is a broad topic that refers in general to steps you can take to modify the visual design of applications and even of the framework itself. It includes modifying CSS stylesheets to affect the appearance of generated HTML pages, key framework user interface components such as the Status Bar and of Java Sections and components. It includes replacing images used as backgrounds in important Sections (such as the **Applications** Tab) and used in user interface components like toolbars on Pages. It includes building custom themes to package and distribute a bundled set of customizations. It even includes redesigning the structure and contents of Sections and Pages.

Getting started with customization

This section provides information about the scope of customization options and different approaches to customization.

What can be customized

You can customize almost everything except for the compiled logic of applications you did not develop (such as the Keeper).

> **Note:** Similar functionality can be implemented in different ways among different applications. Functionality that is customizable in one application may not be in others. For example, if an application implements a toolbar using XSL, it is customizable, whereas if another application implements a Java ToolBar that appears identical, it may not be.

Examples of customization options include:

- You can customize visual styling, fonts, colors, and layout of XML/XSL/HTML Pages by modifying *styles.css* CSS stylesheets and various image files.
- You can customize colors and fonts of Java aspects of the framework (such as the Status Bar, the System Tray, the Flag Panel, and Pepper Java ToolBars) and in Java Sections by modifying *keeper.css.*
- You can customize the content of XML/XSL/HTML Pages by modifying XSL files used to generate HTML pages.

- You can customize XML/XSL/HTML Page toolbars (for example adding new buttons or deleting buttons) by customizing the XSL files used to generate Pages.
- You can create a custom set of default bookmarks for any Web Section and rebuild the application for distribution.
- You can rewrite the Help section of any application, and create multiple versions to support different languages (called *localization)*, and rebuild the application for distribution.
- You can customize an application's Section and Page structure by customizing its definition files: *FactoryBuild.xml* and *PageTemplates.xml.*
- You can customize text displayed in the user interface (for example to switch languages or simply to modify displayed terms) by modifying properties files.
- You can bundle a set of customizations together into a custom *theme* and launch the framework to use the theme, transparently to the user.

The Keeper is an application

Before drilling into details about what can be customized and how to go about customizing, it's important to bear in mind that the Keeper is an application.

The Keeper is an application (almost) like any other. It just happens to have a Section that displays icons for all other applications, is run by default at start-up and has a privileged position in the framework. However, its Sections and Pages, and how they are displayed, are controlled by application declaration and definition files that are instantiated as XML files, just like other applications. The XML instance files are transformed by XSL into HTML, and the HTML is rendered for display according to CSS stylesheets, just like other applications.

The Keeper has its own *design.zip* file containing a *FactoryBuild.xml* file and a *PageTemplates.xml* file (and the other key files), just like every other application.

So, you can customize the Keeper in almost exactly the same way you customize any other application. You port it to the SDK and customize it.

> **Note:** Updating a customized Keeper from the SDK is different than for other applications. See "Updating framework zip files" on page 2-13.

Customizing in Design Mode or in the SDK

Customization can be implemented for running applications using Design Mode. Or, an application can be ported into the SDK, customized, and rebuilt for distribution.

> For information about Design Mode, see "Design Mode" on page 2-13.

A combination of the two approaches is often useful: Customize a run-time application using Design Mode, and, when satisfied, port the application to the SDK with its customizations, then rebuild it for distribution.

Design Mode customization

Customizations in Design Mode apply to the current installed instance of the application (including the Keeper application) only. This is because in Design Mode, you make changes to the installed

application, not to the application in the SDK. Any changes are therefore not reflected automatically in future application builds.

Also, you can't modify program logic or some user interface characteristics derived from Java in Design Mode, because modifying Java source code requires access to the source code and rebuilding the application in the SDK.

> **Note:** You can customize colors and fonts throughout many parts of the framework user interface that derive from Java code and within applications that use a Java user interface by modifying a single CSS stylesheet: *keeper.css*.

In addition, Design Mode customizations only apply when the framework is in Design Mode, even if you created the customizations in Design Mode. This is because when the framework is in Design Mode, it extracts normally archived design files and uses them instead of the archived files. Modifications to design files affect only the extracted files, not the archives. When the framework is relaunched not in Design Mode, it uses the archived design files, even if the extracted files exist.

Customization in Design Mode is helpful nonetheless. The effects of modifications are generally visible immediately. One convenient approach is to customize the application in Design Mode and later to copy the files into the application's directory in the SDK or into a custom theme in the SDK in order to later integrate the changes into the build process.

Customizations made in Design Mode only affect future events in the application. For example, if you modify a Page definition, future Page instances will be based on the modified structure, but existing Page instances reflect the previous structure.

> **Note:** You can port most applications (even those you did not develop) to the SDK and create fresh builds there, thus creating a version of the application that is based entirely on modified files. See "How to port an application into the SDK" on page 10-181.

Design Mode is a convenient way to develop and test non-Java aspects of an application because changes take effect immediately in the running application without any need to rebuild the application.

SDK-based customization

If you want to create a new build of an application (or theme) that includes all customizations, use the SDK to rebuild the application (or theme) using customized files.

For your applications, this is straightforward because the application already exists in the SDK directory hierarchy.

> **Note:** If you customized your applications using Design Mode, you must copy the modified files back into the SDK or the modifications are not carried into future builds. See "How to port an application into the SDK" on page 10-181.

For applications you did not develop, you can still port the application into the SDK. Then you can rebuild it to create a version that is ready for installation that includes your customizations.

For themes, you can simply copy the modified files into your SDK theme.

For information about themes, see "Custom themes" on page 10-167

How customizations are affected by automatic updates

The Pepper Application Framework and its bundled applications are automatically updated from web-based update servers when appropriate, although a user does have the option to decline the update.

Such updates may replace important application files including, *design.zip, data.zip,* jar files and the default theme archive (*common-resources.zip*). Therefore, customizations involving the framework, bundled applications, and the default theme can be lost as a result of updates.

> **Note:** Automatic updates do not affect applications that are not a part of an update server scheme.

Customers interested in deploying an update server to control this process more closely should contact Pepper Computer, Inc.

Customization with CSS

Perhaps the easiest way to make sweeping changes in the visual styling of the framework and applications is by modifying two key CSS stylesheets, both of which are found in the theme archive.

> For information about theme archives, see “Custom themes” on page 10-167.

The two CSS stylesheets are:

- *(theme-resources.zip)/resources/styles/styles.css*

  This is the default stylesheet used for HTML pages in the framework and most other applications.

  Using the classes contained in this stylesheet in your XSL-generated HTML pages helps to create a uniform visual style throughout all applications. This approach also creates a single point of control of the visual styling of all HTML pages and thus simplifies the process of making style modifications.

  For information and procedures for common *styles.css*-based customizations, see “Styles.css-based customizations” on page 10-155.

- *(theme-resources.zip)/resources/styles/keeper.css*

  This stylesheet contains CSS classes that determine the colors and fonts used in Java user interface widgets and components throughout the framework and in applications. For example, ToolBars and ToolBarButtons, the Status Bar (and its Progress Bar), the Flag Panel, Java combo boxes, Java check boxes, Java text labels and more derive their colors and fonts from these CSS classes.

  You can create your own Java widgets and assign them the colors and fonts defined by *keeper.css* classes.

  > **Note:** You cannot create new classes in *keeper.css*. You must use classes that already exist.

  For information and procedures for common *keeper.css*-based customizations, see “Keeper.css-based customizations” on page 10-159.

Tip: Both Default Sections and Java Sections have similar presentation widgets, such as toolbars, check boxes, and so on. To ensure the visual styling for the two Section types is consistent, in the current release it is important to remember to customize both *styles.css* and *keeper.css*.

Other CSS stylesheets

While it is recommended to develop your applications to use the two key framework stylesheets, you can create your own styles sheets and use them instead or in addition to these.

Tip: The downside to this approach is that the visual styling of your applications are not controlled by the two framework stylesheets, making sweeping design changes more difficult.

Therefore, it is important to remember that applications may refer to stylesheets in their own *design.zip* archives. Comprehensive customization may require finding any such exceptional applications and taking the appropriate steps.

Getting started with CSS customization

A good way to get started with CSS-based customization is as follows:

- Launch the framework in Design Mode.

  This extracts the current theme archive and design files for all applications when they launch and causes the framework and applications to use the extracted files instead of the archive files.

  For information about Design Mode, see “Design Mode” on page 2-13.

- Modify the CSS stylesheets and observe the results as you develop your unique visual styling while using the framework or the relevant application.

  Note: Modifying *keeper.css* requires restarting the framework.

- When you are done modifying the CSS stylesheets, create a custom theme that includes the modified stylesheets in the SDK, distribute the custom theme, and configure the framework to launch using the custom theme.

  See “Custom themes” on page 10-167.

Styles.css-based customizations

This section provides information about common customizations you can do by modifying the *styles.css* theme file.

Note: The customizations presented here are just a few of the customizations that are possible and are provided as examples. Developers are encouraged to experiment with customizing *styles.css.*

Customizing the Keeper Applications tab background image

The background image of the Keeper’s **Applications** tab is:

common-resources.zip/resources/styles/background.jpg

How it works

The default background image is applied by assigning the CSS class `Content` to an HTML element. The `Content` class is defined in:

common-resources.zip/resources/styles/styles.css

> **Note:** This is a theme archive file. Customizing such files typically involves creating a custom theme. See "Custom themes" on page 10-167.

In the default *styles.css* stylesheet, `Content` is defined to allow it to be assigned only to `<div>`, `<form>` and `<table>` elements.

The `Content` class sets the `background-image` attribute to a URL that specifies this file, by default to `url(background.jpg)`, which resides in the same directory.

> **Note:** This is a theme archive file. Customizing such files typically involves creating a custom theme. See "Custom themes" on page 10-167.

By default, the **Applications** Tab is generated by the Keeper application's *Main.xsl* transform. An HTML `<div>` element is generated by *Main.xsl* with its class set to `Content`, as follows:

```
<div class="Content" id="packagegrid">
```

The background image is set as a result of the HTML element, its CSS class, and the class's background image.

How to customize it

You can modify the **Applications** Tab's background image in several ways.

- Replace *background.jpg* with the desired image file.

  See "Custom themes" on page 10-167.

- Modify *styles.css*'s `Content` class to use a different background image and place that background image in the same directory.

  See "Custom themes" on page 10-167.

- Modify the XSL transform (*Main.xsl*) that generates the Page to assign the `<div>` element that uses the `Content` class to use a new class you create that is defined with a `background-image` attribute that points to a URL that specifies the background image of your choice, which should reside in the same directory.

  This approach requires porting the Keeper application to the SDK. See "How to port an application into the SDK" on page 10-181.

Customizing the selection block

The *selection block* is the image that displays when an icon is selected on the **Applications** Tab or the **Setting** Tab. You can customize the selection block.

The selection block has two parts:

- The selection block's border

  The thin line forming the square with rounded corners that forms the outer edge of the selection block. The selection block's default border (and its background image) are applied by assigning

the CSS class `FloatingBlock` to an HTML element. The `FloatingBlock` class is defined in:

common-resources.zip/resources/styles/styles.css

Note: This is a theme archive file. Customizing such files typically involves creating a custom theme. See "Custom themes" on page 10-167.

- The selection block's background image

  This is a portable network graphics (png) image file whose dimensions are 1 pixel wide (x axis) by 90 pixels tall (y axis). It is a transparency gradient that is most transparent at the top to least transparent at the bottom.

  The image is:

  common-resources.zip/resources/styles/floatingselection.png

  Note: This is a theme archive file. Customizing such files typically involves creating a custom theme. See "Custom themes" on page 10-167.

  The selection block's background image (and its border) are applied by assigning the CSS class `FloatingBlock` to an HTML element. The `FloatingBlock` class is defined in:

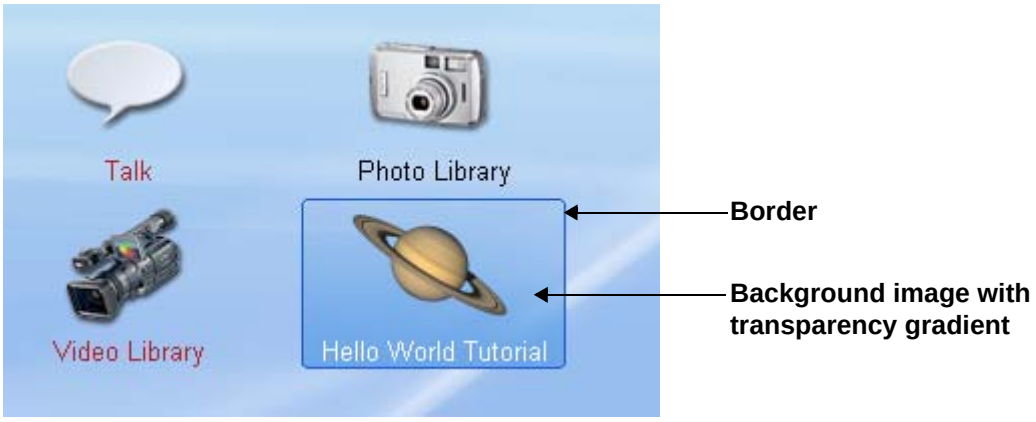
  common-resources.zip/resources/styles/styles.css

  Note: These are theme archive files. Customizing such files typically involves creating a custom theme. See "Custom themes" on page 10-167.

Details of how the `FloatingBlock` class determined the border and background image, and how it varies the visual styling depending on whether the item is selected, are covered below.

Figure 10–1 shows the selection block for the Hello World Tutorial.

Figure 10–1 The default selection block



How it works

The `FloatingBlock` class has different styles applied depending on the values of the `selected` attribute.

- When the HTML element is not selected, the `FloatingBlock` class is applied.

- When the HTML element is selected, the `div[class=FloatingBlock][selected=true]` version of the class is applied.

When the element is selected, the background image and the border's visual styling is set with the attributes in Example 10–1. The attributes in **bold** are the ones you are most likely to modify.

Example 10–1 Section block's CSS class

```
div[class=FloatingBlock][selected=true] {
   background-image: url(floatingselection.png);
   background-repeat: repeat-x;
   background-position: bottom left;
   border: solid 1px #1f5ccf;
   -moz-border-radius: 4px;
   color: white;
}
```

How to customize it

This section covers how to customize the border and the background image of the selection block.

Such customization typically requires developing a custom theme. See "Custom themes" on page 10-167.

Customizing the border

You can customize the selection block's border by modifying the `div[class=FloatingBlock][selected=true]` class in *styles.css*, as follows:

- Set the border's line type, its thickness, and its color with the `border` attribute.
- Set the border's corner radius with the `-moz-border-radius` attribute.

Customizing the background image

You can customize the selection block's background image in the following ways:

- Replace *common-resources.zip/resources/styles/floatingimage.png w*ith an image file appropriate for your customization design.

  If you are using a custom theme, replace the file in your theme's directory in the SDK:

  pepper-sdk/themes/(yourtheme)/resources/styles/floatingimage.png

- Modify the *styles.css* stylesheet to refer to a different image file, and place that file in *common-resources.zip/resources/styles/.*

  Again, if you are using a custom theme, place the file in your theme's directory in the SDK:

  pepper-sdk/themes/(yourtheme)/resources/styles/floatingimage.png

- Modify the XSL transform that generates the HTML page of interest to assign different CSS classes to the HTML element for both its selected and unselected states.

Customizing application icons

Each application has two icons:

- A 64 pixel by 64 pixel icon that displays in the framework's **Applications** tab, and
- A 24 pixel by 24 pixel icon that displays in the framework's System Tray.

The icons are included in the application's *design/images* folder in the SDK. The names and location of the two images are specified in the application's *package.ppld* file.

For information, see "Modifying package.ppld" on page 6-93.

How to customize application icons

There are several options for modifying Application icons.

- If you developed the application, you can simply change the icon files, modify the application's *package.ppld* file if you changed the file names, then rebuild the application.
- If the application is bundled with the framework, you can port the application to the SDK, change the icons, modify the application's *package.pplfd* file as appropriate, then rebuild the application.

  For information on porting an application to the SDK, see "How to port an application into the SDK" on page 10-181.

Keeper.css-based customizations

keeper.css contains CSS classes that define colors and fonts that are used by Java-based user interface entities in the framework and in applications.

keeper.css is loaded when the framework launches and its CSS classes are read into memory.

> **Note:** Because *keeper.css* is only read at framework launch time, you must relaunch the framework after making changes to *keeper.css* in order to see the changes reflected in the user interface.

Many Java aspects of the user interface use the colors and fonts defined by these classes. For example:

- The colors and fonts used to render the Status Bar and its Progress Bar are derived from *keeper.css* classes.

  See "Customizing the Status Bar" on page 10-160.
- The colors used to render System Tray are derived from *keeper.css* classes.

  See "Customizing Flag Panel colors" on page 10-162.
- The colors and fonts used to render Java ToolBars and ToolBarButtons are derived from *keeper.css* classes.

  See "Customizing Pepper ToolBar colors and fonts" on page 10-163.
- If you create Java user interface items, you can also access the colors and fonts defined in *keeper.css* classes and apply them.

Note: Many other classes exist in *keeper.css*. The specifics provided here are meant to help you get started with *keeper.css* customization and are not meant as complete reference documentation.

Customizing the Status Bar

The framework Status Bar can be customized with various *keeper.css* classes as follows:

- The background colors

  See "Customizing Status Bar background colors" on page 10-160

- The border color

  See "Customizing Status Bar bottom border color" on page 10-161

- The Progress Bar color and text color

  The Progress Bar displays inside the Status Bar.

  See "Customizing Status Bar's Progress Bar" on page 10-162

  Note: For help identifying the parts of the framework user interface, see "Anatomy of the user interface" on page 3-23.

Customizing Status Bar background colors

The background coloration of the Status Bar uses two colors that are defined by the following *keeper.css* classes:

- PepperStatusBar.background1

  Defines the color that is opaque at the top of the Status Bar and becomes increasingly transparent until it is completely transparent at the bottom of the Status Bar.

- PepperStatusBar.background2

  Defines the color that is opaque at the bottom of the Status Bar and becomes increasingly transparent until it is completely transparent at the top of the Status Bar.

Example 10–2 shows a sample definition of the two colors in *keeper.css*.

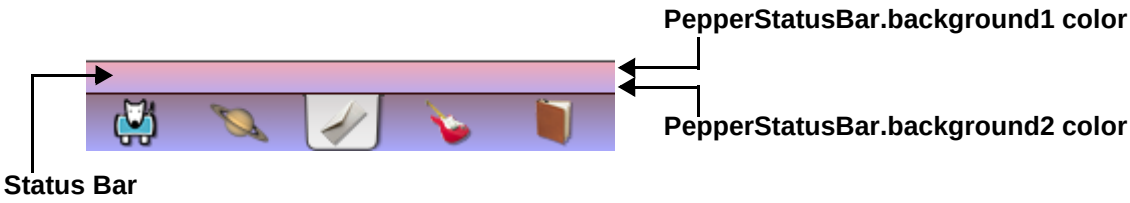
Note: Colors are defined using CSS syntax.

Figure 10–2 shows the Status Bar as it appears with the *keeper.css* class definitions in Example 10–2.

Example 10–2 Setting the Status Bar's background colors

```
PepperStatusBar.background1 {
   color: #eeaabb;
}
PepperStatusBar.background2 {
   color: #bbaaee;
}
```

Figure 10–2 Status Bar background color



Customizing Status Bar bottom border color

The color of the Status Bar's bottom border is determined by the *keeper.css* `PepperStatusBar.borderColor` class, as shown in Example 10–4.

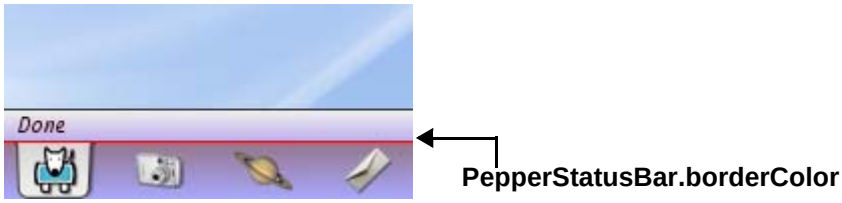
Note: The top border's color is not customizable in the current release.

Example 10–3 Setting the Status Bar's bottom border color

```
PepperStatusBar.borderColor {
   color: red;
}
```

Figure 10–3 shows Status Bar's bottom border with the *keeper.css* class defined as shown in Example 10–3.

Figure 10–3 Status Bar border color



Customizing Status Bar font and text color

The color of Status Bar text (when the progress bar is not displayed) is determined by the *keeper.css* `ProgressBar.selectionBackground` class, as shown in Example 10–4.

Example 10–4 Setting the Status Bar text color

```
ProgressBar.selectionBackground {
   color: #33000;
}
```

The font of Status Bar text (at all times, whether the Progress bar is displayed or not) is determined by the *keeper.css* `PepperStatusBar.font` class, as shown in Example 10–5.

Example 10–5 Setting the Status Bar font

```
PepperStatusBar.font {
   font-family: Lucida Sans Italic;
```

```
    font-style: plain;
    font-size: 10px;
}
```

Figure 10–4 shows Status Bar text with the *keeper.css* classes as defined in Example 10–4 and Example 10–5.

Figure 10–4 Status Bar font and text color

Customizing Status Bar's Progress Bar

The color of the Status Bar's Progress Bar is determined by the *keeper.css* `ProgressBar.foreground` class, as shown in Example 10–6.

Example 10–6 Setting the Status Bar's Progress Bar color

```
ProgressBar.foreground {
    color: #bbbbee;
}
```

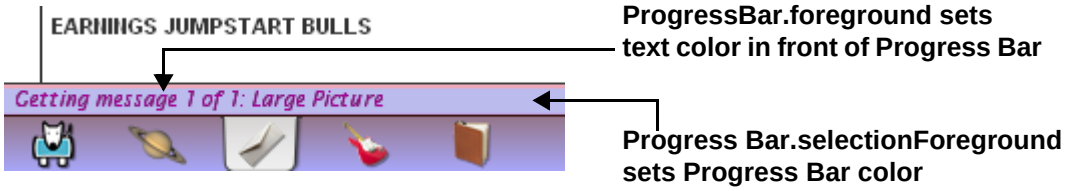
The color of Status Bar text messages displayed when the Progress Bar is visible behind them is determined by the *keeper.css* `ProgressBar.foreground` class, as shown in Example 10–7.

Example 10–7 Setting the Status Bar text color when the Progress Bar is visible

```
ProgressBar.selectionForeground {
    color: #aa0099;
}
```

Figure 10–5 shows the Progress Bar color and the color of Status Bar text in front of the progress bar using *keeper.css* classes defined in Example 10–6 and Example 10–7.

Figure 10–5 Status Bar's Progress Bar colors



Customizing Flag Panel colors

Note: For help identifying the parts of the framework user interface, see "Anatomy of the user interface" on page 3-23.

The background coloration of the Flag Panel uses two colors that are defined by the following *keeper.css* classes:

- PepperFlagPanel.background

  Note: There is no trailing “1” on “background”.

  Defines the color that is opaque at the top of the Flag Panel and becomes increasingly transparent until it is completely transparent at the bottom of the System Tray.

- PepperFlagPanel.background2

  Defines the color that is opaque at the bottom of the Flag Panel and becomes increasingly transparent until it is completely transparent at the top of the System Tray.

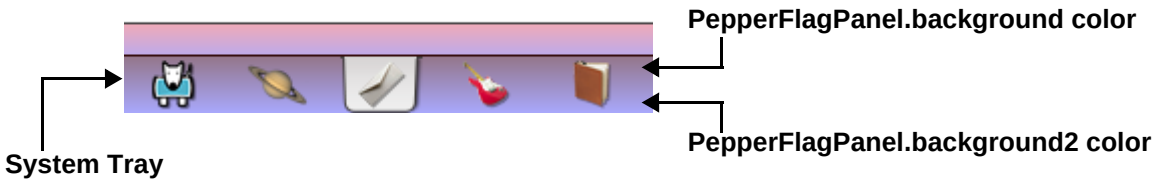
Example 10–8 shows a sample definition of the two colors in *keeper.css*.

Figure 10–6 shows the Flag Panel as it appears with the *keeper.css* class definitions in Example 10–8.

Example 10–8 Setting the Flag Panel background colors

```
PepperFlagPanel.background {
    color: #95788e;
}
PepperFlagPanel.background2 {
    color: #aaaaff;
}
```

Figure 10–6 Flag Panel background colors



Customizing Pepper ToolBar colors and fonts

This section explains how to customize the visual styling of Pepper Java ToolBars. Pepper Java ToolBars are objects that instantiate the framework ToolBar class and whose buttons instantiate the framework ToolBarButton class.

Tip: It is recommended to use the framework ToolBar and ToolBarButton classes instead of JToolBar and JButton in order to enable your Java toolbars to be controlled by the same *keeper.css* classes as all other Java toolbars.

For information about using the framework ToolBar class and ToolBarButton class, see “Java ToolBars” on page 4-52.

Note: This material does not apply to toolbars added to non-Java Sections. Their visual styling is controlled by *styles.css*.

When you add a ToolBar to a Java Section, the framework automatically applies a set of CSS classes in *keeper.css* to it. The result is that all Java ToolBars in all applications have a uniform style that can be modified as a set by modifying these *keeper.css* classes. The classes control the following:

- ToolBar background colors

  See "Customizing ToolBar colors" on page 10-164

- ToolBar bottom border color

  See "Customizing ToolBar bottom border color" on page 10-165

- ToolBarButton colors

  See "Customizing ToolBarButton mouse pressed colors" on page 10-166

- ToolBar font and font color

  See "Customizing ToolBarButton font and font color" on page 10-165

  Note: For help identifying the parts of the framework user interface, see "Anatomy of the user interface" on page 3-23.

Customizing ToolBar colors

The background coloration of Java ToolBars (framework ToolBar objects) depends on two colors that are defined by the following *keeper.css* classes:

- ToolBar.background1

  Defines the color that is opaque at the top of ToolBars and becomes increasingly transparent until it is completely transparent at the bottom of ToolBars.

- ToolBar.background2

  Defines the color that is opaque at the bottom of ToolBars and becomes increasingly transparent until it is completely transparent at the top of the ToolBars.

Example 10–9 shows a sample definition of the two colors in *keeper.css*.

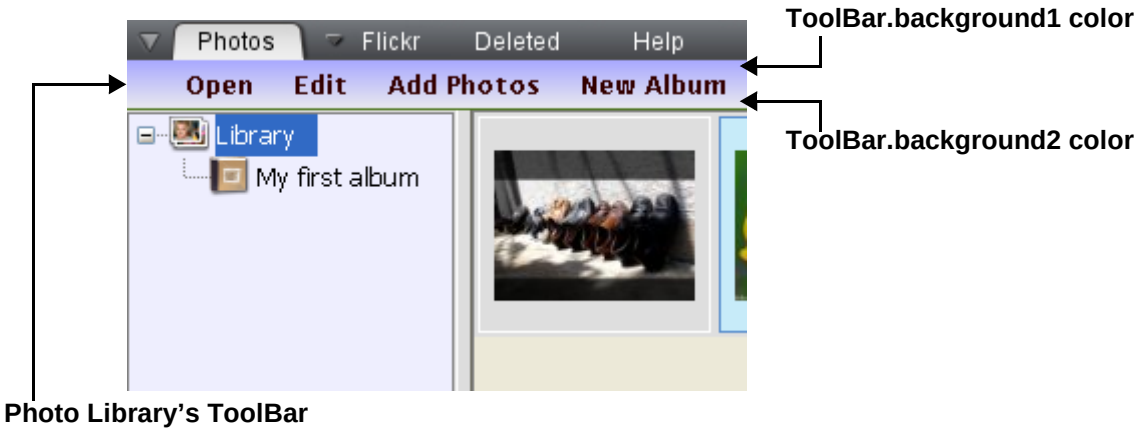
Note: Colors are defined using CSS syntax.

Figure 10–7 shows the Photo Library's ToolBar as it appears with the *keeper.css* class definitions in Example 10–9.

Example 10–9 Setting ToolBar background colors

```
ToolBar.background1 {
        color: #aaaaff;
}
ToolBar.background2 {
        color: #eeeeee;
}
```

Figure 10–7 ToolBar background colors



Customizing ToolBar bottom border color

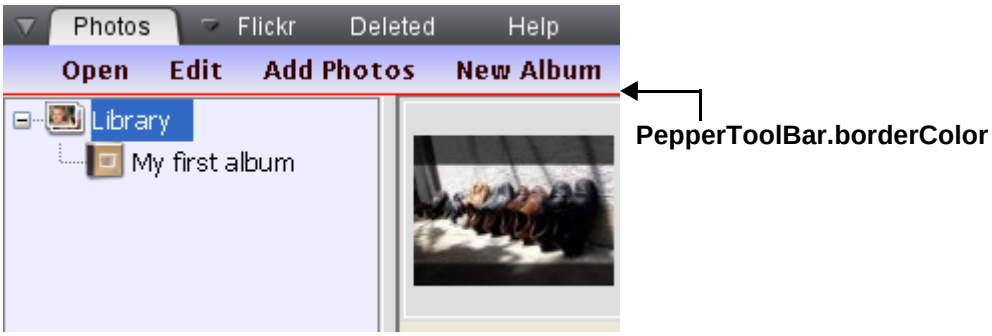
The color of the bottom border of Java ToolBars is determined by the *keeper.css* `ToolBar.borderColor` class, as shown in Example 10–10.

Example 10–10 Setting the ToolBar bottom border color

```
ToolBar.borderColor {
   color: red;
}
```

Figure 10–8 shows a ToolBar's bottom border with the *keeper.css* class definition as shown in Example 10–10.

Figure 10–8 A ToolBar's customized bottom border color



Customizing ToolBarButton font and font color

The font and font colors of ToolBarButtons are defined by the following *keeper.css* classes:

- ToolBar.font

  Defines the font of the ToolBarButton.

- ToolBar.foreground

Defines the color of the ToolBarButton text.

Example 10–11 shows a sample definition of the two colors in *keeper.css*.

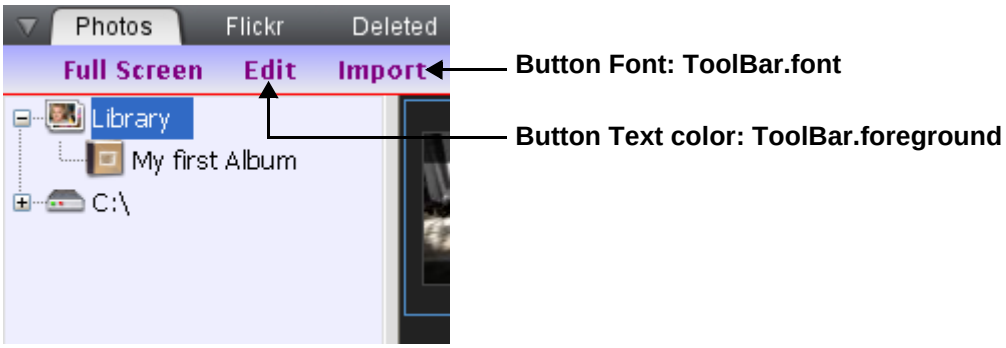
Note: Colors are defined using CSS syntax.

Figure 10–9 shows the Photo Library's ToolBar as it appears with the *keeper.css* class definitions in Example 10–11.

Example 10–11 Setting ToolBarButton font and font color

```
ToolBar.font {
   font-family: Lucida Sans DemiBold Roman;
   font-style: plain;
   font-size: 12px;
}
ToolBar.foreground {
   color: purple;
   /*color: #330000;*/
}
```

Figure 10–9 ToolBarButton font and font colors



Customizing ToolBarButton mouse pressed colors

Two additional ToolBarButton colors are customizable:

- The background color when a ToolBarButton is pressed is determined by the *keeper.css* `ToolBar.selectionBackground` class.
- The foreground color of the text when a ToolBarButton is pressed is determined by the *keeper.css* `ToolBar.selectionForeground` class.

The following example shows a sample definition of the two colors in *keeper.css*.

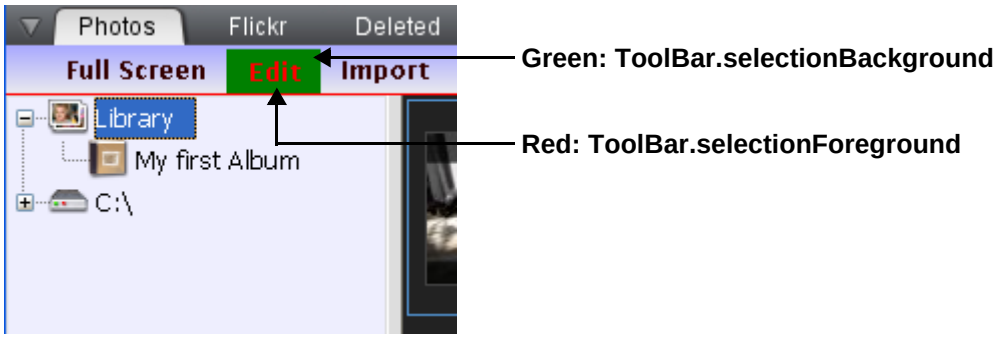
Example 10–12 Setting the ToolBarButton mouse pressed background and text color

```
ToolBar.selectionBackground {
   color: green;
}
ToolBar.selectionForeground {
   color: red;
```

```
}
```

Figure 10–10 shows a ToolBar's bottom border with the *keeper.css* class definition as shown in Example 10–12.

Figure 10–10 A ToolBarButton mouse pressed background color and mouse pressed text color



Custom themes

This section explains how to create custom themes that control the visual styling of the framework and its applications.

Themes overview

When the framework runs, its overall visual styling is controlled by its *theme*. The theme is defined by an archive in the framework's root directory. The default theme archive is named *common-resources.zip*.

You can make a custom theme archive. For example, suppose you develop a custom theme named *premium*. You can turn it into a custom theme archive (by building it in the SDK) named *premium-resources.zip*. You can then use it to display the framework and applications with custom visual styling.

The basic steps for creating a custom theme are:

- Creating a directory for the theme directory in the SDK

  See "Creating a theme area in the SDK" on page 10-169.

- Adding files to it that you want to customize

  They have to be added in the same subdirectory locations as they appear in the default theme.

  See "Adding files to the theme area" on page 10-169.

- Building the theme

  See "Building a theme" on page 10-171.

- Placing the theme archive in the framework's root run-time directory

  See "Adding a custom theme to a framework" on page 10-171.

- Launching the framework to use the custom theme

See "Launching the framework to use a custom theme" on page 10-172.

What's in a theme archive?

The theme archive provides a wide range of files that control the framework's visual styling, such as:

- CSS files,
- Image files for Section backgrounds, icons and other things,
- Properties files that determine displayed text, keyboard shortcuts, enable Java-based language localization, and more,
- XSL files used during framework's first-launch configuration for such things as the default list of email providers displayed to a user as they set up their Mail application, and
- Common files used by all applications, such an Page definitions for Web and Help Sections.

Theme archive files are arranged in particular directories.

For example, a theme archive requires a root *resources* directory.

resources has a number of subdirectories. Probably the most import is *resources/styles*. In here you can find two CSS stylesheets (*styles.css* and *keeper.css*) that are your first stop for most theme customization. *resources/styles* also contains important image files such as *background.jpg* and *floatingselection.jpg*, referred to by CSS classes and important in setting the overall framework look and feel.

A custom theme only needs customized files

Your custom theme only needs to contain the theme files that you modify. That is, a custom theme does not require all files in the default theme archive.

When you build your theme in the SDK, a complete theme archive is created using the default theme archive (*common-resources.zip)* that is included in the SDK's *lib* directory as a base. Any files in your custom theme are added to the new theme archive, replacing any files in the default theme archive that are named the same and that are in an equivalent directory location. The result is a complete theme archive that includes your customizations.

Themes are transparent to applications

From the point of view of an application (including the Keeper application), there is only one theme: the current theme. Its files are always accessed by the framework and by applications in the same manner, regardless of whether the current theme is the default theme or a custom theme.

Accessing theme files in custom applications

Naturally, applications often access theme files. For example, you can design a Page that, when converted into HTML by the Page's XSL transform, links to the main theme CSS stylesheet: *styles.css*.

styles.css resides in the following location:

(themeName)-resources.zip/resources/styles/styles.css

Theme files are accessed within XSL using the `$platform` framework parameter followed by the specific theme directory path and then by the name of the desired file.

> **Note:** Do not state the theme archive name, only its subdirectories, starting with the required *resources* subdirectory.

For example, to use XSL to link to *styles.css* in a generated HTML page, use the following line of code in the XSL transform:

```
<link href="{$platform}/resources/styles/styles.css" rel="stylesheet"
type="text/css" />
```

The Hello World Tutorial provides an example of this. See “Customizing the display with CSS” on page 6-105.

Creating a theme area in the SDK

This procedure explains how to create an area in the SDK for developing a custom theme.

Be default, the SDK contains the following directory:

pepper-sdk/themes

For information about SDK directories, see “SDK directories” on page 2-5.

This is the root directory for all theme development work.

> **Note:** *pepper-sdk/themes* contains a subdirectory named *build*. This directory is used when building custom themes. It contains the *build.xml* file, which should not be modified. See “Building a theme” on page 10-171.

Procedure:

1. Determine your theme name.

   Your theme name is an important constant. It is used to name the theme’s SDK directory, is the first part of the theme’s archive filename, and is used to configure the framework’s -Dtheme system property.

2. Create a subdirectory of *pepper-sdk/themes* that is named the same as your theme.

   For example, to make a theme named “premium,” create the following subdirectory:

   pepper-sdk/themes/premium

3. Every theme has to have a root *resources* directory as well, so create that:

   pepper-sdk/themes/premium/resources

Procedure complete.

Adding files to the theme area

The default theme archive (*common-resources.zip*) contains a specific set of directories and files. You can customize any of these files and place them in your theme. Any applications, including the Keeper, that access any of the files you customize has its visual styling modified as a result.

See “Customizing files derived from the default theme archive” on page 10-170.

You can also add your own files. These are files that do not exists in the default theme archive. Any files you add can only affect applications you develop or customize. Adding files not found in the default theme does not affect the visual styling of the framework or non-custom applications because they never access them. However, adding new theme files may be efficient when developing multiple applications that use the same file because the alternative would be to place the file separately in each of these application's *design* directories. This is not only an unnecessary duplication. It also prevents modifying sets of applications by modifying a single file.

See "Adding custom files" on page 10-170.

Customizing files derived from the default theme archive

This procedure explains how to customize files that exist in the default theme archive.

Procedure:

1. The first step when customizing a default theme file is to extract the *custom-resources.zip* file from *pepper-sdk/lib* to a convenient location.

   Note: Do not extract *custom-resources.zip* into your theme directory.

2. Find the default theme file you want to modify in the extracted archive and take note of its directory location.
3. Create the same directory (or series of nested directories) in your theme directory.

   For example, suppose you want to change the background image of the framework **Applications** Tab. This file is named *background.jpg* and resides in *resources/styles.*

   Create the *resources/styles* directory in your theme archive. For example, if your theme is named *premium*, create the following:

   pepper-sdk/themes/premium/resources/styles

4. If you are replacing a file in its entirety, simply place the new file in the correct directory in your theme directory tree and ensure it is named exactly the same as the file in the default theme archive.

   Continuing with the example of changing the **Applications** Tab background image: place the new *background.jpg* file in the following location:

   pepper-sdk/themes/premium/resources/styles/background.jpg

   When the theme is built, your file is used and replaces the default *background.jpg* file.

5. If you are modifying a part of a default theme file, for example, if you are adding a class to a default CSS stylesheet, copy the file from the extracted *common-resources.zip* and place it in the equivalent directory in your custom theme directory tree.

   When the theme is built, your modified version is used instead of the default version.

Procedure complete.

Adding custom files

You can place new theme files (ones that do not exist in the default theme archive) into any location in your theme SDK area.

Note: It is recommended to place all files in a new subdirectory of *resources* in order to keep custom theme files separate from default theme files.

For example, let's say you want to create a new CSS stylesheet that contains classes you want to use in several new applications. You decide to name the CSS stylesheet *premium.css*. You might decide to put it in a new directory named *premiumStyles*, as follows:

pepper-sdk/themes/premium/resources/premiumStyles/premium.css

As we have seen previously, applications' XSL transforms use the `$platform` framework parameter followed by the appropriate directories to access the new stylesheet.

For example, to use XSL to link to *premium.css* in a generated HTML page, use the following line of code in the XSL transform:

```
<link href="{$platform}/resources/premiumStyles/premium.css"
          rel="stylesheet" type="text/css" />
```

Building a theme

This procedure explains how to build a custom theme.

Procedure:

1. Open a command-line session and set the session's required SDK environment variables as usual.

   For information on setting a session's required SDK environment variables, see "Setting the Build Environment Variables" on page 11-190.

2. Change to the *pepper-sdk/themes/build* directory.
3. Enter the **`ant`** command.

   This builds all themes.

   When the build process is complete, the session output reports build success or failure.

   Successfully built themes are placed in the *themes/build* directory and are named based on the theme name.

   For example, if you build a theme named *premium*, the following theme archive file is generated:

   pepper-sdk/themes/build/premium-resources.zip

Procedure complete.

Adding a custom theme to a framework

After creating a custom theme archive, add it to a framework by copying the custom theme archive into the root directory of the framework installation.

On the Pepper device, the root framework installation directory is:

/opt/pepper

On Windows, the root framework installation directory is:

My Documents/Pepper

Note: Simply adding the custom archive to the run-time framework directory does not enable it. You also have to launch the framework using an system property that specifies the theme name, as explained next.

Launching the framework to use a custom theme

After placing a custom theme archive in the framework's root run-time directory, you can launch the framework to use it instead of the default theme archive.

This is done by configuring the *-Dtheme* framework system property to specify your theme name. The theme name is the name of the theme's SDK directory.

Note: The theme name is not the file name of the custom theme archive file.

For information about the custom theme name, see "Creating a theme area in the SDK" on page 10-169.

System properties are configured differently depending on the framework's platform.

For information about configuring framework system properties, see "Setting framework system properties" on page 2-15.

How to customize web bookmarks

Users can add, delete and edit Web Section bookmarks. This functionality is built into Sections declared as type web (`type="web"`) in *FactoryBuild.xml*.

For information on Section types, see "Attribute: type" on page B-220.

At first launch, a Web Section typically provides a set of default bookmarks. This makes it easy to create Web Sections focused on a particular subject. For example, a web Section in a recipe application might provide bookmarks for recipe web sites.

The next section explains how bookmarks work.

For the procedure to customize the set of bookmarks provided with any web Section at first launch, see "How to customize web bookmarks" on page 10-172.

Bookmark architecture overview

A Web Section's SectionPage displays the default list of bookmarks at the application's first launch. (The SectionPage also provides a list of clippings and browsing history. These are typically empty at first launch.) A Web Section's SectionPage XML instance file is typically named *web.xml* and resides in the application's *data* directory.

Each bookmark is a Page. Default bookmark Page XML files are usually given a meaningful name, such as *pepper.xml*. Bookmark files created at run-time have framework-generated names such as *1152649919327.xml*. Bookmark XML instance files typically reside in the application's *data/Web* run-time directory.

For information about XML instance files, see "Application structure — a run-time sample" on page 4-36.

As the user adds, edits and deletes bookmarks, framework caching rules update the SectionPage to ensure it provides a current list of Page bookmarks.

For information about caching rules, see "Caching" on page 4-53.

Each bookmark provided at first launch with a Web Section is a pre-built Page.

For information about pre-built Pages, see "Pre-built Pages" on page 4-69.

A Web Section does not use a pre-built SectionPage. Instead, the SectionPage XML instance file is created by the build system and automatically includes cached bookmark information for each pre-built bookmark Page.

An example

The Hello World Tutorial application has the following SDK subdirectory.

prebuilt/websection

The following example shows the *FactoryBuild.xml* Section declaration that specifies this directory as the root pre-built directory for the web Section.

```
<section name="NameKey.WebSection" type="web" id="data/web" builtin="true">
   <prebuiltPagesDir>../prebuilt/websection</prebuiltPagesDir>
</section>
```

In the SDK, this directory contains the following subdirectory:

prebuilt/websection/bookmarks

According to the rules for pre-built sections, during the build, the *bookmarks* directory is copied to the application's *data* directory, resulting in the following run-time directory:

data/bookmarks

Also according to the rules for pre-built sections, each XML instance file, in this case bookmark Page XML instance files, must contain an `id` attribute that states its run-time location and filename (without the ".xml" file extension). For a bookmark XML instance file named *pepper.xml*, the `id` attribute in the pre-built *pepper.xml* file must be:

```
id="data/bookmarks/pepper"
```

This `id` value is required because in the executing application, the bookmark file:

- Is named *pepper.xml*, and
- Resides in the *data/bookmarks* directory.

You can force the build system to automatically create the correct `id` attributes by simply deleting the `id` attribute from each pre-built bookmark file's `<page>` attribute.

For example, assume you created the pepper bookmark file in a different application and when you pasted it into your application's correct SDK directory (*prebuilt/web/bookmarks*), its `<page>` attribute is as follows:

```
<page name="Pepper" type="web-bookmark" id="data/folder/pepper">
```

Note: In most cases, the `id` attribute is correct, since web Sections generally use the same bookmark folder structure. However, the developer can use a different structure. In this example, the developer changed the bookmark run-time folder from the usual *data/bookmarks* to *data/folder.*

To fix the bookmarks you have pasted into the SDK pre-built folder, simply delete entire `id` attribute from the `<page>` element in each bookmark XML file, as follows:

```
<page name="Peppe" type="web-bookmark">
```

The result is that the build system detects the missing `id` attributes and inserts correct ones based on the Section's definition in *FactoryBuild.xml*, the directory in which the file resides, and the file name.

For more information about using pre-built Pages, see "Pre-built Pages" on page 4-69.

Customizing bookmarks

This section explains how to customize bookmarks, which means creating a default set of bookmarks that are packaged with a web Section and are present at first launch. You can customize the bookmarks for any web Section, including ones in applications you developed from scratch in the SDK and ones in existing applications that you did not develop. (In this latter case, you have to port the application into the SDK, as explained in this section.)

Customizing bookmarks is fairly straightforward, but it does involve a few steps. As noted, the application whose bookmarks you want to customize must reside into the SDK. Then, you use any web Section to generate the set of bookmarks you want, each of which is created as an xml file. You copy these files into the correct SDK directory, delete the `id` attribute from each, and then you are ready to rebuild the application.

Customizing default bookmarks

This procedure covers:

- Porting an application whose default bookmarks you want to customize into the SDK.
- Generating a set of bookmarks using any application's Web Section that you can then use as your set of default bookmarks.
- Porting the new bookmark XML instance files to the application in the SDK.
- Rebuilding the application for distribution.

Procedure:

1. If the application whose bookmarks you want to customize is not already in the SDK, port it to the SDK.

   For information, see "How to port an application into the SDK" on page 10-181.

2. Use any application with a Web Section to generate the bookmarks you want to use as your default bookmarks.

   Tip: Use the application to delete any bookmarks you do not want to include as default bookmarks.

The bookmark XML instance files reside in a subdirectory of the application's run-time *data* directory, typically *data/bookmarks*. However, depending on how the web Section was designed, they may reside in a different subdirectory of *data*, so you may have to look around.

3. Copy the desired bookmark XML instance files you have created into the SDK, as follows:
   a. Determine the required pre-built directory for the SDK Web Section whose default bookmarks you are customizing by examining the Section's `<prebuiltPagesDir>` element in *FactoryBuild.xml*.

      Note: Examine *FactoryBuild.xml* for the application whose bookmarks you are customizing in SDK.

      If the directory does not exist for the application in the SDK, create it.

   b. Create a further subdirectory to contain the bookmarks, typically *prebuilt/websection/bookmarks*
   c. Copy the bookmarks you created previously into this subdirectory.

      Tip: Delete any bookmarks you do not want to include as default bookmarks for the Web Section.
4. Delete every bookmark's `id` attribute from its `<page>` element.
5. Build the application to create its distribution package.

   For information, see "Building Applications" on page 11-189.

Procedure complete.

How to customize an application's help

Each application, including the Keeper, has a help Section. Help Sections are a unique Section type recognized by the framework.

For information on Section types, see "Attribute: type" on page B-220.

This section explains how help Sections work so that you can create your own, create multiple versions to support localization, and customize existing help Sections.

Help Section overview

All Help Pages (including the SectionPage) are pre-built.

See "Pre-built Pages" on page 4-69.

Because of Help Section support for localization, pre-built Page XML files should be placed in a localized directory.

See "How to localize for different languages" on page 10-177.

For example, *US_en* is a localized directory in the following:

`prebuilt\helpsection\help\US_en`

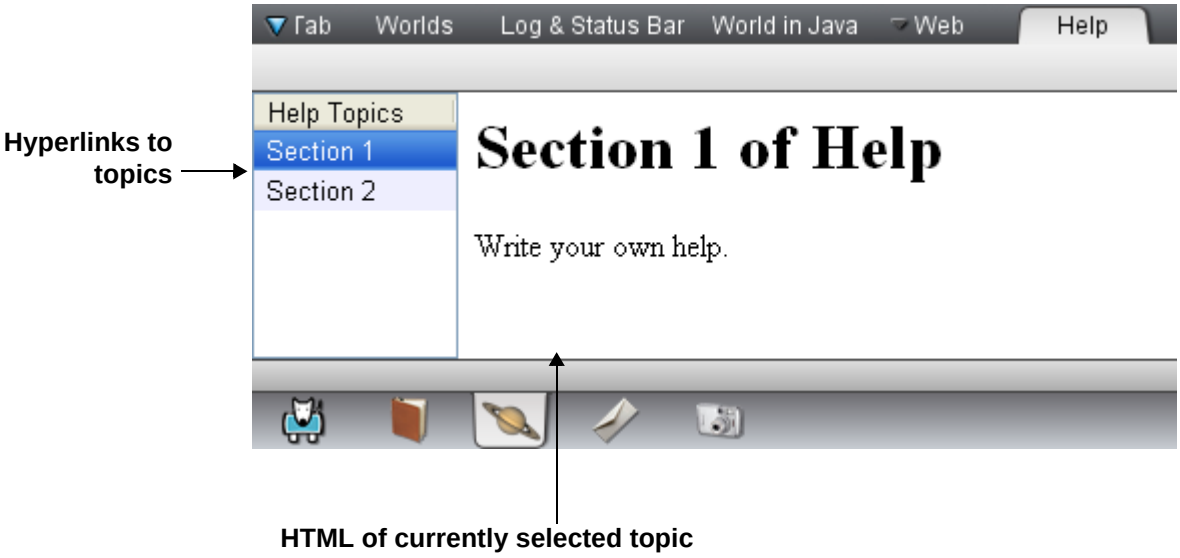
Help Sections are composed of a set of topics. Each topic is a Page that is associated with an HTML file. (XSL is not used to display Help Pages). You specify the HTML file to use for each Help Page in the pre-built Page XML file with the `<template>` element.

For example, the following specifies that the *overview.html* file (in the localized directory) should be displayed for the following Page:

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile deletable="true" readOnly="true">
  <header>
    <template>data/help/locale(overview.html)</template>
    <createDate utc="1039019942806" />
  </header>
  <body>
    <page name="Overview" />
  </body>
</pageFile>
```

A help Section's user interface provides a table of contents of topics. The table of contents displays in a panel on the left side of the help Section. Each topic is clickable and hyperlinked to the topic's Page, which displays the designated HTML file in a panel on the right side of the help Section, as shown in the following.

Figure 10–11 Sample help system



The list of topics is derived from the pre-built SectionPage. As with all SectionPages, a Help SectionPage contains a `<section>` element that contains `<page>` elements for each Page. You have to manually create this in your pre-built SectionPage.

The displayed hyperlink text for each topic is derived from each `<page>` element's `name` attribute. Each `<page>` element's `id` attribute specifies the localized path to the Page file (without a file extension).

The following example shows a pre-built SectionPage.

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile deletable="false" readOnly="true">
    <header>
        <pkgName>Remote desktop</pkgName>
        <pkgVersion>2.1.0</pkgVersion>
        <template>design/help.xsl</template>
        <defaultPageType>default</defaultPageType>
        <createDate utc="1089919577296" />
    </header>
    <body>
        <section name="NameKey.Help" type="help" builtin="true"
id="data/help/locale(Help)">
            <page name="Section 1" id="data/help/locale(section1)" />
            <page name="Section 2" id="data/help/locale(section2)" />
        </section>
    </body>
</pageFile>
```

Customizing help

This procedure explains how to customize a help Section.

Procedure:

1. Port the application into the SDK.

   See “How to port an application into the SDK” on page 10-181.

2. Edit any help HTML files you want to.
3. To add new help topics:
   - Create a pre-built Page for each new topic.
   - Create an HTML file for each Page.
   - In the Page, specify the localized HTML file with the `<template>` element.
   - Manually add the Page’s `<page>` element (including the `name` and `id` attributes) to the pre-built SectionPage XML file.

4. Rebuild the application.

   See “Building Applications” on page 11-189.

5. Distribute the application.

   See “Adding and Distributing Applications” on page 12-195.

Procedure complete.

How to localize for different languages

The framework leverages Java’s *localization* capabilities.

Localization allows you to create different versions of applications that are appropriate for different *locales*. A locale is defined by its region and language.

The material that is localized may simply be written in different human languages, for example French or Korean. It may have different content. For example, your French Help files may have different content than your English ones.

The localized versions are kept in different Java *properties* files, or, for localized help systems, in different sibling directories that are named based on the locale.

When the framework launches, it detects the locale and uses the appropriate properties and help files. Locale detection automatically occurs based on the system's locale. Or, you can explicitly set the locale at framework launch time using two -D system properties (one for region and one for language) passed to the framework at launch time.

For information on setting the locale at framework launch time, see "Setting framework system properties" on page 2-15.

In general, you can localize the following:

- Text displayed in the user interface throughout the framework and applications, for example Tab text and button text

  For information, see "Customizing user interface widget display text" on page 10-178.

- Text displayed in Java Sections

  This is covered in Phase Four of the Hello World Tutorial. See "Using localizable label text" on page 9-144.

- Help files, which can display in the language appropriate for the locale

  For information, see "Customizing help for different languages" on page 10-181.

Customizing user interface widget display text

Text displayed in user interface widgets can be localized to support multiple languages.

Such text includes:

- Tab text
- Button text
- Pull-down list text
- Tooltip text

Key properties files for user interface widget text localization include:

- General user interface widget display text

  For information, see "Localizing CommonStrings.properties" on page 10-179.

- Time zone text

  For information, see "Localizing TimezoneCatalog.properties" on page 10-180.

- Application-specific user interface widget display text

  For information, see "Localizing PackageStrings.properties" on page 10-180.

Creating localized properties files

You can create a localized version of a properties file by:

- Making a copy of the file in the same directory as the default properties file.
- Naming it according to Java's localization rules by adding characters that specify its locale.

  The locale is specified by adding: underscore + (two letter language code) + underscore + (two letter region code), just before the file extension. For example, if the default properties file is named *PackageStrings.properties*, an English, United States version file has the following filename: *PackageStrings_en_US.properties*.

  You could create a version for French speaking users in Canada as follows: *PackageStrings_fr_CA.properties* file. If the framework is launched on a French Canadian system, or if it is launched with French Canadian `-D` arguments, it uses the *PackageStrings_fr_CA.properties* file instead of the default *PackageStrings.properties* file when looking up text for display in the user interface.

  > **Note:** Java localization file naming rules are beyond the scope of this document but are explained on many sites on the internet.

- Editing the contents to provide the localized text.

As noted, the contents of the properties file also have to be customized for the correct language. As explained in the Hello World Tutorial, properties files contain key-value pairs, with one key-value pair on each line.

For example, the following shows a single key-value pair that is used to derive the display text for a Tab:

```
NameKey.Worlds=Worlds
```

Where:

- Everything to the left of the equals sign ("=") is the key.

  The key must not be modified. This is what the framework code looks up when accessing the file.

- Everything to the right of the equals sign ("=") is the value.

  The value is modified as appropriate for the localization.

To localize this for French speakers, it might be changed to the following:

```
NameKey.Worlds=Les Mondes
```

Localizing *CommonStrings.properties*

Text displayed in user interface widgets that is not application-specific, that is, widget text that is common among all applications, is derived from *CommonStrings.properties*.

> **Note:** As with all properties files, you must only modify the value part of each key-value pair. See "Creating localized properties files" on page 10-179

CommonStrings.properties is a theme archive file. You can create a localized version this file (with a localized filename) by creating a custom theme and including it in your custom theme.

For information on creating custom themes, see “Custom themes” on page 10-167.

Localizing *TimezoneCatalog.properties*

On a Pepper device, users can set the time. (On the Pepper Desktop, the time is controlled by the operating system.)

A list of time zones is displayed to the user. This list can be localized to provide place names in the appropriate language. The list of time zones is derived from *TimezoneCatalog.properties*.

TimezoneCatalog.properties is a theme archive file. You can create a localized version this file (with a localized filename) by creating a custom theme and including it in your custom theme.

> **Note:** As with all properties files, you must only modify the value part of each key-value pair. See “Creating localized properties files” on page 10-179

For information on creating custom themes, see “Custom themes” on page 10-167.

Localizing *PackageStrings.properties*

Text displayed in application’s specific user interface is derived from each application’s *PackageStrings.properties* file.

To localize this, port the application into the SDK, create a localized version this file (with a localized filename) and rebuild the application to create a new distribution.

> For information about porting an application into the SDK, see “How to port an application into the SDK” on page 10-181.

Launching the framework with a specified locale

By default, the framework automatically detects and uses the operating system’s locale.

You can launch the framework with two arguments that explicitly set the locale the framework uses:

- -Duser.language=fr

  The two characters on the right side of the equals sign (“=”) set the language using Java’s standard language code.

- -Duser.region=CA

  The two characters on the right side of the equals sign (“=”) set the region using Java’s standard language code

Taken together, these two sample localization properties set the locale to the French language in Canada.

These arguments can be included in a launch resource files so that this occurs automatically, and transparently to the user, at launch time.

> For information about setting -D system property, see “Setting framework system properties” on page 2-15.

Customizing help for different languages

You can create different versions of a help system to support different languages.

For information, see "How to customize an application's help" on page 10-175.

How to port an application into the SDK

Porting an application into the SDK is useful when customizing an application you did not develop.

> **Note:** You cannot customize the Java aspects of an application you did not develop without its source files.

Because you can rebuild the application in the SDK, the customizations are bundled by the build process into the application's distribution package.

This is useful in the following cases:

- Customizing the application and rebuilding it for redeployment

  You can customize Sections and Page definitions, icons, images, and other application-specific presentation aspects of an application, rebuild it and distribute it.

- Rebuilding the application from modified pre-built Pages

  For example, you may want to change the default bookmarks for a Web Section. You can do this by porting the application containing the Web Section into the SDK, modifying its pre-built bookmark Pages and rebuilding it. You could also modify an application's pre-built help Pages, then rebuild and distribute it.

Which applications can be ported to the SDK?

You can port any application from a run-time directory into the SDK, including the Keeper application itself.

Is the rebuilt application complete?

After rebuilding an application in the SDK, its non-Java contents reside in the application's *dist* directory in the SDK, enabling easy distribution.

However, for applications you have ported to the SDK, the *dist* directory does not contain the application's jar file (or jar files), nor does it contain other files that may be listed as required resources in the application's *package.ppld* file (*binder.ppld* in the case of the Keeper application).

So, before distributing the rebuilt application from the SDK, you need to copy any jar files (or other special resources) that are not created by the SDK build from the application's run-time root directory into the SDK *dist* directory.

After rebuilding, how is the application deployed?

You can manually add a rebuilt application to a single framework using Debug Mode's **Add Application** menu item.

For information, see "Adding an application" on page 2-12.

You can distribute a rebuilt application from a web site.

For information, see Distributing an application on the web.

You can deploy a rebuilt application using an update server.

For information, contact Pepper.

Porting an application to the SDK

This procedure explains how to port an application from a run-time framework directory into the SDK.

Note: This procedure uses the Keeper application as an example.

Procedure:

1. Create a directory for the application in the SDK *pepper-sdk/applications/* directory.

   It is recommended to use the same directory name that the application has in the run-time framework, excluding everything from the dash to the right: “-”.

   For example, the bundled framework application has a run-time directory named something like this:

   Keeper-750289dc176f2c65b5ef

   (Everything to the right of the dash is the package GUID and varies from installation to installation.)

   Note: Some applications have a package GUID of “0”.

   In keeping with this recommendation, you would name the Keeper’s SDK directory as follows:

   Keeper

   Note: The application’s SDK directory name is an important constant that must be entered into the application’s *build.xml* file, as discussed below.

2. Create the following subdirectory for the application in the SDK:

   pepper-sdk/applications/Keeper/env

3. Create a *build* subdirectory, as follows:

   a. Copy *applicationTemplate/build* directory into *pepper-sdk/applications/Keeper/*

4. Open *build/build.xml* for editing and change the `<project>` element’s `name` attribute to the application’s SDK directory name, for example:

```
<project name"Keeper" default="all" basedir=".">
```

5. Create the SDK *Keeper/design* directory from the run-time application.

   There are two options.

   - You can use the original built design files contained in *design.zip*.
   - You can use design files that were previously extracted in the run-time Keeper in Design Mode contained the application’s run-time *design* directory.

To create the SDK *design* directory from the unextracted files that represent the application's original built state, do the following:

a. Copy *design.zip* from the application's run-time directory into *pepper-sdk/applications/Keeper*.

b. Extract *design.zip into pepper-sdk/applications/Keeper/design*

design.zip contains a *design* directory, which in turn contains many files and subdirectories. The results of the extraction must be that this contained *design* subdirectory is a child subdirectory of *pepper-sdk/applications/Keeper/*, as follows:

pepper-sdk/applications/Keeper/design

c. Delete *design.zip* from the SDK.

To create the SDK *design* directory from the design files previously extracted in Design Mode (and probably customized), do the following:

a. Copy *design* from the application's run-time directory into *pepper-sdk/applications/Keeper*, resulting in the following:

pepper-sdk/applications/Keeper/design

If the application has any Sections that use pre-built Pages, you must retrieve the Page XML instance files from the run-time *data* directory and place them in correctly named directories in the SDK.

The following steps explain how to check whether there are any Sections using pre-built Pages, and if so, how to retrieve the files and place them in the correct SDK directories.

6. Determine the *prebuilt* subdirectories that are required, if any, as follows:

a. Open *pepper-sdk/applications/Keeper/design/FactoryBuild.xml*.

b. For each pre-built section find the pre-built directory that must be created for the application in the SDK.

Tip: Read through this material first to be sure you understand it. The outcome of this step should be a list of all Sections defined in *FactoryBuild.xml* that use pre-built Pages and, for each, the specific pre-built directory that is expected and that must therefore be created in the SDK.

Every Section is declared as a `<section>` element. There are two ways Sections are declared as using pre-built Pages. They have either a `src` attribute or a child `<prebuiltPagesDir>` element.

You need to find every Section that uses either of these two ways to identify the Section as using pre-built Pages.

Example 10–13 shows the use of the `src` attribute in the Keeper application's Help Section's definition. The value of the `src` attribute, `../prebuilt/HelpSection`, indicates that the application's SDK directory must have a *prebuilt/HelpSection* subdirectory tree. This subdirectory tree must contain all pre-built Pages for the Section,

including the SectionPage. These pre-built Pages must be in a subdirectory of *prebuilt/HelpSection* that is consistent with their `id` attributes, as explained below.

Example 10–13 Pre-built Section identified with the `src` ***attribute***

```
<section name="NameKey.Help" type="help" builtin="true"
      id="data/Help/locale(Help)" src="../prebuilt/HelpSection" />
```

Example 10–14 shows the use of the `<prebuiltPagesDir>` element in the Settings Section definition. The text contents of `<prebuiltPagesDir>` element, `../prebuilt/SettingsSection`, indicates that the application's SDK directory must have a *prebuilt/SettingsSection* subdirectory. This subdirectory must contain all pre-built Pages for the Section. (Because the `src` attribute is not used, the SectionPage is not pre-built.) As with the previous example, these pre-built Pages must be in a subdirectory of *prebuilt/HelpSection* that is consistent with their `id` attributes, as explained below.

Example 10–14 Pre-built Section identified with the `<prebuiltPagesDir>` ***element***

```
<section name="NameKey.Settings" id ="data/Settings" type="settings" builtin="true">
   <prebuiltPagesDir>../prebuilt/SettingsSection</prebuiltPagesDir>
</section>
```

For the Keeper application, based on the two examples shown here, two pre-built directories are required:

pepper-sdk/applications/Keeper/prebuilt/HelpSection

pepper-sdk/applications/Keeper/prebuilt/SettingsSection

If you have found no Sections using pre-built directories, this procedure is complete and you can modify and build the application as you would any other SDK application.

7. Create the pre-built subdirectories identified as required in the previous step.

   As noted, for the Keeper application in this running example, based on the two results of the previous step, the following two pre-built directories must be created:

   pepper-sdk/applications/Keeper/prebuilt/HelpSection

   pepper-sdk/applications/Keeper/prebuilt/SettingsSection

8. Port the application's *data.zip* file to the SDK and extract it as *dataFromRuntime*.

   Pre-built files are contained in the run-time *data.zip* file. *data.zip* contains a *data* directory, which in turn contains files and subdirectories. There is a subdirectory (or subdirectory tree) for each Section's pre-built files. These subdirectories have to be copied into the SDK *pre-built* directories that were created in the previous step.

   The following sub-steps cover porting and extracting *data.zip*. The following step covers copying the pre-built files into the new *pre-built* directories.

   a. Copy *data.zip* from the application's run-time directory into *pepper-sdk/applications/Keeper*.

   b. Extract *data.zip* into *pepper-sdk/applications/Keeper/*

      The results of the extraction must be that this contained *data* subdirectory is a child subdirectory of *pepper-sdk/applications/Keeper/*, as follows:

pepper-sdk/applications/Keeper/data

c. Rename the *data* directory to:

pepper-sdk/applications/Keeper/fromDataRuntime

The *data* directory is created by the build process. When the build occurs, it is populated with pre-built Page files and with new files created during the build that are based on Section definitions in *FactoryBuild.xml* in combination with Page definitions in *PageTemplates.xml*. Renaming *data* to *dataFromRuntime* enables you to use it as a resource containing pre-built files while preventing the build from overwriting it.

9. Copy pre-built directories and Pages into the new pre-built directories.

To understand this step, consider that during a build, everything (directories and files) contained inside in a pre-built directory specified with a `src` attribute or `<prebuiltPagesDir>` child element is copied into the *data* directory (which is compressed into *data.zip*).

For example, before a normal build, the Keeper's pre-built Help directory looks like this:

prebuilt/HelpSection/Help/(locale directories)/(pre-built files)

As we have seen, for this section the `src` attribute specifies finding pre-built subdirectories and files for this Section here:

`../prebuilt/HelpSection`

Therefore during a normal build, the following subdirectories and files (that is, those inside *prebuilt/HelSection*) are copied into the *data* directory:

Help/(locale directories)/(pre-built files)

Resulting in this:

data/Help/(locale directories)/(pre-built files)

To port the application into the SDK, you have to do the reverse:

Copy:

Help/(locale directories)/(pre-built files), which resides in *dataFromRuntime*

Into:

prebuilt/HelpSection

Resulting in this:

prebuilt/HelpSection/Help/(locale directories)/(pre-built files)

Note: *(locale directories)* are used to provide sets of pre-built files that are sorted by locale-specific directories with names that follow the Java localization standards. Each locale-specific directory contains all required files, but they are written in a human language appropriate for the locale.

For the other Section using pre-built Pages in this running example, the Settings Section, you would have to:

Copy:

Settings/(pre-built files), which resides in *dataFromRuntime/*

Into:

prebuilt/SettingsSection

Resulting in this:

prebuilt/SettingsSection/Settings/(pre-built files)

10. If you want to use any of the files in the run-time *data* directory as pre-built files in the revised application, copy the files from the *dataFromRuntime* the appropriate *prebuilt* subdirectory.

    For example, if you have generated bookmarks for a web Section, and you would like them to appear at first launch of the rebuilt application, copy the bookmark XML instance files into the appropriate SDK directory.

    Note: If you copy any files from *dataFromRuntime* into the SDK and if you modify any of them in the SDK, you must delete the corresponding file in the run-time *data* directory. This because at run-time, an application always checks for a file in the run-time *data* directory first. If it finds it, it uses it and never checks for the file in *data.zip.* The outcome of the SDK build of the application is a new *data.zip* file. If you don't delete the file from *data* its new version in *data.zip* is never used.

11. Copy all files except for the application's ppld file from the application's root run-time directory into the application's SDK *dist* directory.

    This ensures the *dist* directory contains all required resources, such as the application's jar file, any other application-specific jar files the application may require, and any other required files.

    Note: The application's required resources are listed in its ppld file inside the `<resources>` element. Although the Java run-time environment is listed there, you do not need to copy it because it is included with every Keeper.

Procedure complete.

Customizing an application's Sections

As we have seen, the Section instances (Tabs) of an application at first launch are each declared in its *FactoryBuild.xml* file. You can modify this file to remove Section instances that are created by the build. You can also add Section instances.

Remove a Section instance by commenting out or deleting its `<section>` element from the *FactoryBuild.xml* file. Then, rebuild the application in the SDK. However, If the application was previously installed in a framework, its previous Section instances may exist as XML files in its *data* directory. Therefore, before refreshing the run-time application from the newly built version, delete the run-time *data* directory.

A new Section can be created by the build with a new `<section>` element in the *FactoryBuild.xml* file.

Note: In the current release, you cannot use new framework Actions (classes that extend ProgramAbstractAction) in Sections you add to applications that you did not develop because new framework Actions must be registered in the application's base class, and you do not have the application's source code for modification.

Adding a new Section is development work that requires the same knowledge as adding Sections to your original applications, which is the topic of much of this book and therefore cannot be explained in full here.

In both cases, deleting and adding Sections, the application has to be ported into the SDK to rebuild it.

11

Building Applications

What is Building a Pepper Application?

Applications are developed by creating and editing a wide range of files, including:

- XML files
- XSL files
- Java source files
- JavaScript files
- CSS files
- Other files

These files reside in directory locations that are precisely specified.

When you build an application, a number of operations occur, including:

- Java source files (.java files) residing in *pepper-sdk/applications/(application)/src/*.** are compiled into Java class files (**.class* files) and bundled into a jar file, which is placed in the *pepper-sdk/applications/(application)/dist* directory.
- Pre-built Pages are copied from specified directories into the *data* directory.
- Non-pre-built Page XML instance files are created from XML definition files that you have created or modified.
- A *pepper-sdk/applications/(application)/data* directory is created and loaded with required files.
- A *pepper-sdk/applications/(application)/dist* directory is created and loaded with required files.

This is not a complete list. The point is that the build system absolutely requires that various directories and files exist in specific locations.

Ant is the Build System

Pepper applications are built with Ant. Ant is an open source freely available build system that is a part of the Apache project. Ant scripts are included in the SDK and are pre-configured to build SDK applications. You do need to download and install Ant itself.

Ant version 1.6 or higher is required. (Ant 1.7 has not been tested.)

For information about Ant, see its web site: http://ant.apache.org

Setting up Application's Build System

Before an application can be built, its *build.xml* file must be edited. It is necessary to enter the application's project name. The project name is used by the build system as the file name of the application's jar file.

For additional information on Java aspects of the build, see "How Java is Compiled and Jarred During the Build" on page 11-193.

For the procedure to modify the *build.xml* file, see "Modifying build.xml" on page 6-92.

Setting the Build Environment Variables

Building is done from the command line. A number of environment variables have to be set in the command line session before building can occur. These can be set manually in the command line window one at a time, or they can be set by customizing a setup file, then executing it in the command line window.

> **Note:** Customizing the setup file is the recommended approach because it is more convenient: once you customize the file once, you can execute it in new command line windows to easily set the build environment.

There are two versions of the setup file. One, *setup.bat*, is for Windows. The other, *setup.sh*, is for Linux. Both files are provided with the SDK in its root directory: *pepper-sdk/*.

This procedure explains how to edit the setup file to customize the environment variables.

Procedure:

1. Open the setup file appropriate to your operating system:
   - *pepper-sdk/setup.bat* for Windows
   - *pepper-sdk/setup.sh* for Linux

   Here are the contents of *setup.bat*:

```
set PEPPER_HOME=c:/pepper-sdk
set ANT_HOME=%ANT_HOME%
set JAVA_HOME=%JAVA_HOME%
set CLASSPATH=%CLASSPATH%;%PEPPER_HOME%/bootstrap/env/anttasks.jar;
set PATH=%PATH%;%ANT_HOME%/bin;%JAVA_HOME%/bin
```

Here are the contents of *setup.sh*:

```
#!/bin/sh
export PEPPER_HOME=$HOME/pepper-sdk
export ANT_HOME=$ANT_HOME
export JAVA_HOME=$JAVA_HOME
export CLASSPATH=$CLASSPATH:$PEPPER_HOME/bootstrap/env/anttasks.jar
export PATH=$PATH:$ANT_HOME/bin:$JAVA_HOME/bin
```

2. Ensure the `PEPPER_HOME` variable is set to the full path to the pepper SDK installation directory.
3. Ensure the `ANT_HOME` variable is set to the full path to the Ant installation directory.
4. Ensure the `JAVA_HOME` variable is set to the full path to the Java SDK directory.
5. Save and close the file.

Procedure complete.

Build commands

This section explains useful Ant commands for building applications.

> **Note:** Ant build commands must be run from the application's *build* directory. For example, if you are going to build the sample Remote Desktop application, you must be in *pepper-sdk/applications/remotedesktop/build*

ant

Builds the current application. It also builds the application's javadoc when the Java source files have been modified.

ant clean

Removes all files for the current application that were generated by a previous build.

> **Note:** This fails if the files cannot be removed for any reason.

ant rebuild

Performs an **`ant clean`** and then an **`ant`**.

Building an Application

After Setting up Application's Build System, you can build it.

Building is done from the command line.

Note: A number of environment variables have to be set before building. These can be set manually in the command line window or they can be set by customizing then executing a setup file in the command line window. See “Setting the Build Environment Variables” on page 11-190.

Procedure:

1. Open a command line window.
2. Change to the *pepper-sdk* directory.
3. Set environment variables by running the setup file, as follows:

   Note: You only need to run the setup program once during a command line session, before the first build in the session, not each and every time before you build.

   Windows, enter **`setup`**

   Linux, enter **`. ./setup.sh`**

   Note: On Linux, the correct command is a period (“.”) followed by a space (“ ”) followed by “./setup.sh”.

4. Move to the application’s *build* directory.

   For example, if you are building the sample Remote Desktop application, whose application directory is named *remotedesktop*, move to:

   pepper-sdk/applications/remotedesktop/build

5. If you have built this application previously, remove all built files that were generated by the build system with the **`ant clean`** command, as follows:

```
>ant clean
```

   Note: It is not always necessary to run **`ant clean`** before building. However, it is a good idea in order to eliminate the possibility that your new build may seem to work but actually does not. This can occur if the application depends on files that are no longer created by the build but that still exist because they were created during a previous build. **`ant rebuild`** executes an **`ant clean`** and then an **`ant all`**.

6. Confirm success of the build clean by analyzing the output.

   The next to last line reports success or failure.

7. Build the application with **`ant all`** command, as follows:

   Note: Instead of separately executing **`ant clean`** and then **`ant all`**, you can execute the single **`ant rebuild`** command, which does both.

```
>ant
```

8. Confirm build success by analyzing the output

   During the build, information is displayed, including the results of compilation and framework activities. The second to last output line reports build success or build failure.

   Note: If the build fails, a good place to start troubleshooting is by analyzing the details of the information output into the command line session during the build.

Procedure complete.

How Java is Compiled and Jarred During the Build

The build system compiles Java source files into Java class files and creates a jar archive file that is included in the application's distribution (*/dist*) directory. This process occurs automatically according to the settings in two files, which must be edited: *build/build.xml* and *design/package.ppld*.

The Java build system has the following parts:

- Java source files are compiled into Java class files.

  All Java source files existing in the application's *src* directory are automatically compiled into Java class files during the build.

- Java class files are bundled into a jar file and placed in the *env* directory.

  The class files are automatically bundled into a jar file that is placed in the application's *env* directory. You specify the jar filename in the application's *build.xml* file in the `<project>` element's `name` attribute. (This is the "project name"). For example, assume the project name is "helloworldtutorial", then the `<project>` element should be:

```
<project name="helloworldtutorial" default="all" basedir=".">
```

  This results in a jar file named *helloworldtutorial.jar* in the *env* directory.

  Note: The build system automatically adds the ".jar" extension to the filename.

- The jar is unsigned, and its classes are assigned appropriately limited permissions by the framework when they are loaded.

  See "Unsigned Jar Permissions" on page 11-193.

  Note: For information about developing signed applications, please contact Pepper Computer, Inc.

- The specified jar file(s) is included in the application's distribution directory.

  The final step is placing the jar file (or files) into the *dist* directory. You specify the jar file (or jar files) to include in the *dist* directory in the *package.ppld* file. For each jar file you want to include, you must add a `<jar>` element with an `href` attribute that names the jar file. The `<jar>` element occurs inside the `<resources>` element.

  For example, to include *helloworldtutorial.jar*, add the following (**bold**):

```
<resources>
   <jar href="helloworldtutorial.jar" />
   <jar href="data.zip"/>
   <jar href="design.zip"/>
</resources>
```

Unsigned Jar Permissions

When the application's classes are loaded by the Keeper from the unsigned jar archive, they are assigned appropriately limited permissions roughly equivalent to those of an applet, but with some modifications, as follows:

- The classes can read, write and delete files in the application's root install directory and the Keeper's *tmp* directory.
- The classes can also read certain Keeper files, such as those stored in the theme archive (*common-resources.zip* or the custom theme archive, if any).
- The classes can read the following Java system properties:

  java.version, java.vendor, java.vendor.url, java.class.version, os.name, os.version, os.arch, file.separator, path.separator, line.separator, java.specification.version, java.specification.vendor, java.specification.name, java.vm.specification.version, java.vm.specification.vendor, java.vm.specification.name, java.vm.version, java.vm.vendor, and java.vm.name.

Adding Existing Jar Files to the Build

Your application may require classes in existing jar files. Adding such existing jar files to your application is a two step process:

- Place the existing jar file in the application's *env* directory.

  Note: The *env* directory may not exist before you have built the project the first time. In this case, simply create the directory.

- Specify that the jar file is a required resource in the *package.ppld* file.

  Specify the existing jar file with the `<jar href=″jarfilename″>` element, inside the `<resources>` element. For example, to include *myJar.jar*, assuming the project's main jar file is *helloworldtutorial.jar*, as follows (**bold**):

```
<resources>
   <jar href="helloworldtutorial.jar" />
   <jar href="myJar.jar" />
   <jar href="data.zip"/>
   <jar href="design.zip"/>
</resources>
```

12

Adding and Distributing Applications

Overview

This chapter explains how to add an application to a Keeper and how to distribute applications over the web.

There are two ways to add an application to a Pepper Application Framework:

- Adding an application from a web site

  This is the typical method for small-scale distribution of completed applications.

  > **Note:** Customers interested in larger-scale distribution and update services should contact Pepper Computer, Inc.

  The application is added as a new instance and does not replace an existing instance of the application, if any.

  Framework 3.0.3 or higher is required.

- Adding an application from a local file system

  This is the typical method for adding applications during development.

  The framework must be in Debug Mode.

Distributing an application on the web

This section covers distributing your applications from a web site.

> **Note:** Support for this feature starts with Pepper Application Framework 3.0.3 and is not available on the Pepper Pad 2.

There are two aspects of this distribution method:

- Posting your applications on the web server
- Adding an application from the web site

Note: In the current release, applications are installed but are not updated using this mechanism.

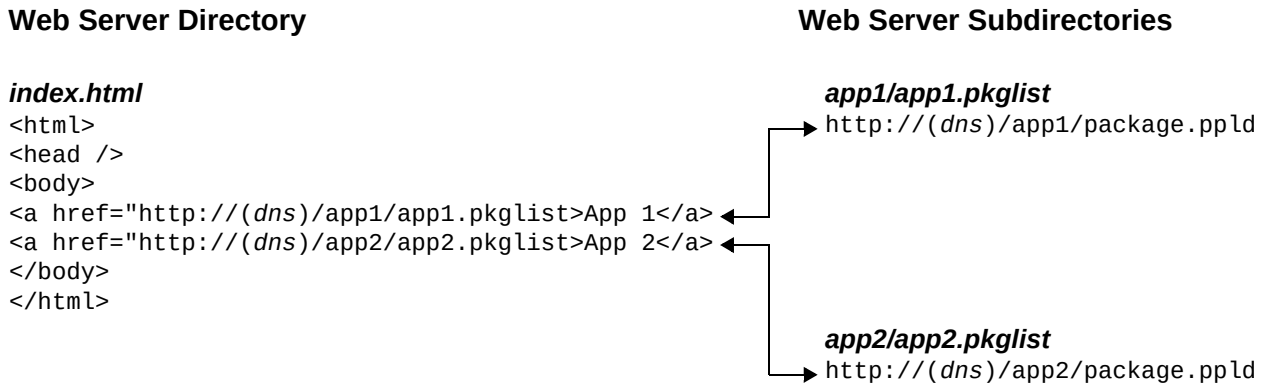
Web distribution mechanism overview

Post an application to a web server as follows:

- Place the application's distribution package in a dedicated web server directory.
- Create a file with a *pkglist* extension (for example: *app1.pkglist*) in the application distribution package directory.
- Edit the pkglist file to contain a URL to the application's *package.ppld*.
- Add a hyperlink to the pkglist file to a web page.

This approach is shown in Figure 12–1.

Figure 12–1



Posting your applications on a web server for distribution

This procedure explains how to post your applications on a web server for distribution.

Procedure:

1. Create a file on the web server with the following file extension: *pkglist*

   For example: *applications.pkglist*

   Note: This is called the *pkglist* file.

2. Create a directory on the web server for each application you are distributing.

   For this example, assume there are two applications that we'll call *application1* and *application2*.

   Therefore, create the following directories on the web server:

   (serverRoot)/application1

   (serverRoot)/application2

3. Place each application's distribution files into its directory.

The set of distribution files includes all files in the application's *dist* directory in the SDK.

For example, application1's *dist* directory is:

pepper-sdk/applications/application1/dist/

4. Add a URL for each application you want to distribute to the pkglist file.

   Each URL must specify the application's *package.ppld* file.

   In this example, the *applications.pkglist* file should have the following contents:

```
http://(dns)/application1/package.ppld
http://(dns)/application2/package.ppld
```

Procedure complete

Adding an application from a web site

This procedure explains how to add an application from a web site.

> **Note:** Pepper Application Framework 3.0.3 or higher is required; this is not supported on the Pepper Pad 2.

Procedure:

1. In any web Section, browse to the pkglist file.

   The web Section displays links for each application distributed through this pkglist file.

2. Click on the appropriate link.
3. Verify your intention to install the application as prompted.

Procedure complete

Adding a local application in Debug Mode

This section covers adding a local application to a Pepper Application Framework. This approach:

- Requires that the framework has physical access to the application's distribution package.
- Works only when the framework is in Debug Mode.
- Enables automatic refreshing of the application on application launch when any application distribution files have changed.
- Is the typical method for adding an application during development.

Making an application's files accessible to the framework

To add an application using **Debug** > **Add Application**, the framework must have physical access to the built files.

The built files a framework needs access to are those in the application's *dist* directory. (See "Application-specific directories" on page 2-6.) For example, if you want to add "myApplication" to a framework, you have to make the files in the following directory available to the framework:

pepper-sdk/applications/myApplication/dist

If the framework is on a different system than the development environment, you need to take steps to make the built files (the application distribution package) available. There are two methods for this:

- You can copy the *dist* directory to a USB thumb drive and insert the thumb drive into the Pepper device.

  See "Making files available with a USB thumb drive" on page 12-198.

- You can enable ssh on the Pepper device and use scp to copy the files in the *dist* directory to the Pepper device.

  See "Copying files to the Pepper device with ssh and scp" on page 12-199.

  Note: ssh, scp and related command line programs are not available on a Windows system by default. You can obtain them in various ways. For example you can download shareware versions from the internet. Or, you can install cygwin, which provides a Unix/Linux-like command line environment on Windows including these commands.

Making files available with a USB thumb drive

This procedure explains how to access files on a USB thumb drive from the Pepper device.

> **Note:** This procedure is useful when adding an application you developed to the Pepper device. You could also use ssh and scp; see "Copying files to the Pepper device with ssh and scp" on page 12-199.

Procedure:

1. Insert the USB thumb drive into a USB connector on the development system.
2. Copy the application's *dist* directory to the thumb drive.

   For example, if you are adding "myApplication" to the Pepper device, copy the following directory (and contents) to the thumb drive:

   pepper-sdk/applications/myApplication/dist
3. Remove the thumb drive from the development system.
4. Insert the thumb drive into the Pepper device USB connector.

   The *dist* directory (and any other files) on the USB thumb drive is now accessible to the framework.
5. Open and Xterm window on the Pepper device by pressing **ctrl** + **shift** + **1**.
6. In the Xterm window, change to the contents of the thumb drive.

   The thumb drive is mounted here:

   /media/usbhd
7. Copy the files from the thumb drive to the Pepper device with the **cp (source) (target)** command, for example:

   ```
   cp /media/usbhd/myfile.txt /opt/pepper/myCopiedFile.txt
   ```

Procedure complete.

Copying files to the Pepper device with ssh and scp

ssh is a secure shell utility that is bundled with most version of Linux and with Pepper devices. ssh enables creating a remote connection from another system (Windows or Linux) to the Pepper device. You can then use scp to copy files to the Pepper device.

> **Note:** This procedure is useful when adding an application you developed to the Pepper device. You could also use a USB thumb drive; see "Making files available with a USB thumb drive" on page 12-198.

If you want to connect with scp from a Windows system, you must install the scp software onto Windows. PuTTY is a Windows application that supports ssh and scp that is freely downloadable on the internet.

> **Note:** Selecting, installing and configuring the ssh application on the development platform is beyond the scope of this document.

Before you can use ssh, it must be enabled on the Pepper device. See "Enabling ssh on the Pepper device" on page 12-199.

After enabling ssh on the Pepper device, you can use scp to copy files to the Pepper device. See "Using scp to copy files to the Pepper device" on page 12-200

Enabling ssh on the Pepper device

This procedure explains how to enable the Pepper device for ssh, including:

Procedure:

1. Set a password for the root user account on the Pepper device as follows:
   a. Open an Xterm window on the Pepper device by pressing the **ctrl** + **shift** + **1** keys.
   b. In the Xterm window, set the root user account password with the **passwd** command, and then follow displayed instructions.

      Note: By default, the root user account on the Pepper device does not have a password set. However, ssh requires that the root user account have a password.

      Note: Creating a root user password and enabling ssh creates a security risk because it is possible that other entities could gain access to the Pepper device through ssh.

      Note: If you forget the password, you can reset it at any time by simply launching an Xterm window and using the **passwd** command.
2. Turn ssh on with the following command:

   Pepper Pad 2:

   /etc/init.d/ssh start

   All other Pepper devices:

   /etc/init.d/sshd start

   Note: It normally takes a minute or two for ssh to run the first time you launch it on the Pepper device.
3. Optionally configure ssh to always be on and available with the following command:

   Pepper Pad 2:

chkconfig ssh on

All other Pepper devices:

chkconfig sshd on

Procedure complete.

Using scp to copy files to the Pepper device

This procedure explains how to use scp to copy files from the external development system to the Pepper device.

Note: The development system must have scp. Installing and configuring scp is beyond the scope of this document.

Note: Before you can use scp to connect to the Pepper device, ssh must be enabled on the Pepper device. See "Enabling ssh on the Pepper device" on page 12-199.

Note: To use ssh, the Pepper device must be connected to Wi-Fi and have a valid IP configuration. Verify these by launching the Pepper device Web application and opening a web page.

This procedure covers the following:

- Connecting from the external system to the Pepper device with scp over ssh
- Transferring the files (typically the application's *dist* directory) to the Pepper device

Procedure:

1. Determine the Pepper device's IP address, as follows:
   a. Open an Xterm window on the Pepper device by pressing the **ctrl** + **shift** + **1** keys.
   b. Display network configuration information with the **ifconfig** command.
   c. Find the `wlan0` section of the display. It should appear as follows:

```
wlan0     Link encap:Ethernet  HWaddr 00:90:4B:23:3F:54
          inet addr:10.0.1.5  Bcast:10.0.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:21745 errors:0 dropped:15 overruns:0 frame:0
          TX packets:2289 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4483118 (4.2 Mb)  TX bytes:257332 (251.3 Kb)
          Interrupt:36
```

   d. Find the wlan0 IP address, shown in bold here:

```
inet addr:10.0.1.5  Bcast:10.0.1.255  Mask:255.255.255.0
```

2. From the Xterm window on the Pepper device, make a directory to receive the files you are about to copy, as follows:
   a. Change to the /opt directory with the **cd /opt** command.
   b. Make a directory with the **mkdir directoryname** command, where **directoryname** is the name of your new directory.
   c. Move into the new directory with the **cd directoryname** command.

3. On the external system from which the files are to be copied, open a command line window that supports the **scp** command.
4. In the scp command line window, move to the directory that contains the files you are to copy (typically the *dist* directory of the application you are adding to the framework).
5. Copy the files in the current directory to the directory you created on the Pepper device with the following command:

   scp *.* root@(Pepper device IP address):/opt/directoryname

   Where:

   - **scp** is the scp program command.

     The actual program command may vary depending on the scp installation. for example, the command for Putty is **pscp**.

   - ***.*** means copy all files in the current directory.
   - **root** specifies logging into the remote system (the Pepper device) using the root user account.
   - **(Pepper device IP address)** is the Pepper device's IP address determined previously with the **ifconfig** command and looking at the wlan0 interface.
   - **/opt/directoryname** specifies the directory on the Pepper device into which the files are to be copied.

   For example, if the command is `scp`, the Pepper device's IP address is 10.0.0.1, and the directory into which the files are to be copied is: */opt/remotedesktop*, then the command is:

   scp *.* root@10.0.0.1:/opt/remotedesktop

   Note: After executing the command you are prompted to enter the root user account password, which you set previously in "Enabling ssh on the Pepper device" on page 12-199.

6. You can confirm the files are present on the Pepper device from the Xterm window by moving to the directory and executing the **ls** command, which displays the files in the directory.

Procedure complete.

Adding a local application to the framework

Once the Pepper Application Framework has physical access to the files in the application's *dist* directory, you can add the application to the framework.

Procedure:

1. Put the framework in Debug Mode by pressing **ctrl** + **shift** + **0**

   When the framework is in Debug Mode it displays a **Debug** menu above the Tabs.

   Note: You can also launch the framework in Debug Mode. See "Debug Mode" on page 2-10.

2. Add the application as follows:

   d. Select the **Debug** > **Add Application** menu item.

   A file browser dialog displays.

e. Browse to the location where the files reside.

   If the files are on a thumb drive, browse to the following directory:

   /media/usbhd

f. Select the *package.ppld* file and click the **Open** button.

   If you are adding the Keeper application after having customized it in the SDK, select the *binder.ppld* file.

   For information using the SDK to customize the Keeper, see "Getting started with customization" on page 10-151.

At this point, the application is added to the framework and its icon displays in the **Applications** Section.

Procedure complete.

Refreshing an application in the framework

When developing an application through various stages, it is generally necessary to test each stage in the Pepper Application Framework. So, the framework needs a refreshed version of the newly built application.

The framework automatically updates applications:

- If it is in Debug Mode, and
- If it has access to the *dist* directory files from which the application was installed, and
- If any of those files have been modified.

  Note: If the framework does not have access to the *dist* directory from which the application was installed, it does not detect any new files and does not automatically update to the newer version of the application. See "Making an application's files accessible to the framework" on page 12-197.

For example, let's suppose you are developing your application on a Windows system and testing it in a framework executing on the same system. You add the application to the framework by browsing to the application's *dist* directory in the SDK. Installing the application copies all files in the *dist* directory into the installation directory. These files are used (but not modified) as required resources during execution. Each time you launch the application, the framework compares the last modified dates of its copies of the *dist* files to the last modified dates of the files in the directory from which it installed the application. If it finds that any files have been modified since installation, it copies the newer files into its installation directory.

Such update events are recorded in the framework log.

A

Sample Page Files

This appendix provides sample Page files relating to Phase Two of the Hello World Tutorial.

Worlds SectionPage sample

This is an example of the worlds SectionPage XML file.

It is generated by the build. As with all SectionPage instances generated by the build, its file name (and Section ID) is set in the `<section>` declaration in *FactoryBuild.xml*. It is updated by the framework at run time according to caching rules. Note the cached `<page>` elements (and their `<worldName>` children), each of which corresponds to a world Page. (World Pages are generated at run time). See an actual instance in your Keeper in the application's *data* directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile>
    <header>
        <pkgName>helloworld</pkgName>
        <pkgVersion>1.0</pkgVersion>
        <template>design/worlds.xsl</template>
        <noCache>0</noCache>
        <defaultPageType>world</defaultPageType>
        <createDate utc="1149701242450" />
    </header>
    <body>
        <section name="NameKey.Worlds" type="worlds" id="data/Worlds" builtin="false">
            <page id="data/1149701242380" name="" type="world">
                <worldName>mercury</worldName>
            </page>
            <page id="data/1149709198941" name="" type="world">
                <worldName>saturn</worldName>
            </page>
            <pageRemoved id="data/1149873491901" date="1149873508996" />
            <page id="data/1149873500574" name="" type="world">
                <worldName>venus</worldName>
            </page>
            <page id="data/1149873510648" name="" type="world">
                <worldName>mars</worldName>
            </page>
```

```
        </section>
    </body>
</pageFile>
```

World Page sample

This is an example of a world Page XML file.

It is generated by the framework at run time when the user creates a new world. As a result, its file name (and Page ID) is a numeric string reflecting the system time when it was created. See actual instances in your Keeper in the application's *data* directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<pageFile>
    <header>
        <pkgName>helloworld</pkgName>
        <pkgVersion>1.0</pkgVersion>
        <template>design/world.xsl</template>
        <noCache>0</noCache>
        <defaultPageType>default</defaultPageType>
        <createDate utc="1149701242380" />
    </header>
    <body>
        <page id="data/1149701242380" name="" type="world">
            <worldName>mercury</worldName>
            <worldRadius>1</worldRadius>
            <yearLength>2</yearLength>
            <dayLength>3</dayLength>
            <distanceFromSun>50000</distanceFromSun>
            <hasWater>true</hasWater>
            <hasLife>true</hasLife>
            <planVisit>2</planVisit>
        </page>
    </body>
</pageFile>
```

Sample Page Files

World Page sample

B

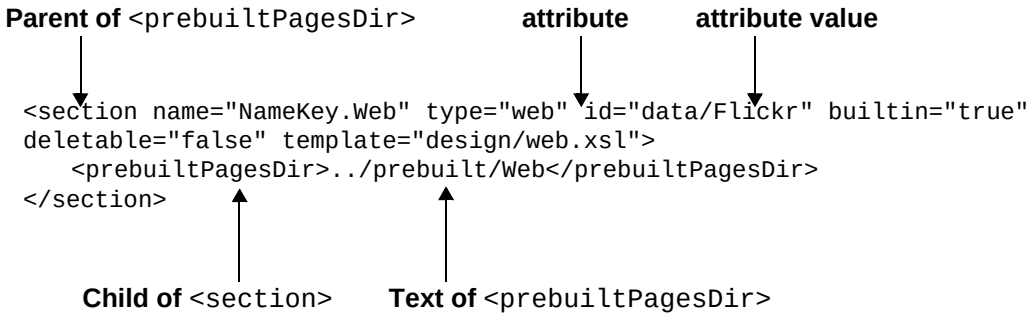
XML Reference

This chapter provides reference documentation for key framework XML files and the XML elements and attributes they contain.

XML documentation conventions

The XML reference material in this chapter uses the terms in italics shown in Figure B–1.

Figure B–1 Terms describing XML for documentation purposes



package.ppld

The root file through which applications (also known as “packages”) are added to a Keeper. You configure this file with application-specific information that enables the Keeper to add and execute the application. Such information includes:

- The application’s title
- Zip and jar resources that are needed to execute the application
- The name of the application’s required Java base class

This file must reside in the application’s *design* directory.

<jnlp>

Parent

None: root element of file.

Children

<information>, <security>, <resources>, <application-desc>

Text

None

Attribute: spec

Required. Yes

Description: Must be set to 1.0+.

Example: `<jnlp spec=”1.0+”>`

<information>

Parent

<jnlp>

Children

<title>, <packageType>, <packageGUID>, <mimetype>, <vendor>, <homepage>, <description>, <icon>, <thumbnail>, <deletable>

Text

None

Attribute

None

<title>

Parent

<information>

Children

None

Text

Required: Yes.

Description: Defines the name of the application as displayed in the Keeper.

Used by the framework (after spaces are removed, if any) with the packageGUID to determine the name of the application's installation directory.

Used by the framework (after spaces are removed, if any) with the packageGUID to create the packageID parameter that is passed to XSL files for programmatic use.

Attribute

None

<packageType>

Parent

<information>

Children

None

Text

Required: Yes

Description: Identifies to the Keeper each application type for purposes of installing and updating applications. Has particular importance when adding an application if a pre-existing application of the same type exists. Also used to communicate to an update server the currently installed applications. Also relevant when Sharing and Syncing.

When no application of the specified type is pre-installed and the application is added, it is simply installed.

The `packageType` can be set to equal the directory name of the application in the SDK in order to ensure `packageType` uniqueness, at least locally. For example, if the application's SDK directory is:

pepper-sdk/applications/HelloWorldTutorial

Then the `packageType` should be set to `HellowWorldTutorial`, as follows:

```
<packageType>HellowWorldTutorial</packageType>
```

Note: To ensure package type uniqueness you can encode your own DNS name into the application's directory name and the `packageType`.

Attribute

None

<singleton>

Parent

<information>

Children

Text

Required: No

Description: If present and set to `true`, this application can be developed to support Sharing and Synchronization with other applications of the same type on other Keepers.

Attribute

None

<packageGUID>

Parent

<information>

Children

None

Text

Required: Yes

Description: Must be set to `0`.

Attribute

None

<mimetype>

Specifies a mime type and registers this application to handle new files (for example downloaded from a Section) the specified mime type.

Parent

<information>

Children

None

Text

None

Attribute: Name

Required: Yes

Description: Specifies the mime type.

Example: `<mimeType name="image/jpeg" />`

<vendor>

Parent

<information>

Children

None

Text

Required: No

Description: Indicates the developer of the application. This is for information purposes only. The value is not accessed programmatically.

Attribute

None

<homepage>

Parent

<information>

Children

None

Text

None

Attribute: href

Required: No

Description: Optionally indicates the homepage of the application developer. This is for information purposes only. The value is not accessed programmatically.

Value: URL

Example: `www.pepper.com`

<description>

Parent

<information>

Children

None

Text

Required: No

Description: Provides the tooltip text displayed when the mouse is held over the application's icon in the Keeper's system tray.

Attribute

None

<icon>

Parent

<information>

Children

None

Text

Required: Yes

Description: The path (including filename) to a small (32 pixel by 32 pixel) image file representing this application. This icon is used to represent the application in the Keeper Flag Panel when the application is running. The image file must be stored in the *design/images* directory.

Example: `<icon>design/images/remotedesktop-32.png</icon>`

Attribute

None

<thumbnail>

Parent

<information>

Children

None

Text

Required: Yes

Description: The path (including filename) to an image (64 pixel by 64 pixel) representing this application. This icon is used to represent the application in the Keeper's **Application** Tab at all times, whether it is currently running or not. The image file must be stored in the *design/images* directory.

Example: `<thumbnail>design/images/remotedesktop-64.png</thumbnail>`

Attribute

None

<deletable>

Parent

<information>

Children

None

Text

Required: Yes

Description: Sets whether the application can be deleted by the user from the Keeper without entering Debug Mode. Since most users never enter Debug Mode, it is recommended to make the application deletable.

Values: `true` or `false`

Example: `<deletable>true</deletable>`

Attribute

None

<security>

Parent

<jnlp>

Children

<resources>

Text

None

Attribute

None

<resources>

Parent

<jnlp>

Children

<jar>

Text

None

Attribute

None

<jar>

Parent

<resources>

Children

None

Text

None

Attribute: href

Required: Yes

Description: Indicates to the Keeper the name of a zip file or a jar file required to execute this application. Most applications require this element for at least the following three files:

- *data.zip*
- *design.zip*
- The jar file that contains the required base class that extends AbstractPepperProgram

  Note: The actual jar file name created by the build is set in *build.xml*.

If your application uses classes in pre-existing jar files that must be included in the project, this element must be repeated to name the jar file and the jar file must be placed in the *env* directory. See "Adding Existing Jar Files to the Build" on page 11-194.

Example:

```
<jar href="helloworld.jar"/>
<jar href="data.zip"/>
<jar href="design.zip"/>
```

<application-desc>

Parent

<jnlp>

Children

<packageVersion>, <requiredKeeperVersion>

Text

None

Attribute: main-class

Required: Yes

Description: Identifies (by package path and class name) the application's required base class (the class that extends AbstractPepperProgram). This is the entry point for application execution.

See "AbstractPepperProgram life cycle" on page 4-29.

Example:

```
<application-desc main-class="com.pepper.HW.HelloWorld">
```

Where:

`com.pepper.HW`

Is the package path

`HelloWorld`

Is the application base class that extends AbstractPepperProgram

<packageVersion>

Parent

<application-desc>

Children

None

Text

The version of this application.

The format supports two to four numbers delimited by periods. This format is consistent with the following application revision numbering scheme:

major.minor.patch.build

For applications distributed from an update server, the version is used to determine whether the installed application needs an update.

Attribute

None

<requiredKeeperVersion>

Parent

<application-desc>

Children

None

Text

The minimum version of the Keeper required to run this package. When the application launches, its requiredKeeperVersion is compared to the actual Keeper version. If the actual Keeper version is lower than the requiredKeeperVersion, the application launch is aborted and a pop-up message displays indicating the Keeper needs to be updated.

Example:

```
<requiredKeeperVersion>3.0.3</requiredKeeperVersion>
```

Attribute

None

FactoryBuild.xml

This file declares the Section instances created during the build. Each Section instance is created based on a specified Page type as defined by a `<sectionPage>` in *PageTemplates.xml*.

This file must reside in the application's *design* directory.

<factoryBuild>

Root element of the file.

Text

None

Attributes

None

Parent

None

Children

<packageList>

<packageList>

Contains a <section> element for each of the application's Section instances.

Parent

<factoryBuild>

Children

<section>

Text

None

Attribute

None

<section>

Each `<section>` defines a Section instance to be created by the build.

The default order of the Sections in the application is the order in which they occur in *FactoryBuild.xml*, with the exception that built-in Sections are to the right.

See "Attribute: builtin" on page B-220.

Parent

<packageList>

Children

None

Text

Empty

Attribute: name

Required.

Sets text displayed on the Section's Tab.

There are two options for using this field to set the Tab text:

- Using a key to reference a key-value pair in PackageStrings.properties
- Using a literal

The recommended option is use a key to reference a key-value pair in *PackageStrings.properties* to derive the displayed text from the application's *design/PackageStrings.properties* file. This approach enables changing the displayed text, for example for language localization or for application customization, through multiple versions of the *PackageStrings.properties* file with localized file names without having to rebuild the application. To do this, use an attribute value that starts with "`NameKey.`" or "`Label.`" and that completes with a key that is unique in *PackageStrings.properties.*

For example, the following `name` value:

```
<section name="NameKey.UniqueKey"...
```

Combined with the following key-value pair in the application's *PackageStrings.properties* file:

```
NameKey.UniqueKey=Localizable Text
```

Results in Tab text: "Localizable Text"

You can also enter literal text to be displayed. For example:

```
<section name="Literal Text"...
```

Results in: "Literal Text".

Attribute: type

Required.

See “Attribute: type” on page B-223.

Attribute: id

Required.

Sets the Sections Section ID and the corresponding name of the Section’s SectionPage XML instance file (without the period followed by the “xml” extension). The value must start with “data/”. What follows “/data” is the XML file name (without the “.xml” extension).

For example, the following Section definition:

```
<section name="NameKey.Worlds" type="worlds" id="data/Worlds"
                deletable="true" builtin="false" />
```

Results in the following SectionPage XML file in data.zip:

data/Worlds.xml

Whose Section ID is:

data/Worlds

Attribute: builtin

Required. Yes

Description: Sets the position of Tabs with respect to other tabs, as follows. All Sections that you develop must have this attribute set to `false`. All ready-made Sections that are provided with the SDK (Web, Help, and Settings) have this attribute set to `true`. At program launch, Tabs are positioned from left to right in the order in which they are defined by their `section` elements in *FactoryBuild.xml*. The ready-made Sections must be defined last in *FactoryBuild.xml.* When a new Tab is created during application execution, it is created to the right of all the developer-designed Tabs (those with `builtin=”false”`) and to the left of ready-made Tabs (those with `builtin=”true”`).

Values: `true` or `false`

Example: `builtin=”false”`

Attribute: src

Required. Required for pre-built SectionPages

Description: Specifies an SDK directory that contains a pre-built SectionPage and optionally pre-built Pages.

Example:

```
<section name="NameKey.LogStatusbarJava" type="java" id="data/LogStatusbarJava"
             deletable="true" src="../prebuilt/LogStatusbarJava" />
```

Attribute: deletable

Required. yes

Description: Sets whether the Section instance is deletable by the user through tab controls.

Values: true or false

Example:

```
<section name="NameKey.LogStatusbarJava" type="java" id="data/LogStatusbarJava"
             deletable="true" src="../prebuilt/LogStatusbarJava" />
```

<prebuiltPagesDir>

Parent

<section>

Children

None

Text

Required: Required when pre-built Pages are used.

Description:

Optional, but required for Sections with a pre-built SectionPage or pre-built Pages.

Identifies a directory that contains the Section's pre-built Pages.

Directory identification is relative to the application's *design* directory.

Example:

```
<section name="NameKey.Web" type="web" id="data/Flickr" builtin="true"
                deletable="false" template="design/web.xsl">
   <prebuiltPagesDir>../prebuilt/Web</prebuiltPagesDir>
</section>
```

Attribute

None.

PageTemplates.xml

Defines the Pages (SectionPage and Pages) in the Application.

This file must reside in the application's *design* directory.

For the full set of ready-made framework Pages, include the following into *PageTemplates.xml*:

```
<xi:include href="../resources/pages/SectionWeb.xml" />
<xi:include href="../resources/pages/Bookmark.xml" />
<xi:include href="../resources/pages/Clipping.xml" />
<xi:include href="../resources/pages/SectionSettings.xml" />
<xi:include href="../resources/pages/SectionHelp.xml" />
<xi:include href="../resources/pages/SectionJava.xml" />
```

<pageTemplates>

Parent

None: root element of file.

Children

<packageName>, <packageVersion>, <sectionPage>, <basePage>

Text

None

Attribute

None

<packageName>

Parent

<pageTemplates>

Children

None

Text

The name of the package. This should exactly equal the name of the package's root directory.

Attribute

None

<packageVersion>

Parent

<pageTemplates>

Children

None

Text

The version of this package.

Attribute

None

<sectionPage>

Each Section requires a `<sectionPage>` element. The `<sectionPage>` element defines each Section's SectionPage and is used by the build and at run time to create new SectionPage instances.

Parent

<pageTemplates>

Children

<template>, <defaultPageType>, <cacheRules>, <cacheRules>

Text

None

Attribute: type

Required: Yes

There are two categories of `type` values: framework-defined and developer-defined. Framework-defined types are discussed first.

Framework-defined type of `default`

PageTemplates.xml must have a SectionPage defined with a type of `default`, or its build fails.

The definition with a type of `default` is used as the SectionPage template when the a new Section is created during the build or during application execution with an unspecified type.

For additional information, see "Tab control" on page 4-81.

Note: If you disable creation of new Tabs, the build process still requires a SectionPage definition with `type="default"` in *PageTemplates.xml*. The easy way to accomplish this is to include a ready-made SectionPage definition in *PageTemplates.xml* with the following line:

```
<xi:include href="../resources/pages/SectionDefault.xml" />
```

Framework-defined type of `web`

This type is required for Web Sections and is provided by including *SectionWeb.xml* into *PageTemplates.xml*, as follows.

```
<xi:include href="../resources/pages/SectionWeb.xml" />
```

When you create a `web` type Section, you must also ensure the ready-made *Bookmark.xml* and *Clippings.xml* files (which contain Page definitions) are included in the application's *PageTemplates.xml* file, as follows:

```
<xi:include href="../resources/pages/Bookmark.xml" />
<xi:include href="../resources/pages/Clipping.xml" />
```

Framework-defined type of `java`

This type is required for Java SectionPages and is provided by including.*SectionJava.xml* into *PageTemplates.xml*, as follows.

```
<xi:include href="../resources/pages/SectionJava.xml" />
```

Framework-defined type of `help`

This type is required for Help Pages and is provided by including *SectionHelp.xml* into *PageTemplates.xml* file, as follows:

```
<xi:include href="../resources/pages/SectionHelp.xml" />
```

Framework-defined type of `settings`

This type is required for Settings Sections and is provided by including the *SectionSettings.xml* file into the application's *PageTemplates.xml* file, as follows:

```
<xi:include href="../resources/pages/SectionSettings.xml" />
```

Developer-defined types

You can create SectionPages with any type value you choose. The only requirement is that the *PageTemplate.xml* file must include a SectionPage of the same type.

For example, the second Section of the Hello World Tutorial application has is defined in *FactoryBuild.xml* as follows:

```
<section name="NameKey.Worlds" type="worlds" id="data/Worlds" deletable="true"
builtin="false" />
```

As required, its *PageTemplates.xml* file has a `<sectionPage>` whose type is "`worlds`", as follows:

```
<!-- Define Worlds SectionPage -->
<sectionPage type="worlds">
    ...
</sectionPage>
```

Example: `<sectionPage type="default">`

<basePage>

Each type Page (non-SectionPage) requires a `<basePage>` element to define the Page.

Parent

<pageTemplates>

Children

<template>, <cacheRules>

Attribute: type

Required: Yes

Defines the type of Page created. See type attribute description for "Attribute: type" on page B-220.

<template>

Identifies the file used to display the Page. The file can be any of the following:

- A XSL transform in the *design* directory

  The specified XSL transform uses the Page as a source and generates HTML for display.
- A local HTML page or a URL to an HTML page

  The specified HTML is displayed.
- A XUL file

Parent

<sectionPage>

Children

Text

Required: Yes

Description: The path and filename from the package's root directory (but not including the root directory) to the XSL file that processes this Page. Page XSL files are always stored in the *(application)/design* directory. The text for an XSL file named "connection.xsl" would therefore be: *design/connections.xsl.*

Example: `<template>design/connections.xsl</template>`

Attribute

None

<defaultPageType>

Parent

<sectionPage>

Children

None

Text

Required: Yes

Description: Identifies the default <basePage> Page definition that is used to generate a new Page at run-time when the Page type is not specified.

Example: `<defaultPageType>connection</defaultPageType>`

Attribute

None

<cacheRules>

Used in `<sectionPage>`s to define the rules through which child Page data is automatically cached into the SectionPage.

See Caching rule syntax.

Parent

<sectionPage>

Children

<apply>, <template>

Text

None

Attribute: match

Required: yes

Description: Must be `page`

Example: `match="page"`

<apply>

Specifies the child data (elements and attributes) that are to be cached from the Page to the SectionPage.

See Caching rule syntax.

Parent

<cacheRules>

Attribute

None

<template>

See Caching rule syntax.

Parent

<apply>, <template>

Children

None

Attribute: match

An xpath statement that selects a child element or attribute.

<basePage>

Parent

<pageTemplates>

Children

<template>, <page>

Text

None

Attribute: type

Required: Yes

Description: See Attribute: type.

Example: `<basePage type=”connections”>`

<section>

Contains the Section-specific structure of element’s that together define the SectionPage data.

See Application structure from an XML perspective.

Parent

<sectionPage>

Children

Developed-defined.

Text

None.

Attribute

None

<page>

Contains the Page-specific structure of element’s that together define the Page data.

See Application structure from an XML perspective.

Parent

<basePage>

Children

Developed-defined.

Text

None.

Attribute

None

PackageStrings.properties

Contains information used to generate localizable user interface text displayed in the application. For example, Tab labels, text fields, and messages displayed on the application's status bar are derived from this file.

The labeling information in the file is stored as key-value pairs.

- Each line contains a single key-value pair.
- The key is the text to the left of the equal sign.
- Its value is the text to the right of the equal sign.

Keys are accessed programmatically and during the build process to retrieve the associated value.

Properties example

The following example shows four lines that are in the Remote Desktop application's *PackageStrings.ppld* file.

```
NameKey.PackageName=Remote Desktop
NameKey.Connections=Connections
NameKey.LogViewer=Log Viewer
NameKey.WebSection=Web
```

The *FactoryBuild.xml* file declares the Section instances. Each `<section>` element has a `name` attribute that must have one of these name-value pairs as its value.

The following example shows two sections (of several) that are declared in the *FactoryBuild.xml* file of the Remote Desktop application.

- In the first `<section>` in the following example, the `name` attribute has a value of `NameKey.Connections`, which becomes the key by which the final displayed value is looked up in the *PackageStrings.ppld* file. If the *PackageStrings.ppld* file has the contents shown above, the retrieved and displayed value is "Connections".
- In the second `<section>` in the following example, the `name` attribute has a value of `NameKey.WebSection`. This final displayed value is "Web".

```
<section name="NameKey.Connections" type="default"
                id="data/connections" deletable="false" />
<section name="NameKey.WebSection" type="web"
                id="data/web" builtin="false">
   <prebuiltPagesDir>../prebuilt/websection</prebuiltPagesDir>
</section>
```

This approach also makes it easy to build different versions of the *PackageStrings.ppld* file for each langue (localization). To do this, create language specific versions of the file and place only the correct one in the *design* directory before building the application.

C

Glossary

Table C–1 Terms

Term	Description
Action	A Java class that follows framework rules and therefore can be triggered across the LiveConnect JavaScript-to-Java Bridge. See “Java Actions” on page 4-66.
base class	Java class required for every application that extends `AbstractPepperProgram`. See “AbstractPepperProgram life cycle” on page 4-29.
Bridge	The LiveConnect bridge that enables JavaScript in Page HTML to interact with framework Java and the application Java, often through Actions. See “JavaScript and Mozilla LiveConnect” on page 4-62.
Debug Mode	A Keeper operational mode useful for application developers. Debug Mode allows you to add an application, display the current page’s HTML, XML or DOM tree, display network information, reset the Keeper, and more. See “Debug Mode” on page 2-10.
Design Mode	A Keeper operational mode useful when developing and customizing applications and when customizing the Keeper. See “Design Mode” on page 2-13.
distribution package	The set of files that constitute a built application from which the application can be installed.
Keeper	The root Pepper Application Framework application that provides the **Applications** Tab.
message catalog	Generic term for a set of properties files in which key-value pairs are used to contain text strings used in applications. Message catalogs enable write-once-use-many message definition. The application-specific message catalog is the *PackageStrings.properties* file. See “How to localize for different languages” on page 10-177.

Table C–1 Terms (continued)

Term	Description
package	Informal term for the distribution unit for Pepper applications. See “Application distribution package” on page 4-31.
Page	The unit of data organization and storage. A default type Section consists of one or more Pages (the first one is always a SectionPage). See “Sections and Pages” on page 4-32.
Pepper application	An application that runs in the Pepper Application Framework.
Pepper device	A device that runs the Pepper Application Framework. Examples include Pepper Pads, mini boxes, the Desktop, and all-in-ones.
Pepper Application Framework	The Pepper framework that provides an execution environment for Pepper applications and a graphical environment for them.
Pepper Desktop	The product name for the bundled Pepper Application Framework and on Windows.
Pepper Linux	A lightweight Linux distribution designed to support the Pepper Application Framework.
Pepper Pad	The product name of the hand-held hardware/software device that runs Pepper Linux, the Pepper Application Framework and Pepper applications.
Section	An application is divided into Sections. Sections display as tabs. There are two types: default Sections and Java Sections. See “Sections and Pages” on page 4-32.
Tab	A Section as represented in the GUI. See “Sections and Pages” on page 4-32.
theme	A bundle of files that together determine a wide range of design features, from colors and fonts to images and XSL pages. See “Custom themes” on page 10-167.

C Glossary

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

U

V

W

X