Using webMethods Mobile Designer

Version 10.3

October 2018

WEBMETHODS

This document applies to webMethods Mobile Designer Version 10.3 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: MD-UG-103-20191112

Table of Contents

About this Guide

This guide contains information about using webMethods Mobile Designer to create mobile applications for multiple device platforms.

With respect to processing of personal data according to the EU General Data Protection Regulation (GDPR), appropriate steps are documented in *webMethods Mobile Development Help*, *Managing Personal Data*.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service* , APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.

Convention	Description
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at "http://documentation.softwareag.com". The site requires credentials for Software AG's Product Support site Empower. If you do not have Empower credentials, you must use the TECHcommunity website.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to "empower@softwareag.com" with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at "https://empower.softwareag.com/".

You can find product information on the Software AG Empower Product Support website at "https://empower.softwareag.com".

To submit feature/enhancement requests, get information about product availability, and download products, go to "Products".

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the "Knowledge Center".

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at "https://empower.softwareag.com/public_directory.asp" and give us a call.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at "http://techcommunity.softwareag.com". You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 Getting Started

About Mobile Designer

webMethods Mobile Designer Overview

webMethods Mobile Designer provides a set of standardized coding abstraction layers, processes, and utilities that help you to develop mobile applications and port the applications across multiple platforms.

Mobile Designer also includes a cross compiler that you can use to compile the same source code so that it can run on a wide variety of platforms. You do not have to wait until the post-development process to apply the requirements for all mobile platforms required. You can apply cross-compilation requirements throughout the development process and launch a multi-platform solution simultaneously.

Mobile Designer Build-Time Component

The Mobile Designer build-time component is a standard build script that each project uses for each device to:

- Compile and compact the resources including data-arrays, text, images, fonts, and palettes.
- Create the parameters a specific build uses.
- Replace (hot-swap) the generic code in the underlying run-time Mobile Designer code branches with code specific to the device.
- Reference any project-specific code branches, compilation stubs, or inclusion libraries.
- Compile the code.
- Optionally obfuscate the code.
- Package, build, and sign the final binary.

Mobile Designer Cross Compiler and Run-Time Classes

In Mobile Designer you can cross compile your mobile application's Java code into C++ or Java. Mobile Designer can then compile the application code using the native tools that were installed with the target platform's SDKs. Mobile Designer then links the compiled code to the libraries installed with the SDK. When building an application for the target platform, the relevant target-platform SDKs must be installed and configured. For more information, see "Setting Up Platforms" on page 27.

You can use the run-time classes described in the *webMethods Mobile Designer Java API Reference* to provide a wide array of features found on mobile devices. To handle device-specific differences, Mobile Designer provides the classes and abstraction layers to provide a consistent base for building your application. The abstraction layers provide the ability to load images, detect interruptions, and handle text, as well as other functions, such as to establish an HTTP connection.

Parameter-Driven Projects

Mobile Designer projects use parameters and properties to simplify the process of including and/or excluding features based on a target device. Mobile Designer predefines some parameter and property values, for example, the profile settings for devices. You can override the predefined settings for a specific mobile application project and/or for a specific target device.

When creating an application, typically you have common logic that works for all target devices. However, you might require branches in the logic to address the needs of a specific target device. For example, you might need to omit or alter a feature for a target device, or you might need to position an image relative to the screen dimensions for a target device. To accommodate device-specific logic, your application logic can branch based on parameter values that are set using the device profile settings.

Properties and parameters also drive how Mobile Designer builds applications for a target device. Mobile Designer drives builds using a combination of properties, parameters, and paths, all of which you can customize and override. For example, these properties, parameters, and paths control stubs against which to compile, packagers to use when making the final binaries, details about screen dimensions, the sound APIs, and settings that are most appropriate for a target device, and other general and application-specific settings.

Configuring Mobile Designer

About Configuring Mobile Designer

You configure Mobile Designer by altering the properties located in the following file:

Mobile Designer_directory/sdk.properties

The properties in the sdk.properties file apply globally to all Mobile Designer projects. See the following for more information:

- **For instructions on how to update and add properties to the sdk.properties file**, see "Updating the sdk.properties File to Configure Mobile Designer" on page 18.
- **For a description of the configuration properties**, see "Mobile Designer Configuration Properties (sdk.properties)" on page 18.

Note: You can set properties for a project that override the settings in the sdk.properties file.

Updating the sdk.properties File to Configure Mobile Designer

The sdk.properties file is an ASCII text file that you can edit with any text editor. Update the sdk.properties file with name value pairs, using the following format:

```
property=value
```

Note: The sdk.properties file contains default values for some properties. If the value you need to specify for a property is a path, a best practice is to use the convention shown in the default unless you are working on a different operating system. For example, if a path uses c:\\a\\b\\c, it is best to specify the path using that notation rather than c:/a/b/c. Although both notations are technically correct, some third-party tools might encounter issues.

When updating Windows paths, use a forward slash character (/) or an escaped slash character (\\) in the properties file.

To add or update properties in the sdk. properties file

1. Open the sdk.properties file in your Mobile Designer installation directory using a text editor.
2. Locate the property you want to update, or add a property if it does not already exist. For more information about the properties you can specify and valid values, see "Mobile Designer Configuration Properties (sdk.properties)" on page 18.
3. **Save** and close the sdk.properties file.

Mobile Designer Configuration Properties (sdk.properties)

The properties in the sdk.properties file are settings that apply globally to all Mobile Designer projects.

JAD and Manifest Files

Use the JAD and manifest files properties to configure the vendor name and URL that Mobile Designer uses when creating project and metadata files.

Although the JAD and manifest files properties have default values, you should set values for these properties to specify information for your own organization.

Tip: You can override the JAD and manifest files properties on a project-by-project basis, such as when creating ported builds on an application for a

separate development company. To do so, specify the JAD and manifest files properties in your project's targets/_defaults_.xml file. For more information about setting project properties, see "Setting Project Properties" on page 86.

project.jad.vendor.name

Specifies the vendor name included in project and/or application metadata files that are bundled with the final binary. This is usually used as part of the data that uniquely identifies an application in an App store.

Value Vendor name.

Default `Software AG`

Example `project.jad.vendor.name=My Company`

project.jad.vendor.url

Specifies the URL to the vendor's website.

Value URL

Default `http://www.softwareag.com`

Example `project.jad.vendor.url=http://www.mycompany.com`

Java Compiler Check

When Mobile Designer executes an Ant target, the Java compiler is usually required. Use the `mobiledesigner.javac.detection.mode` property to specify whether Mobile Designer checks for the Java compiler when executing an Ant target and the action to take if the Java compiler is not present.

mobiledesigner.javac.detection.mode

Specifies whether you want Mobile Designer to check whether the Java compiler is present in a user's currently configured version of Java when the user executes a Mobile Designer Ant target. Mobile Designer checks for javac.exe on Windows or javac on Macintosh.

Value One of the following:

- `fail` if you want Mobile Designer to check for the Java compiler when a user executes an Ant target. If the Java compiler is not found, Mobile

Designer immediately stops the running Ant target with an error message.

- `warn` if you want Mobile Designer to check for the Java compiler when a user executes an Ant target. If the Java compiler is not found, Mobile Designer displays a warning message and continues to execute the Ant target.
- `none` if you want Mobile Designer to execute the Ant target without checking for the Java compiler.

Default `warn`

Example `mobiledesigner.javac.detection.mode=fail`

KZip Property

You can configure Mobile Designer to use KZip for JAR compression. When KZip is not available, Mobile Designer uses 7Zip as the default JAR packager if 7Zip is configured. If 7Zip is not configured, Mobile Designer falls back to using 'jar'.

You can download the Windows version of KZip from "Ken Silverman's Utility Page". You can download the Linux and Mac OS X version from "JonoF's Games and Stuff".

j2me.packager.kzip

Specifies the location of the KZip executable.

Value Path to the installed KZip executable.

Default `C:/Program Files/KZip/kzip.exe`

Example `j2me.packager.kzip=D:/Program Files/KZip/kzip.exe`

7Zip Property

You can configure Mobile Designer to user 7Zip for JAR compression. When 7Zip is not available, Mobile Designer uses 'jar' as the default JAR packager. You can download 7Zip for most platforms here: "http://www.7-zip.org/download.html".

j2me.packager.7zip

Specifies the location of the 7Zip executable.

Value Path to the installed 7Zip executable

Default `C:/Program Files/7-Zip/7z.exe`

Example `j2me.packager.7zip=D:/Program Files (x86)/7-Zip/7z.exe`

Localization Property

You can configure the language you want Mobile Designer to use for text in dialogs and samples. This configuration affects Mobile Designer dialogs, such as the Activate-Handset and Multi-Build dialogs.

mobiledesigner.locale

Specifies the language code.

Value One of the following language codes:

- `en` (English)
- `zh` (Chinese)
- `fr` (French)
- `de` (German)
- `iw` (Hebrew)
- `ja` (Japanese)
- `pl` (Polish)
- `ru` (Russian)
- `es` (Spanish)
- `tr` (Turkish)

Default `en`

Example `mobiledesigner.locale=de`

Proguard Obfuscator Settings

Use the Proguard obfuscator settings to provide information about your installed version of Proguard if you want to use Proguard for Android. Mobile Designer only supports obfuscation using Proguard for Android.

Note: If you have the Android SDK installed, Proguard is included in the SDK.

Mobile Designer has been tested with Proguard versions 3.7 through to 5.1.

Caution: Proguard version 4.0.1 and later includes method collapsing. If your application has collapsible methods, this might cause problems if a device has built in method size limits. If this is the case, you can change parameters to use an earlier version of Proguard or different execution parameters for that particular device.

proguard.library.root

Specifies the Proguard directory that contains the proguard.jar file.

Value Path to the Proguard directory.

Default None.

Example `proguard.library.root=C:/Proguard/lib`

obfuscator.proguard.version

Specifies the Proguard configuration and mapping you want to use.

Value Configuration and mapping from the Tools/Proguard folder.

Default One of the following:

- Proguard version 4.7 if the version supplied with the Android SDK is available
- Otherwise:
  - If building for the Android platform, Proguard version 4.3
  - If building for another platform, Proguard version 4.0.1

Example `obfuscator.proguard.version=proguard_4.7`

obfuscator.proguard.library.filename

Specifies the name of your Proguard library.

Value Proguard library name without the .jar extension.

Default None

Example `obfuscator.proguard.library.filename=proguard`

obfuscator

Enables or disables use of the obfuscator.

Value
- `proguard` if you want to use Proguard as the obfuscator.
- `none` to disable the obfuscator.

Default `none`

Example `obfuscator=proguard`

Project Build Settings

You can specify the name of the project directory to use for the output of builds.

project.build.dir.rel.root

Specifies the name of the folder that you want to use for the output of builds. This folder is relative to a project's base directory.

Value Folder name to use for the output of project builds.

Default `Builds`

Example `project.build.dir.rel.root=Output`

Proxy Settings

Configure proxy settings if you need to use a proxy server to connect to the Internet.

Note: The Software AG Installer prompts for proxy settings during installation, and the values provided at installation are saved to the sdk.properties file.

proxy.hostname

Specifies the proxy server host name.

Value Host name of the proxy server.

Default No default.

Example `proxy.hostname=proxyserver`

proxy.port

Specifies the proxy server port number.

Value Port number for the proxy server.

Default No default.

Example `proxy.port=1080`

proxy.username

Specifies the user name of a user with authority to connect to the proxy server.

Value User name

Default No default

Example `proxy.username=Administrator`

proxy.password

Specifies the password associated with the user name specified in the `proxy.username` property.

Value Password for the proxy server.

Default No default.

Example `proxy.password=secret`

Platform-Specific Properties

You must configure the location of the third-party SDKs and compilers that you want Mobile Designer to use when producing mobile application bundles. The following table lists where you can find information about the required configuration for each platform.

Note: For a list of supported SDKs, see "Supported SDK Versions" on page 28.

Platform	Description
Android	"Configuring Mobile Designer for the Android SDK" on page 30
iOS	"Configuring Mobile Designer for the iOS Platform" on page 37

Environment Variable for Mobile Designer

With previous versions, Mobile Designer created an environment variable, `MOBILE_DESIGNER`, on the computer when Mobile Designer was installed. As a result, the installation process set the value of `MOBILE_DESIGNER` to the location of the latest Mobile Designer installed instance. As of version 9.8, this environment variable is considered deprecated and is no longer created by default. However, you can continue to use it if you want. If you want to continue using this feature, you must create/alter the variable yourself in the indicated locations. If you do not want to use the `MOBILE_DESIGNER` environment variable any more, you can disable it. Then use the following when calling Ant from the command line or IDE:

```
-Denv.MOBILE_DESIGNER=Mobile Designer_dir
```

Mobile Designer uses the environment variable at run time to determine its location so that it can locate Mobile Designer-specific information, for example, the sdk.properties file.

If you have multiple Mobile Designer installations on a single machine, you can set the `MOBILE_DESIGNER` environment variable to indicate the current instance of Mobile Designer to use.

The following indicates where you can find the environment variable:

- On Windows, in **System Properties > Advanced > Environment Variables** in the **User variables** group
- On Macintosh:
  - OSX 10.7 or earlier in the ~/.MacOSX/environment.plist file
  - Later versions in /etc/launchd.conf

Preparing Linux Systems for Mobile Designer

When installing Mobile Designer on 64-bit Linux installs, the 32-bit glibc package must also be provided. Depending on your distribution, it can be installed via one of the following commands.

- For RedHat Enterprise Linux, CentOS and Fedora users:

  ```
  sudo dnf install glibc.i686
  ```

  Users with older versions of Fedora, RedHat Enterprise Linux, and CentOS may have to use `yum` in place of `dnf`.

- For SuSE Enterprise Linux, the glibc-32bit package must already be installed. This is best checked by attempting to install it. Issue the command:

  ```
  sudo zypper install glibc-32bit
  ```

  If the package is already installed, the output indicates this.

Optionally, if the user wants to use Mobile Designer without the Mobile Development perspective of Software AG Designer, it is useful to install Ant and Gradle directly.

- For RedHat Enterprise Linux, CentOS and Fedora users:

  ```
  sudo yum install ant gradle
  ```

- For SuSE Enterprise Linux:

  ```
  sudo zypper install ant gradle
  ```

Note: Some Linux distributions may not have Gradle or Ant available through the standard package repository. To install SDKMAN! and Gradle and/or Ant manually, refer to "https://gradle.org/install/".

Additionally, users can use the JDK provided by the Software AG Installer as default by issuing the following commands. For all platforms, it may be necessary to repeat this process for javac and other commands.

- For RedHat Enterprise Linux, CentOS and Fedora users:
  - `sudo alternatives --install /usr/bin/java java [SAG_INSTALL_DIR]/jvm/jvm/bin/java`
  - `sudo alternatives --config java`
  - Then select the new default.
- For SuSE Enterprise Linux:
  - `sudo alternatives --install /usr/bin/java java [SAG_INSTALL_DIR]/jvm/jvm/bin/java`
  - `sudo alternatives --config java`
  - Then select the new default.

2 Setting Up Platforms

Supported SDK Versions

SDK Versions that Mobile Designer Supports

The following table lists the SDK versions that Mobile Designer supports for each platform.

Platform SDK	Supported versions
Android SDK	5.0 through 8.1 **Note:** The minimal SDK version you must use to compile your project is 6.0 (API 23). However, you can still run your project on devices with a lower API.
iOS SDK	8.3.2 through 11.4

Setting Up the Android Platform

About Setting Up the Android Platform

Before you can use Mobile Designer to build applications for the Android platform, you must set up your environment. The following table lists required and optional tasks to set up the environment.

Setup	Required or Optional	Tasks
Development Environment	Required	Install the Android Studio IDE along with the Android SDK. For more information, see: ■ "Installing the Android Studio IDE" on page 29.

Setup	Required or Optional	Tasks
Mobile Designer Configuration	Required	Update the Mobile Designer sdk.properties file to provide information about the installed Android SDK. For more information, see "Configuring Mobile Designer for the Android SDK" on page 30.
Emulators	Optional	Define Android emulators if you want to test Android applications on emulated devices. For more information, see "Setting Up an Android Virtual Device (Emulator)" on page 31.
	Optional	Specify a proxy server to use if the Android emulator must access the Internet through a proxy server. For more information, see "Using the Android Emulator with a Proxy Server" on page 32.

The procedures in this documentation do *not* cover all possible setups and scenarios. Refer to the Android Developer website at "http://developer.android.com/index.html" for further details.

Installing the Android Studio IDE

Use this procedure to install Android Studio IDE along with Android SDK on Windows, Mac OS, or Linux so that you can use it with Mobile Designer.

To install the Android Studio IDE

1. Open the following webpage in a browser "https://developer.android.com/studio/install".
2. Follow the installation procedure for your platform.
3. After successful installation, in the Android SDK Manager, scroll down the list to **Extras** and select **Google Cloud Messaging for Android Library (Obsolete)**, **Google Play Services**, **ConstraintLayout for Android**, and **Solver for ConstraintLayout**.

Configuring Mobile Designer for the Android SDK

If you selected default location when installing Android SDK, you must configure Mobile Designer to provide information about the Android Studio installation path.

To configure Mobile Designer for the Android SDK

1. Use a text editor to open the following file:

   Mobile Designer_directory/sdk.properties

2. Locate the Android section of the file.
3. Set the values for the properties in the following table.

> **Note:** When specifying paths in the sdk.properties file, use a forward slash character ("/") or an escaped slash character ("\\") to separate folders, even when specifying Window paths.

Property and Setting
`android.studio.install.dir` Optional. Nominates the install directory for the Android Studio IDE. This will be used to find templates and scripts used to create the Android Studio project structure. Set this property only if you changed the default installation location. The defaults are: ■ For Windows: `C:/Program Files/Android/Android Studio` ■ For Mac: `/Applications/Android Studio.app/Contents`
`android.package` Set to the default package prefix that Mobile Designer uses when creating the final build for an Android device.
`android.bin.dir.root` Optional. Set to the path of the location of the default Android SDK that you want Mobile Designer to use to compile mobile applications for Android devices. Use this property only if you have to use standalone Android SDK. By default, Android SDK installed with Android Studio will be used. ■ For Windows: `C:/Users/<your username>/AppData/Local/Android/Sdk` ■ For Mac: `/Users/<your username>/Library/Android/sdk`
`android.studio.temp.build.dir`

Property and Setting	
Optional (Windows only). With Windows PCs, the combination of file and folder names created during the build process may go over the character limit allowed (approx. 240 characters). This will cause unexpected build failures. Use the `android.studio.temp.build.dir` property to nominate a folder with a short path that can be used as a temporary build directory. Use this property only if you want to change the default value. The default is `C:/mdtemp`.	

4. Optionally, if you want to override default values for a project, set the project property `project.android.sdk.version.override` to the API number you want to use.

   Note: Using an API earlier than 23 may cause failed builds.

5. Optionally configure Proguard, which is included in the Android SDK. For information about the properties you need to set to configure Proguard, see "Proguard Obfuscator Settings" on page 21.
6. Save and close the file.

Setting Up an Android Virtual Device (Emulator)

You can define Android emulators that you can use to test mobile applications.

To create an Android Virtual Device (Emulator)

1. Open the project in Android Studio, and select **Tools > Android > AVD Manager**.
2. From the AVD Manager tool, select **Tools > Manage AVDs**.

   The AVD Manager displays the virtual devices that you have defined.
3. Click **New** to create a new device.
4. Perform the following in the window for creating a new Android virtual device:
   a. In the **Name** field, type a meaningful name for the device you are adding.

      Tip: It is helpful to include the Android API level and screen size in the name.

   b. Select the API level you want to emulate.

      The selected API level determines the version of the Android operating system that runs on your virtual device, as well as default features for that device.

   c. Set the remaining settings that you want to emulate. These settings include:
      - Specifying and/or creating an SD Card image
      - Altering the screen size of the device (the "skin" section)

- Adding any extra hardware features, such as GPS

5. Click **Create AVD**.

Starting the Android Virtual Device (Emulator)

Note: If the Android emulator must access the Internet through a proxy server, you must define proxy information. See "Using the Android Emulator with a Proxy Server" on page 32.

To start an Android virtual device

1. Open the project in Android Studio, and select **Tools > Android > AVD Manager**.
2. Select the virtual device you want to start.
3. Click **Start**.

Using the Android Emulator with a Proxy Server

If the Android emulator must access the Internet through a proxy server, you must specify the proxy server to use.

Proxy Information to Provide

To provide proxy server information, use one of the following formats based on whether authentication is required for the proxy server:

- If user authentication is not required, use:

```
proxyMachineName:port
```

- If user authentication is required, use:

```
username:password@proxyMachineName:port
```

Specifying the Proxy Information

You can specify the proxy server by either providing the proxy information when starting the emulator or by defining an environment variable.

- **To specify the proxy server when starting the emulator,** start the Android Emulator using the `-http-proxy proxy` option, where *proxy* is the proxy information using one of the formats described above.
- **To specify using an environment variable,** set the `http_proxy` to the value you want to use for *proxy*, where *proxy* is the proxy information using one of the formats described above.

  The Android emulator checks the value of the `http_proxy` environment variable when it starts up and uses the value if one is defined.

Note: Launching the Android emulator with the `-verbose` flag displays the current host name and port used for the proxy.

Setting Up the iOS Platform

About Setting Up the iOS Platform

Before you can use Mobile Designer to build applications for the iOS platform, you must set up your environment. The following table lists required and optional tasks to set up the environment.

Setup	Required or Optional	Tasks
Development Environment	Required	Install the Apple Xcode IDE and iOS SDK.
	Required	Perform setup for signing applications. To deploy applications to iOS devices, the applications must be signed. For more information, see "About Signing iOS Applications" on page 35.
Mobile Designer Configuration	Optional	Update the Mobile Designer sdk.properties file to provide information about the iOS platform setup. For more information, see "Configuring Mobile Designer for the iOS Platform" on page 37.
Simulators	Optional	Use tools such as Proxifer to capture and re-direct messages sent over your Internet connect, inspect network traffic, and assist in debugging. For information about Proxifier, see "http://proxifier.com".

The procedures in this documentation do *not* cover all possible setups and scenarios. Refer to the Apple Developer website at "https://developer.apple.com/devcenter/ios/index.action" for further details.

Installing the Apple Xcode IDE

Mobile Designer supports Xcode 7.3.1 and later. For information about supported iOS SDK versions, see "SDK Versions that Mobile Designer Supports" on page 28.

Note: The iOS SDK is included as part of the Xcode installation. Be aware of the following:

- Use of Xcode 7.3.1 requires OS 10.11.5 El Capitan.
- Developing and testing applications that run on iOS 9 requires Xcode 7.
- Developing and testing applications that run on iOS 10 requires Xcode 8.
- Developing and testing applications that run on iOS 11 requires Xcode 9.

This section provides only the minimal information for installing the Apple Xcode IDE so that you can use it with Mobile Designer. For additional details, refer to information about the Apple Xcode IDE on the Apple Developer website at "https://developer.apple.com/xcode".

To install the Xcode IDE from the Mac App Store

1. Ensure that your environment meets the requirements for the Xcode IDE.

   You can find the requirements at "https://developer.apple.com/support/ios/ios-dev-center.html"

2. On your Macintosh, open the Mac App Store.
3. Search for Xcode.

   Note: You can find Xcode in iTunes at: "https://itunes.apple.com/us/app/xcode/id497799835?mt=12". From iTunes, use the link to view Xcode in the Mac App Store.

4. Install Xcode from the Mac App Store.

   Note: You might need to sign in with a user account that has membership in the iOS development program.

5. After installation, open Xcode.
6. On the System Component Installation screen, install **Device Support**.
7. When the installation completes, click **Start Using Xcode**.
8. Select **Xcode > Preferences**.
9. Select the **Downloads** tab.
10. Next to **Command Line Tools**, click **Install**.

About Signing iOS Applications

To deploy applications to iOS devices, the applications must be signed. During development, you can sign applications using a development certificate that is limited to a small set of devices. To deploy to the App Store, you must sign the application with a distribution certificate.

If you develop on a Macintosh for iOS, you can use your existing signing environment on the same machine, or you can import an existing environment to another machine. If you do not have an existing signing environment, you must create a signing environment.

Using an Existing Signing Environment

If you are already developing on a Macintosh for iOS and intend to use the same machine for developing iOS applications with Mobile Designer, you can use your existing signing environment.

To use an existing signing environment on the same machine

1. Locate your .mobileprovision files and note the path to the following so that you will have it available to configure Mobile Designer.

   - Ad-hoc profile you intend to use for development
   - Distribution profile, if you have one

   You will need the path to the .mobileprovision files when you configure project-specific properties for signing, such as `ios.provisioning.profile.appstore` and `ios.provisioning.profile.adhoc`. For more information, see "Setting Properties to Build for an iOS Device" on page 118.

2. Export your existing certificates using one of the following methods:

   - If you use Xcode to automatically manage your certificates, you can export your existing certificates using the Xcode Organizer.
   - Download your certificates from the iOS provisioning portal, at "https://developer.apple.com/ios/manage/overview/index.action".

Importing the Signing Environment from Another Macintosh

To use your existing signing setup on another Macintosh, you export signing information from an existing Macintosh and copy it to the Macintosh where you want to develop iOS applications with Mobile Designer.

To import an existing signing environment to another machine

1. On the Macintosh with signing environment you want to use, start the Keychain Access application, which is in the Applications/Utilities folder.
2. In the Keychain Access application, in the **Keychains** panel, ensure that **login** is selected.
3. In the **Category** panel, select **Certificates**.
4. Make a note of the following information:
   - Full name of your iPhone Developer certificate. The full name is typically:

     iPhone Developer: *Firstname Lastname*
   - Full name of your iPhone Distribution certificate. The full name is typically:

     iPhone Distribution: *CompanyName*
5. Export the private key associated with the developer certificate:
   a. In the **Category** panel, select **Keys**.
   b. Select the private key that is associated with the developer certificate. Select **File > Export Items**.
   c. When prompted, create a password for exporting. You will need to supply this password when importing the private key to the target Macintosh.

      Important: Do not use the password you use to login to your Macintosh.

   d. When saving the private key, be sure to save in **Personal Information Exchange (p12)** format.

   Important: You should keep a backup copy of your private key by copying the key to removable media and storing it somewhere safe. If you lose your private key, for example, due to a hardware failure, you will not be able to deploy applications. This is particularly important when you want to update old versions of an application submitted to the App Store. Apple does not keep information about your private key.
6. Move the exported p12 key file to the Macintosh where you want to develop iOS applications with Mobile Designer.
   a. Copy the p12 key file to the target Macintosh and save in any location.
   b. Double-click the p12 key file to begin the key import process.
   c. When prompted for a password, supply the password you created when exporting the private key.

   The private key and required certificates are imported into the target Macintosh
7. Download the appropriate .mobileprovision files for ad-hoc and distribution from the developer portal, "https://developer.apple.com/ios/manage/provisioningprofiles/

index.action", or copy the. mobileprovision files from your existing environment to an appropriate location from the target Macintosh.

Creating a New Signing Environment

If you do not have a signing environment, you must set up your environment. You can find setup instructions on the iOS Provisioning Portal at "https://developer.apple.com/ios/manage/overview/index.action".

The setup steps include the following:

- Create and install iOS development certificates
- Nominate device IDs for development
- Nominate an App ID
- Create a Development Provisioning Profile

Note: When your environment is ready to distribute iOS applications as Ad-Hoc or App Store builds, you must create the appropriate .mobileprovision files. You configure Mobile Designer to specify where the .mobileprovision files are located.

Configuring Mobile Designer for the iOS Platform

After installing the Apple Xcode IDE and setting up your environment for signing iOS applications, you can configure Mobile Designer to provide information about the iOS platform setup.

To configure Mobile Designer for the iOS platform

1. Use a text editor to open the following file:

   Mobile Designer_directory/sdk.properties

2. Locate the iOS section of the file.
3. Set the values for the properties in the following table.

Property and Setting
`ios.bundle` Set to the prefix to use for the CFBundleIdentifier. The CFBundleIdentifier is a unique identifier for your application bundle. For the prefix, it is recommended that you use your company's domain name, with each portion in reverse order. For example, for the domain name mycompany.com, the recommended identifier is "com.mycompany". An example of setting the property for this identifier is:

Property and Setting
`ios.bundle=com.mycompany.`

Note: It is important to include a trailing period to act as a separator when specifying the `ios.bundle` property. Mobile Designer appends your application's name directly to the `ios.bundle` value to create your application's unique CFBundleIdentifier. For example, for an application named “MyApp”, the name is "com.mycompany.MyApp".

Note: If you have several Xcode versions installed on your system and want to use a specific one, run the following command in the terminal:

```
sudo xcode-select -s path_to_Xcode.app/contents/Develop
```

4. Save and close the file.

3 Creating Mobile Application Projects

Setting Up a Mobile Application Project

About Mobile Application Projects

You set up a mobile application project for each mobile application you want to develop. The project contains the application code, defines the devices you want the application to support, and references all the resources that the project requires.

Perform the following actions to set up a mobile application project:

- Create the mobile application project as described in *webMethods Mobile Development Help*, *Creating a New Mobile Project*.
- Code your application using Java, specifically J2ME. Mobile Designer provides several run-time classes that you can use in your application. For more information, see "Coding a Mobile Application" on page 44 and *webMethods Mobile Designer Java API Reference*.
- Software AG recommend that you use the default UniversalResHandler implementation that Mobile Development provides as a resource handler. The resouce handler identifies the resources that your project requires. You can still code your own resource handler though this is deprecated and not adviced. For more information, see "Defining Resources for a Mobile Application Project" on page 73.
- Set properties for your project. Although there are numerous properties you can define for a project, Mobile Designer provides initial settings and/or defaults for most. However, there are a few properties you must set for your project. For more information, see "Setting Project Properties" on page 86.
- Set parameters for your project. Parameters contain settings about devices and resources. Additionally, you can define your own application-specific parameters. In your application code, you use parameters to perform such tasks as loading resources or branching the logic based on parameter values to address the needs of specific target devices. For more information, see "Where You Define Parameters" on page 87.
- Mobile Development will add devices that you want your application to support to your project when you define launch configurations. For more information, see *webMethods Mobile Development Help*, *Building a Mobile Project*. However, you can continue to add devices using Ant targets as described in "Adding Devices to a Mobile Application Project" on page 91.

Sample Projects Provided with Mobile Designer

Mobile Designer comes with the sample projects described in this section. The sample projects are located in the following directory:

Mobile Designer_directory/Samples

Expense Tracker

The Expense Tracker sample project uses many NativeUI objects. It demonstrates how to solve design and implementation difficulties when developing mobile applications. It features user interface conventions that are common requirements for mobile applications, such as dynamic list population and display; data entry, storage and reporting mechanisms; handling for multiple device platforms and form factors.

Library JSON

The Library JSON sample project is an example of a library project. It demonstrates how you can precompile parts of your codebase into separate libraries that you can then reference in another project. Specifically, this project uses third-party JSON Java code. For more information about creating and using libraries, see "Linking to External (3rd Party) Native Libraries/Frameworks" on page 68.

You compile the project for the target platforms for which you want to use the library. A library project's build.xml references libtargets.xml rather than targets.xml. The target libraries for this sample project are J2ME (for Phoney), Android, and iOS. To use this sample project, you can import it, then cross-compile and build the library using the +Library-Build Ant task.

Note: For a sample of a project that references this library, see the NativeUI JSON sample project.

NativeUI Asset Catalog

The NativeUI Asset Catalog application demonstrates the use of a simple Asset Catalog setup. A single image, "TheImage.png", is included at various resolutions for Android and iOS devices.

NativeUI Contacts

The NativeUI Contacts application demonstrates the use of the Personal Information Management (PIM) APIs defined for JSR 75.

The sample demonstrates using the PIM APIs to:

- Read information for contacts that already exist in a device's address book. This can be done for devices running on any platform.
- Edit existing contacts in a device's address book and adding new contacts to a device's address book. This can be done for devices running on platforms that support editing the address book.

NativeUI Credential Store

The Native UI Credential Store application demonstrates the use of the APIs for the CredentialStore and IsolatedKeyValueStore classes. It is possible to load and store mappings for server URLs to user name/password credentials, and more generic key/value settings that may need to be held in isolated storage.

Note: For this sample application, no consideration is made to the underlying security model that backs up these classes. Care should be taken to ensure that the implementation matches any storage requirements for this class of data.

NativeUI Database

The NativeUI Database sample project demonstrates the use of databases on supported platforms.

NativeUI Demo

The NativeUI Demo application demonstrates the use of all the major native user interface (NativeUI) classes in Mobile Designer.

The sample also demonstrates how to support tablet devices. It contains code to determine whether it is running on a tablet based on the screen size of the device. When running on a tablet, the application uses multiple panes in the user interface. For more information about using panes, see *webMethods Mobile Designer Native User Interface Reference*.

Mobile Designer provides three versions of the NativeUI Demo.

- _NativeUIDemo_ was hand coded using Mobile Designer functionality and uses the classic run time.
- _NativeUIDemoNew_ was hand coded using Mobile Designer functionality and uses the new run time. It showcases some additional features only available in the new run time.
- _NativeUIDemoX_ was created using Mobile Development.

  This version of the sample illustrates how to use Mobile Development to create a mobile application. In this sample, many of the user interface objects were added explicitly in the Outline Editor. For example, a button was added using the Mobile DevelopmentButton object. However, some objects were defined using the dynamic objects that Mobile Development offers, for example, the DynamicDisplayObject object. The dynamic objects were used to illustrate how to use dynamic objects and how to provide user code for dynamic objects.

  Mobile Development provides a default resource handler that handles most resources. However, this project also uses a custom resource handler to illustrate

handling cases when custom code/dynamic objects require resources that the default resource handler cannot accommodate.

Note: To use the _NativeUIDemoX_ version of the sample, you need to import the sample project into Software AG Designer. The _NativeUIDemoX_ sample project includes the information for the model. It also includes user logic in the src folder (that is, the user space). However, before you can use the project, you must use the Mobile Development **Generate Source Code > Application Model and API** command to generate sources for the project. This command generates the logic to execute the model in the gen/src folder and the Mobile Development API in the gen/api-src folder. For more information about generating sources, see *webMethods Mobile Development Help*.

NativeUI Exercise

The NativeUI Exercise sample project shows the process of creating a simple native user interface voting application, complete with model answers for each of the steps.

NativeUI Hello World

The NativeUI Hello World sample project contains the bare minimum needed to display some text and transition between two views. You can copy this project to assist in learning the native user interface classes in Mobile Designer.

NativeUI JavaScript Map

The NativeUI JavaScript Map sample project gives an example of how to send simple messages into and out of a nUIWebView. The project is designed to be used with Mobile Development. The important code resides in the MyMapViewControllerImpl class.

NativeUI JSON

The NativeUI JSON sample project shows the interaction with a JavaScript Object Notation (JSON)-based server, fetch, and display data.

Note: This sample references the library created using the Library JSON sample project.

NativeUI Location

The NativeUI Location sample project shows the use of the Location API with native user interface classes to display the user's current location.

NativeUI My Graphical Element

The NativeUI My Graphical Element project demonstrates the use of a custom user-created native user interface element to draw a chart.

NativeUI My Native Element

The NativeUI My Native Element project demonstrates how to add platform-specific native code to create a new custom visual component that works along with the NativeUI objects that Mobile Designer provides.

NativeUI PDF Demo

The NativeUI PDF Demo sample project demonstrates the use of native platform code injection to display a PDF.

NativeUI Push Notifications

The NativeUI Push Notifications sample project demonstrates the use of push notifications on supported platforms.

NativeUI SOAP

The NativeUI Soap sample project uses SOAP to communicate with a remote server, fetch data, and display it on the device.

Coding a Mobile Application

Migrating to the New Run-Time Classes

If you use projects that were built with a version prior to 10.1, you must migrate these projects to use the new run-time classes provided with version 10.1.

Enabling the New Run-Time Classes

To enable the new run-time classes

1. Enable the new run time by adding the following line to the _defaults_.xml:

```
<property name="project.runtime.uses.nativeui.phoney" value="true"/>
```

Note: This step is not required if you use Mobile Development.

2. Re-activate the project to make the new run-time classes available. (Ignore compile errors.)
3. Refresh the project in Software AG Designer.

Refactoring MyApplication

To refactor the MyApplication class

1. Change the base class of the MyApplication class from Application to MDApplication.
2. Add `implements IMDApplicationListener` to the class declaration, and implement all methods.
3. Remove setAppCanvas() method. The project specific initialization logic can be added to the init() method. For detailed information, see "Binding MyCanvas with MyApplication" on page 48.

Refactoring MyCanvas

To refactor the MyCanvas class

1. To support previous event handling, remove `extends CanvasNativeUI` from the class declaration and add `implements nUIEventListener`.
2. In the sizeChanged() method, remove the first line, super.sizeChanged().

Refactoring Access to Handlers

The instance field was removed from all handlers. Every direct access to the instance field must be replaced with the getInstance() method.

To refactor access to handlers

1. Replace AudioHandler.instance with AudioHandler.getInstance().
2. Replace CameraHandler.instance with CameraHandler.getInstance().
3. Replace ImageHandler.instance with ImageHandler.getInstance().
4. Replace TextHandler.instance with TextHandler.getInstance().
5. Replace ResourceHandler.instance with ResourceHandler.getInstance().

Using CameraHandler in the New Run Time

The CameraHandler class at com.softwareag.mobile.runtime.media is deprecated from version 10.3 and is no longer recommended for use in applications. The new API to use is in com.softwareag.mobile.md.media and consists of the ImagePicker class and an IImagePickerCallback interface.

Using the ImagePicker Class

The ImagePicker exposes two public static methods for general use, pickFromCamera(IImagePickerCallback callback) and pickFromGallery(IImagePickerCallback callback).

Note: Activating the camera is a long-running task that can consume a lot of resources. Because of this, the mobile operating system may background or close applications. This includes the application that opened the camera in the first place. When control returns to the application, it will be via the IImagePickerCallback. Be aware that the application may not be in exactly the same state as before.

You should create an implementation of IImagePickerCallback for yourself. When the mobile operating system returns control to the application, either onImagePicked() or onImagePickingFailed() will be called. If the image picking has failed, the reason why will be indicated as an int value (currently one of error, user cancelled, or no permission), and a String with optional extra information that the operating system or the ImagePicker may provide. If no extra information is available, the String may be empty or null.

To create an IImagePickerCallback implementation, use the following code:

- **In a method**

```
ImagePicker.pickFromCamera(new MyImagePicker());
```

- **In a separate class or as an inner class**

```
class MyImagePicker
{
  public onImagePicked(Image img)
  {
    System.out.println("Got an Image, " + img.getWidth() + "x"
       + img.getHeight());
    //do something with the Image...
  }
  public onImagePickingFailed(int result, String error)
  {
    switch(result)
    {
      case STATUS_ERROR:
      System.out.println("An unexpected error occured. Further info :"
        + error);
      break;
      case STATUS_CANCELLED:
      System.out.println("The user closed the camera without taking a picture.");
      break;
      case STATUS_NO_PERMISSION:
      System.out.println(
         "The user / OS did not grant permission to use
       the camera in this app, Further info : " + error);
      break;
    }
  }
}
```

Capturing Barcodes Using ImagePicker

You can use barcodes or QR-codes as a convenient way to get long URLs, configuration data, product codes, etc. into an application. In order to support this, the ImagePicker harnesses the native capabilities of devices to detect barcodes in images.

Note: For iOS, iOS 11 or greater is required.

Barcodes may be detected in images already held in the device's gallery, using live video from the camera, or using static images captured by the camera. Barcode values detected in the image are returned to a class conforming to IBarcodePickerCallback.

```
if(use_camera)
{
   ImagePicker.scanBarcodeFromCamera(new MyBarcodeCallback());
}
else
{
   ImagePicker.scanBarcodeFromGallery(new MyBarcodeCallback());
}
```

The classMyBarcodeCallback can then be implemented as follows. Note that there may be more than one barcode in the image, so onBarcode() returns an array of String values.

```
class MyBarcodeCallback implements IBarcodePickerCallback
{
 public void onBarcode(String[] barcodes)
 {
  System.out.println("Got " + barcodes.length + " barcodes.");
  for (int i = 0; i < barcodes.length; i++)
  {
   System.out.println("You scanned : " + barcodes[i]);
  }
 }
 public void onBarcodeFailed(int result, String error)
 {
  System.out.println("Barcode scanning failed, error code :
                " + result + ", with message : " + error);
       }
}
```

Note: As with other methods in ImagePicker, it is possible that the mobile OS may have backgrounded or closed and re-started your application during the image selection process. This is important to bear in mind when your callback returns.

Furthermore, you can limit the types of barcodes detected using the following code:

```
//No 2D barcodes, please!
ImagePicker.scanBarcodeFromGallery(new MyBarcodeCallback(),
  ImagePicker.BARCODE_TYPE_LINEAR_ALL);
```

Live Barcode Scanning

When scanning a barcode directly from live video feed, there are a few extra considerations. The live scanning methods also take a frame rate in their arguments. Devices pick a value that can be supported by the hardware that most closely matches your required frame rate. The frame rate should be carefully chosen. If the value is too

high, this can cause the device to work too hard, draining the battery and slowing down the UI. Similarly, if the frame rate is too slow, it gives a poor experience for the user.

Tip: Most handsets perform well with an update rate somewhere between 15-20fps.

When scanning live video, your IBarcodePickerCallback must also implement wouldAcceptBarcode(). For each frame in which valid barcodes are detected, the callback is asked if this frame's barcodes should stop the scanning process. Code in wouldAcceptBarcode() must be as performant as possible, and must make no changes to the application state. The onBarcode() method is called as normal after wouldAcceptBarcode() signalled that the barcodes have been accepted.

```
public boolean wouldAcceptBarcode(String[] barcodes)
{
   if(barcodes != null && barcodes.length > 0)
   {
  for(int i = 0; i < barcodes.length; i++)
  {
   if(barcodes[i] != null && barcodes[i].startsWith("2345"))
   {
    return true; //This is our product, handle actions in onBarcode()
   }
  }
   }
   return false; //reject barcodes, keep scanning
}
```

Binding MyCanvas with MyApplication

To bind MyCanvas with MyApplication

1. Add the new private field `private MyCanvas canvas` to the MyApplication class.
2. Override the init() method.
   a. Call super.init() to initialize Mobile Designer default specific bits.
   b. Set application listener to receive `Application` events.
   c. Create a new MyCanvas instance.
   d. Register MyCanvas to receive all UI events from nUIController by calling nUIController.addEventListener(canvas, false).
3. Forward the information of the new size to the canvas instance in the onSizeChanged method.
4. Override the getMainWindow() method, and pass a new main window back.

   MyApplication should now look like this :

```
public class MyApplication extends MDApplication implements
                    IMDApplicationListener
                    {
                    private MyCanvas canvas;
                    public void init() {
                    super.init();
```

```
// register for application events
setApplicationListener(this);
// create MyCanvas instance
canvas = new MyCanvas();
// register canvas to receive all UI events.
nUIController.addEventListener(canvas, false);
}
public nUIWindowDisplay getMainWindow() {
// return a new main window instance
return canvas.onCreateMainWindow();
}
public void onSizeChanged(int width, int height)
{
// forward to the canvas instance.
canvas.sizeChanged(width, height);
}
/// .... other methods
```

Replacing Deprecated or Removed Methods and Fields with New API

To replace deprecated or removed methods and fields

1. Replace `CanvasDimensions.CURRENT_SCREEN_WIDTH` with nUIController.getScreenWidth().
2. Replace `CanvasDimensions.CURRENT_SCREEN_HEIGHT` with nUIController.getScreenHeight().
3. Replace `CanvasDimensions.CURRENT_SCREEN_PPI` with nUIController.getScreenPPI().
4. Replace Canvas.getWidth() with nUIController.getScreenWidth().
5. Replace Canvas.getHeight() with nUIController.getScreenHeight().

Additional Steps

The following additional steps must be taken:

To finish migration

1. Optimize the import in all classes to ensure there are no references to the old run time anymore.
2. Access the application instance from code by calling the com.softwareag.mobile.runtime.RuntimeInfo.getApplication() method.

Mobile Designer-Provided Run-Time Classes

webMethods Mobile Designer contains many run-time classes that provide an array of features that you can use in your application. For detailed information on run-time classes, see *webMethods Mobile Designer Java API Reference*.

Application and Parameter Classes

com.softwareag.mobile.runtime.core.MDApplication

The MDApplication class contains the minimal functionality to start an application. You should use it in all projects and refactor old projects which use the deprecated Application class. For detailed information on migration, see "Migrating to the New Run-Time Classes" on page 44. The MDApplication class is the application's entry point. You must subclass it to start an application. You can override the init() method to initialize some resources or variables. This method is called only once just after the MDApplication instance was created. If you override this method, you must call super.init() to initialize internal bits by MDApplication.

To provide your own content, you must override the getMainWindow() method and return a new nUIWindowDisplay object with the views you want to show.

In contrast to the deprecated run time, the new run time does not spawn any additional thread. Instead, it uses a native main thread in which the particular application was started. All Mobile Designer callbacks are called on this main thread. You should return as fast as possible to avoid UI freezes. All CPU-intensive tasks or tasks that wait for response should be executed asynchronously. To run tasks asynchronously, you can use runAsync methods provided in the MDApplication class.

The MDApplication class resides in the nativeUI library, and it assumes that nativeUI library is used in the project.

com.softwareag.mobile.runtime.Parameters

When Mobile Designer runs the resource handler that you create for your project, it automatically creates the Parameters static class, which contains the parameters that drive the Mobile Designer run-time code for a particular build. The Parameters class, com.softwareag.mobile.runtime.Parameters, contains information about the devices and project-specific parameters, such as resource, resource block, and text IDs.

Mobile Designer uses the following naming conventions for the parameter names:

- `PARAM_MD_***` defines parameters controlling Mobile Designer run-time source code functionality
- `PARAM_***` defines application-specific parameters
- `RESBLOCKID_***` defines identifiers for all the resource blocks
- `RESID_***` defines identifiers for all the individual resources
- `TEXTID_***` defines identifiers for all the lines of text
- `MENUID_***` defines identifiers for all the menus

When the remaining run-time classes are not used in a project, you can generate the Parameters class by reference in third-party code.

Run-Time Comms Classes

com.softwareag.mobile.runtime.comms.MessageConnectionHandler

Use the MessageConnectionHandler class to control the detection of incoming SMS messages using the Wireless Messaging API, to send SMS messages to other phones, and to handle Push Notifications.

Mobile Designer sets the default value for each device. You can override the value Mobile Designer sets setting the mobiledesigner.runtime.core.class.comms.messageconnection project property to `none` or `wma`.

Note: The MessageConnectionHandler implements the MessageListener class.

Note: The httpstream variant of HttpConnectionHandler allows HTTP DELETE messages to specify an optional POST body.

Run-Time Database Classes

Mobile platforms can support a database. For example, many platforms support the SQLite database. Mobile Designer provides the com.softwareag.mobile.runtime.database library that contains classes and methods you can use in your mobile applications to execute SQL statements. Using the database library allows your mobile application to access database information. The database library provides a uniform manner for using a database independent of the target platform and database system. For more information about the database library, see the *webMethods Mobile Designer Java API Reference*.

Note: Support for the database library is provided only for the following platforms:

- Android
- iOS
- Phoney

To use the database library in a mobile application, the project.handset.uses.Database project property must be set to `true` (default).

For an example that illustrates how to use the database library, see the NativeUI Database sample.

Important: On most platforms, the Cursor class is a buffered cursor, which stores all query results in memory. Use caution when querying database tables that contain data that uses the blob data type, for example, images.

Run-Time Media Classes

com.softwareag.mobile.runtime.media.AudioHandler

Use the AudioHandler class to control the sound and vibration functionality when writing for the main common audio libraries available on mobile devices. The media classes try to keep the phone's backlight on when possible for the selected device.

The Mobile DesignerAudioHandler class sets the value for each device.

com.softwareag.mobile.runtime.media.CameraHandler

Use the CameraHandler class to initialize a device's camera, take snapshots, and stop the camera.

Mobile Designer sets the values for the CameraHandler class for each device. You can override the value Mobile Designer sets using the mobiledesigner.runtime.core.class.camera project property.

com.softwareag.mobile.runtime.media.ImageBase

The ImageBase class provides the base level of the image creation and drawing functionality, including:

- Decoders for the various image encoding methods that are part of the Mobile Designer resource handler
- Image caching used with some devices that have problems freeing images from memory
- Multi-celled image support, stored as individual images at run time, or as a single image which is clipped and drawn
- Transformations for all devices
- Image dimension information to support any multi-cell images or transformations

You can access the ImageBase functionality through the ImageHandler class.

The `DRAWIMAGETRANSFORM_***` values and the `LOADIMAGETRANSFORM_***` values are not interchangeable. The `LOADIMAGETRANSFORM_***` values are used to cache multiple images at load time. For more information, see the *webMethods Mobile Designer Java API Reference*.

com.softwareag.mobile.runtime.media.ImageHandler

Use the ImageHandler class to load, draw, and manage images. Mobile Designer sets the values for the ImageHandler class for each device.

com.softwareag.mobile.runtime.media.PngParser

The PngParser class provides a PNG-encoding method that can create an image from pixel and palette data. In Mobile Designer, you can create the image dynamically at run time, or use image compression methods that exceed the default PNG format.

When opaque mutable images render faster on a device than immutable images, the PngParser makes adjustments to improve performance when rendering the image.

You can generate palettized (PNG-8) and true-color (PNG-24) PNG files using the PngParser methods when full pixel or palettized information is provided.

com.softwareag.mobile.runtime.media.TextHandler

The Mobile DesignerTextHandler class provides the following functionality:

- Loads text files
- Supports multi-languages
- Stores hyphenation guidelines for text splitting
- Dynamically splits text based on hyphenation guidelines
- Injects text
- Supports system and bitmap font
- Draws proportional and fixed width font in all systems
- Handles all the expected metric queries such as font height, Character width, and String width

Run-Time Serializer Class

com.softwareag.mobile.runtime.serialize.Serializer

The Serializer class provides the ability to serialize a Java class into a binary stream and to deserialize a binary stream back to a given Java class.

To enable a class to be serialized, you must ensure it implements the com.softwareag.mobile.runtime.serialize.Serialzable interface (not the J2SE java.io.Serializable interface). A class that implements the Serializable interface can then be passed to the Serializer class for serialization and deserialization.

Mobile Designer sets the values for the Serializer class for each device. You can override the value Mobile Designer sets by setting the mobiledesigner.runtime.core.class.serialize project property to `cldc11`.

Run-Time Storage Classes

com.softwareag.mobile.runtime.storage.RecordStoreHandler

Use the Mobile DesignerRecordStoreHandler class to control saving data in the application's record store. The RecordStoreHandler class also determines whether saving the data is performed immediately or is cached and saved on termination. Use the cache and save option with slow devices.

For access to the record store, call the derivatives of setRecordStoreData and getRecordStoreData.

com.softwareag.mobile.runtime.storage.ResourceDataTypes

The ResourceDataTypes class contains a set of helper methods to retrieve primitive data types from your resources with the run-time ResourceHandler class. Two variants are present that enable including the float and double methods with Connected Limited Device Configuration (CLDC) 1.1 devices.

Mobile Designer sets the ResourceDataTypes value for each device. You can override the value Mobile Designer sets by setting the mobiledesigner.runtime.core.class.datatypes project property to `cldc11`.

com.softwareag.mobile.runtime.storage.ResourceHandler

Use the ResourceHandler class to manage the loading and caching of resources, resource packs, and resource blocks.

Depending on parameter settings, devices can:

- Cache the entire resource packs
- Cache individual blocks
- Load only the resources that are individually stored in the application bundle

You must use clean-up methods to ensure that the mobile application handles memory management in the most appropriate way for all devices.

Run-Time Utility Classes

com.softwareag.mobile.runtime.utility.Maths

The Maths class is a fixed point math library that contains methods you can use in your applications for conversion to and from a fixed-point number, trigonometric functions, square and cube roots methods, and a random method. The random method is included for instances when your application relies on the random method returning the same value across all devices. Built-in JVM implementations can differ from one device to another in their number handling. As a result, if you are porting your code to anything other than Java, you could encounter differences in the output produced by the random function.

The parameter `PARAM_MD_MATHS_FP_SHIFT` controls the accuracy of the math performed, with a fixed point value of 1 equal to `1 << PARAM_MD_MATHS_FP_SHIFT`.

The Maths class is based on Connected Limited Device Configuration (CLDC) 1.1. The Maths class does not contain references to the primitive types float or double.

com.softwareag.mobile.runtime.utility.PlatformRequest

Use the PlatformRequest class to launch the browser on devices that support it.

Image Caching

Mobile Designer supports caching of javax.microedition.lcdui.Image data. This happens transparently, without your intervention. Calls to Image.loadImage() first check if the data exists in the internal image cache. If it does, the previously created image instance is returned. This behaviour is enabled by default. You can disable it, if required.

Note: Due to the way caching is performed, altering the image through the graphics object associated with it alters *all* references to the loaded file. If you want to alter a single instance of the image, you should create a copy of the image before. For instructions, see "Copying an Image for Drawing" on page 56.

Managing the Image Cache

It can be necessary to remove images from the cache, for instance, if the application is about to perform a memory-intensive action or if an image is not needed for a long period of time. The nUIController class provides additional static methods to clear the image cache. nUIController.clearCache() removes all images from the image cache. nUIController.clearCache(String) removes a single image from the cache, with the path and file name provided that was entered when loading it initially.

Note: Although these methods are also available in the Image class itself, NativeUI performs additional management. Therefore, you are recommended to use the methods in nUIController.

Disabling Image Caching for the Whole Application

For certain applications, caching images is undesirable. You can disable the image cache by calling the static method nUIController.enableCache(boolean enableCache) and passing `false` as an argument. Ideally, this should be done as soon as possible after the application has started.

Copying an Image for Drawing

In most cases, caching images is preferable. But if you want to alter a single instance of an image that was created by an Image.loadImage() call, you should copy it to a separate instance of the image before. This is shown in the following example that uses the non-null image "oldImage".

```
//AARRGGBB data
int[] imageData = new int[oldImage.getWidth() * oldImage.getHeight()];
oldImage.getRGB(imageData, 0, oldImage.getWidth(), 0, 0, oldImage.getWidth(),
oldImage.getHeight());
Image newImage = Image.createRGBImage(imageData, oldImage.getWidth(),
oldImage.getHeight(), true);
```

Asset Catalogs

Mobile Designer supports loading images from native asset catalogs. Inside the catalog, an image can be provided at multiple sizes and referred to with one name. This allows the handset's operating system to select the most appropriate size of the image available in the catalog with regard to the device's screen, taking pixel density, and resolution.

Creating Asset Catalogs for an Application

Asset catalogs can be created using webMethods Mobile Development, or using platform-specific IDEs such as Xcode, or they can be created manually. The NativeUI Asset Catalog sample project gives an example of an asset catalog.

Note:

- It is not mandatory for an asset catalog to define an image for all device/resolution combinations. The handset's operating system determines the nearest possible match according to the platform's guidelines.
- You should create images that work with all devices at a given dpi, resolution, or scale. You are recommended to create handset-specific images only when there is a need to do so. This facilitates coding applications, reduces the number of assets to be created, and reduces the size of the final build.
- You are not recommended to create asset catalogs manually for iOS.

Creating Asset Catalogs using Mobile Development

If you want to create an asset catalog for a project created in Mobile Development, most of the work required to build and import the catalog is done automatically. For further information about how to work with assets in Mobile Development, see *webMethods Mobile Development Help*.

Creating iOS Asset Catalogs Manually with Xcode

To create iOS asset catalogs with Xcode

1. Launch Xcode.
2. Select **File > New > File...** from the menu at the top of the screen.
3. Select the **iOS** filter at the top of the screen, then scroll to the **Resources** section, and select **Asset Catalog**.
4. Click **Next** and specify where to save your asset catalog.
5. To add a new image to the catalog, right-click the left-hand pane of the screen, and select **New Image Set**. A new section opens in the right-hand pane for this new image set.
6. Double-click the image name and specify a new image name.
7. If you require more icon categories, right-click the **Image** set, and select **iPhone** or **iPad**.

Creating Android Asset Catalogs Manually

Android requires the following folders:

```
drawable
drawable-ldpi
drawable-hdpi
drawable-xhdpi
drawable-xxhdpi
drawable-xxxhdpi
```

Assets are copied directly into these folders using file names containing only lowercase letters, numbers (0-9), ".", and "_" (e.g., "my_header_image_14.png"). The file name used for the different versions of the image must be exactly the same in all folders. No sub-directories are allowed.

Other folders, such as drawable-nodpi, may also be appropriate for your project. For more information, see "https://developer.android.com/guide/practices/screens_support.html", sections *Range of Screens Supported* and *Using Configuration Qualifiers*.

Importing an Asset Catalog Using the Resource Handler

The AntTaskResourceHandler class contains the method addDirContentAsAssetCatalogs(String dir). This method includes all sub-directories of the indicated folder as asset catalogs in the application. As an example, the resources directory of a project may be structured as follows:

```
resources
+---assetcatalogs
|   +---android
|   |   +---drawable
|   |   +---drawable-hdpi
|   |   +---drawable-ldpi
```

```
|   |   +---drawable-xhdpi
|   |   +---drawable-xxhdpi
|   |   \---drawable-xxxhdpi
|   +---ios
|   |   \---AppAssets.xcassets
```

The following code would then add the asset catalog for an Android handset:

```
rh.setResourceReadSubdirectory("assetcatalogs");
rh.addDirContentAsAssetCatalogs("android");
```

Loading an Image from an Asset Catalog

In your application code, the class com.softwareag.mobile.nativeuiassetcatalog.AssetImageLoader is available. This class contains a static method, loadImage(String image_name). Enter a value for *image_name* that relates to the name of the resource you want to load, e.g. "MyImage.png". No additional path is required.

To assist with cross-platform compatibility, the given file name is automatically converted to lowercase on platforms that require it.

Android Auto-scaling

When selecting a handset-appropriate version of your image from the asset catalog, most handsets select the image that best suits the current screen and use it directly without any further modification. Android devices, however, attempt to match the selected image to the screen as closely as possible by scaling the image during loading.

This auto-scaling feature on Android can be disabled by calling the following method:

```
AssetImageLoader.setAndroidScaling(false);
```

Note: This is not available on other platforms (including Phoney) and must be done as a native-injection.

Mobile Designer Logging API

Mobile Designer provides a java.util.logging API that is based on the Java Logging API standard. The java.util.logging package that Mobile Designer provides contains classes and interfaces that are based on the Connected Limited Device Configuration (CLDC) 8 standard. You can find information about this package in the javadocs for the java.util.logging package provided with the CLDC 8 standard.

Note: The Mobile Designer version of java.util.logging does *not* include the LoggingPermission class.

The following table describes some limitations and differences in the Mobile Designer version of the java.util.logging API based on platforms:

Platform	Limitation or Difference
All Platforms	The Logger class that Mobile Designer provides does *not* include a Logger.getGlobal() method. To access the global logger, use the following: `Logger.getLogger(Logger.GLOBAL_LOGGER_NAME)`
Android	When running applications on the Android platform, the application uses the Java Standard Edition (Java SE) Logging API. It does not use the java.util.logging package that Mobile Designer provides. When configuring loggers, take into consideration that the global logger is not a root logger. The following code shows how to obtain the root logger in the logger's hierarchy. `Logger rootLogger = Logger.getLogger(` `  Logger.GLOBAL_LOGGER_NAME);` `if (rootLogger.getParent() != null)` `{` `  rootLogger = rootLogger.getParent();` `}`
iOS	When running applications on the iOS platform, the application uses the java.util.logging API that Mobile Designer provides rather than the Java SE Logging API. The standard output for the ConsoleHandler class is console in Xcode.
Phoney	When running applications in the Mobile Designer Phoney utility, the application uses the Java SE Logging API. It does not use the java.util.logging package that Mobile Designer provides.

DateFormat API

Mobile Designer provides an API to allow access to the internal date formatting logic of the mobile platform. This is provided through the com.softwareag.mobile.md.text.DateFormat class. Methods are provided to format a java.util.Date as a String conforming to a given format, and to parse Strings with a specified format back to Date objects.

File Management API

To ensure data and file persistence on the device, an API is required to access, modify, and delete files locally from the device. To work with files and folders, the com.softwareag.mobile.md.io.File class is available.

Virtual file system

To provide a consistent way to work with files and directories on every mobile platform, a virtual file system is introduced. Every root folder of the virtual fily system is mapped to the best suited folder on the device.

Path separator

For separating folders, a slash symbol (“/”) is used.

Root folders

The root of the virtual file system is “/”. It contains the following root folders:

- documents: for reading and writing access
- resources: for reading access only
- cache: for reading and writing access. The folder can be cleaned by system.
- external_storage: for reading and writing access on external storage

Valid characters for file and directory names

To avoid inconsistency between different platforms, the acceptable characters are reduced to the following:

- letters: A-Z, a-z
- digits: 0-9
- symbols: [-._]

If you use any other character, an `IllegalArgumentException` is thrown. However, the list() method can return names with any symbols that the particular native file system allows. As a result, the method cannot be used to construct new file instances. To operate such files, you can use the listFiles() method.

Android root folders mapping

- documents: mapped to /files
- cache: mapped to /cache
- resources: mapped to the Android Asset folder
- external_storage: mapped to the external SD card if it exists. Additional permissions might be needed.

iOS root folder mapping

- documents: mapped to /Documents/
- cache: mapped to /Library/Caches
- resources: mapped to [AppName.app]

- external_storage: is not mapped.

Connectivity Status API

Knowing the status of the network connections on a mobile device is crucial for modern applications. Aside from important gains with power-saving, it also allows for a much better user experience. Knowing which types of network connections are available allows you to make important decisions regarding data sync, costs, and priority of information queues.

For each platform, status detection of the different connection types is supported, such as WiFi, Bluetooth, WiMAX, wired Ethernet, or VPN.

To help manage this information, the com.softwareag.mobile.md.net.ConnectivityStatusRequester class is available.

Querying the Current Connection State

If an application needs to know the connection state infrequently, it can be useful to poll for it directly. The method getConnectivityStatus() returns `true` if any interface is connected. If a specific interface is required, you can use isWiFiConnected(), isCellularConnected(), or getCurrentConnectivityTypes().

Registering for Connection Status Notifications

For longer-term network traffic, for example a "background sync"-type situation, it is usually desirable to watch for notifications of network changes. You can register a class that implements the interface com.softwareag.mobile.md.net.IConnectivityWatcher. Every time a network interface changes state, information is provided to your application through the onConnectivityChanged() method. This information includes the network type, the new state that it has moved to, and the overall connectivity status of the device (connected to a network through at least one connection, or fully offline).

Permissions API

In the past, mobile platforms would require an application to declare any required permissions at install time. Nowadays, there are a much larger number of permissions that an application may ask for, and users have become more wary of blindly granting these permissions up front. It now makes sense for applications to ask for permission for an operation or resource at run time when it is needed. That way, the user can be made aware of what the application is doing, and why it needs that particular permission. To support this use case, Mobile Designer provides a class to ask for permissions at run time, com.softwareag.mobile.md.permissions.PermissionsManager.

The PermissionsManager class defines the permission types that may be queried. Although their names are self-explanatory in most cases, some have additional caveats:

- `PERMISSION_LOCATION_GENERAL` - A rough estimate of location, usually through IP address lookup or nearest cell tower.
- `PERMISSION_LOCATION_PRECISE` - GPS will be requested if available, otherwise this will fall back to the "general" permission.
- `PERMISSION_WRITE_EXT_STORAGE` and `PERMISSION_READ_EXT_STORAGE` - Will always be denied on devices that have no hardware provision for external storage.

Determining Existing Permissions

To test if an application already has permission for a resource, the method hasPermissionFor() can be used:

```
if(PermissionsManager.hasPermissionFor(PermissionsManager.PERMISSION_CAMERA))
{
  //do something with the camera
  ...
}
```

Requesting New Permissions

In situations where the application needs to request a permission, the method requestPermissionFor() is used. This method may take a while to negotiate, and as such, is asynchronous. The following example shows how to create an myPermissionCallback object which implements the IPermissionCallback interface in com.softwareag.mobile.md.permissions:

```
PermissionsManager.requestPermissionFor(PermissionsManager.PERMISSION_CAMERA,
myPermissionCallback);
```

If permission is allowed or denied, or if the application is interrupted, the myPermissionCallback object is notified. If the application is interrupted, you can request the permission again (if it is appropriate).

Registering Applications for Data Sharing (custom URIs and MIME-types)

Many mobile operating systems now allow you to register applications to handle certain URI schemas or file types. Mobile Designer provides a mechanism to define these registrations. Applications may register themselves as handlers for custom URIs, file types, or both.

Defining Data Sharing for an Application

To register for data sharing, the registerDataSharing Ant task is used. Definitions should be given in the project's targets/_defaults_.xml, and look like this:

```
<registerDataSharing>
    <entry scheme="customurl"/>
    <!-- App will handle URIs starting "customurl:" -->
    <entry mimeType="application/pdf"/> <!-- App will accept PDFs -->
</registerDataSharing>
```

If you want to just handle audio files:

```
<registerDataSharing>
    <entry mimeType="audio/*"/> <!-- App will accept any audio -->
</registerDataSharing>
```

You can find a list of currently supported MIME-types for files in the section below: "Supported MIME-types" on page 63.

If you want to just handle custom URIs:

```
<registerDataSharing>
    <entry scheme="myurl"/> <!-- App will handle URIs starting "myurl:" -->
</registerDataSharing>
```

Handling Data Sharing Events

When the application is asked by the operating system to handle data sharing, either Application.onStartAppFromURL(String scheme, String url) or Application.onStartAppFromFile(String mimeType, String absolutePath) will be called. These calls react to user interaction coming from outside the application and may occur at any point in the application's lifecycle.

The default implementations of onStartAppFromURL() and onStartAppFromFile() store the incoming data and allow it to be queried through getters in the base Application class.

Alternatively, it is possible to override onStartAppFromURL and onStartAppFromFile from a class that extends Application. Doing so will ensure that the application is notified directly. At this point, standard Java classes are available for use. If the application has been started by the operating system, onCreateMainWindow() on the NativeUI side will not have been called yet.

Note: If the getter functionality in Application is still required, it is vital that any overriding method for onStartAppFromURL() or onStartAppFromFile() calls super.onStartAppFromURL() or super.onStartAppFromFile(), respectively.

Supported MIME-types

The following is a list of MIME-types recognised by Mobile Designer when defining the type of content that can be handled.

Note: In case of wild-cards (e.g., "audio/*"), the mobile operating system underneath will likely have more specific MIME-types that can be defined to narrow the range of file types associated with the application.

- application/octet-stream
- application/pdf
- application/vnd.ms-excel
- application/vnd.ms-powerpoint
- application/vnd.openxmlformats-officedocument.presentationml.presentation
- application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
- application/vnd.openxmlformats-officedocument.wordprocessingml.document
- application/x-zip-compressed
- application/zip
- application/zip-compressed
- audio/*
- image/*
- multipart/x-zip
- text/*
- video/*

Credential and Isolated Storage API

In order to store more sensitive data, most mobile platforms now have some concept of *isolated storage* or *protected storage*. Mobile Designer provides an API to allow applications to make use of this kind of storage through the classes in the com.softwareag.mobile.md.io package. Default implementations are provided to store key/value pairs and server to user name/password mappings. In cases where a different security model is desired,you can create your own classes conforming to the interfaces in this package.

Note: It is important to be aware of the limitations and concepts defined with each platform when dealing with sensitive data to ensure that the data is stored in an acceptable way. The following *Security Considerations* section gives information on the default classes provided and their native implementations to assist in this regard.

Security Considerations

The following table provides details about the APIs and classes used to store data for the CredentialStore and IsolatedKeyValueStore.

Platform	CredentialStore	IsolatedKeyValueStore
Android	IsolatedKeyValueStore class	android.content.SharedPreferences object
iOS	Keychain Services	NSData with NSDataWritingFileProtectionComplete and NSURLIsExcludedFromBackupKey set.
Phoney	Flat File (unprotected)	Flat File (unprotected)

Orientation API

As of Mobile Designer 10.3, Fix 5, orientation mode can only be specified at runtime. By default, screen orientation is not locked to one specific position and rotates together with the device. To lock screen orientation to portrait mode, you can override the init() method of the MDApplication class with the following code:

```
IOrientationService orientationService = (IOrientationService)
   getService(IMDApplication.ORIENTATION_SERVICE);
orientationService.lock(IOrientationService.ORIENTATION_PORTRAIT);
```

The Orientation API deprecates the following properties:

- project.handset.landscape.mode
- project.handset.orientation.limiter
- project.handset.portrait.mode.orientation

Location API

Mobile Designer provides an API to access the location services of the mobile device. This is provided through classes in the com.softwareag.mobile.md.location package. A class conforming to the ILocationManager interface can be obtained using the getService() method of MDApplication as follows:

```
ILocationManager locationManager =
   (ILocationManager)MyNewApplication.getInstance()
      .getService(IMDApplication.LOCATION_MANAGER_SERVICE);
```

Location updates are passed back through a class conforming to the ILocationWatcher interface. This interface allows information about location updates, errors during use,

and initial setup failures to be provided. The locationUpdatesEnded() method is called at any time that location updates are stopped. This can happen normally through ILocationManager.stopLocationUpdates() or after an error occured. Calls to locationUpdatesEnded() do *not* happen if an initial setup failure occurs, since a failure at that level produces no location updates.

```
class ResultHandler implements ILocationWatcher
{
    public void locationUpdate(Location location)
    {
        System.out.println("Location was updated at "
              + location.timestamp + "\n" +
           "Latitude  : " + location.latitude + " +/- "
              + location.latitudeAccuracy + "\n" +
           "Longitude : " + location.longitude + " +/- "
              + location.longitudeAccuracy + "\n" +
           "Altitude  : " + location.altitude  + " +/- "
              + location.altitudeAccuracy);
    }
    public void locationUpdatesEnded()
    {
        System.out.println("Location updates ended.");
    }
    public void onError(int why, String description)
    {
        System.out.println("An error : " + why + " " + description);
    }
    public void onFailure(int why, String description)
    {
        System.out.println("A failure : " + why + " " + description);
    }
}
```

To configure the location hardware, create a LocationSettings object and pass it to the appropriate call in the ILocationManager. The LocationSettings object encapsulates the application's preferences in terms of an accuracy/power consumption trade-off. It also determines if the updates should come back to the application code at a preferred time interval or a preferred distance travelled.

Note: For both distances and times given, the operating system makes a best-effort attempt to get as close as possible to the requested value, but this is not exact due to many factors (including GPS signal loss, operating system, power or CPU constraints, requests or interrupts from other applications, buffering, etc). Therefore, the application must not make any direct assumptions based only on the fact that an update happened.

Use the following code to configure the location hardware:

```
//prefer updates once per minute (average)
LocationSettings settings =
    new LocationSettings(LocationSettings.PROFILE_HIGH_POWER_ACCURATE, 60000L);
ILocationWatcher myLocationWatcher = new ResultHandler();
locationManager.beginLocationUpdates(myLocationWatcher, settings);
```

It is important to consider that the one-shot getLocation() method also requires a LocationSettings object. This is because the location hardware may not already be in use and needs to be initialised fully in case it is needed elsewhere before the pending request is served. The getLastKnownLocation() method does not need to switch any hardware on to

return a result and so does not need a LocationSettings object to function (and can therefore be very useful in low-power situations).

Additionally, a CoordinateHelper class is available to provide some methods to assist in coordinate conversion to/from human-readable forms and measuring distances between Location objects.

Location Updates in Background Mode

To receive location updates in background mode, you must enable them explicitly using the following Ant property:

```
<property name="project.handset.background.location.updates"
value="true"/>
```

Note: Android 8.0 (API level 26) and later limits the amount of updates in background mode to only a few updates per hour.

Geocoding

As well as offering a few helpful methods for coordinating conversion and distance measuring, the CoordinateHelper class also has some methods that allow for geocoding, that is, conversion between street addresses and latitude/longitude pairs.

Before writing applications that employ geocoding, think carefully about how it will be used. Geocoding relies on servers provided by Apple (for iOS devices) and Google (for Android). If you are considered to be "over-using" the service, you may find your applications rate limited or banned. A few good rules of thumb are:

- Make at most one geocoding request per user action.
- If the user could make multiple geocoding requests against the same location, it is worth caching the results. Doing so will make the application more responsive.
- If the user is moving around and the application is using geocoding to update the user's address, only send requests when the user has moved a significant distance *and* a significant amount of time has passed. A request rate higher than 1 per minute is likely to be frowned upon.
- Do not start a geocoding request unless the result is seen immediately by the user. Applications running in the background should continue gettting location updates without asking for new geocoding data until the application is brought to the foreground again.

Note: Not all handsets support geocoding. It is important to check the result of canUseGeocodingApi() before calling other geocoding methods.

To convert from a latitude/longitude pair to a street address, use the findPlacesBy() method:

```
if(CoordinateHelper.canUseGeocodingApi())
```

```
{
 Location myLocation = new Location();
 myLocation.latitude = myLatitude; //double value, defined elsewhere
 myLocation.longitude = myLongitude;
 CoordinateHelper.findPlacesBy(myLocation, 10, new GeoCallback());
}
```

The GeoCallback object has an array of Placemark objects passed back in its onPlacesFound() method, along with the original myLocation object. Usually, the results are ordered from closest to furthest match.

To get a set of coordinates from a street address, use the findLocationsBy() method:

```
if(CoordinateHelper.canUseGeocodingApi())
{
 String address = "10 Downing Street, London, England";
 CoordinateHelper.findLocationsBy(address, 10, new GeoCallback());
}
```

The GeoCallback object has an array of Location objects passed to it, indicating potential matches.

Native Authentication API

You can use native authentication methods, like identification with a Touch or Fingerprint ID, Face ID, PIN, or some other platform-specific methods, for some security-critical tasks like login, showing sensible data, publishing data, etc.

Note: Before a particular authentication method can be used, it must be configured on the user's device.

Use the following code to request native authentication:

```
IDeviceAuthenticator deviceAuthenticator = (IDeviceAuthenticator)
   MDApplication.getInstance().getService(IMDApplication.DEVICE_AUTHENTICATOR_SERVICE);
deviceAuthenticator.authenticate("Authentication is required",
   "This action requires authentication.", this);
```

If you request native authentication, you must provide the IDeviceAuthenticationCallback method. This method informs you if authentication was successful or not. Authentication fails if a device does not support native authentication or if it is not fully/correctly configured.

Linking to External (3rd Party) Native Libraries/Frameworks

You can link an application to an external native library or framework using the addExtraLibs Ant task. This task may be used in _defaults_.xml or the individual handset targets themselves. For `<addExtraLibs>`, you must set the `platform` attribute. Inside this tag, libraries can be defined using `<library>`. You must specify a `name` attribute that points to the name of the library or framework to be used. It must end with the correct file extension: `.framework` for a framework, `.dylib` for a dynamic library, and `.a` for a static library.

iOS

Note: Under iOS, a static library should be built as a FAT version. This file should contain code for all the required architectures.

```
<addExtraLibs platform="ios-app">
    <library name="MobileCoreServices.framework"/> <!-- using default path for
    Frameworks -->
    <library name="Twitter.framework"/>
    <library name="libz.1.2.5.dylib"/> <!-- using default path for libraries -->
</addExtraLibs>
```

Here, the default paths are used to import libraries. These default paths are determined by the SDK that is currently used for the compilation, the device architecture, and the target build type. For iOS applications using an iOS 8.3 SDK, the default paths are:

For the simulator

- Frameworks:
  - /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator8.3.sdk/System/Library/Frameworks
- Libraries
  - /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator8.3.sdk/usr/lib
  - /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator8.3.sdk/usr/include

For iOS 8.3 on devices

- Frameworks:
  - /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS8.3.sdk/Developer/SDKs/iPhoneSimulator.sdk/System/Library/Frameworks
- Libraries
  - /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS8.3.sdk/Developer/SDKs/iPhoneSimulator.sdk/usr/lib
  - /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS8.3.sdk/Developer/SDKs/iPhoneSimulator.sdk/usr/include

The location of the default paths cannot be altered, but custom paths for libraries and includes may be specified using the `libPath` and `includePath` attributes. Static libraries will be used if they have their `libPath` and `includePath` set directly or if they are stored in a location pre-configured as a search path within Xcode.

You can also import libraries from custom locations using the optional `libPath` and `headerPath` attributes.

```
<addExtraLibs platform="ios-app">
<!-- use a custom path for libraries-->
    <library name="MyTestLib.1.2.dylib" libPath="/Users/libs" headerPath="/
    Users/include"/>
    <library name="MyStatictLib.1.2.a" libPath="/Users/libs" headerPath="/
    Users/include"/>
</addExtraLibs>
```

The `libPath` attribute gives the location of the actual library/framework file itself, and the `headerPath` gives the location of any header files that can be used for compilation.

Note: Both the `libPath` and `headerPath` attributes must currently be a full path (e.g., /Users/*your_name* /*your_project* /*path_to_libs*), not relative paths.

Android

Android uses `android-apk` as setting for the `platform` attribute, and a .jar file as library. You can refer to a .jar file that is located inside the Android SDK's platforms folder when you specify the `name` attribute, or you can specify a custom location for a library using `libPath`.

For an example, see the following codeblock:

```
<addExtraLibs platform="android-apk">
  <library name="SomeDefaultLib.jar"/>
  <!-- Library will be found in appropriate sdk/platforms/android-X folder,
  where X is the current API level -->
  <library name="MyCustomLib.jar"
  libPath="C:/path_to_custom_libraries" />
  <!-- a jar in a custom location -->
</addExtraLibs>
```

Using System.getProperty to Obtain Device Information

You can use the Java System.getProperty(String) method to return system information for the device on which your application is running. The table below lists properties that you might find useful.

When invoking the Java method, supply the property name as the String input variable. For example, if you want to use the `mobiledesigner.device.name` property, invoke the following:

```
System.getProperty("mobiledesigner.device.name")
```

Property and Description
`mobiledesigner.device.firmware`

Property and Description
The System.getProperty method returns information that identifies the firmware of the device. If you are running the application in Phoney, the System.getProperty method returns the full Mobile Designer version number including the build number, for example, “9.5.1.2.344”.
`mobiledesigner.device.name` The System.getProperty method returns information that identifies the device hardware, for example, “iPhone4S”. If you are running the application in Phoney, the System.getProperty method returns “Phoney”.
`mobiledesigner.device.uid` The System.getProperty method returns information that uniquely identifies the specific device on which the application is running. This is typically a unique String.
`mobiledesigner.display.ppi` The System.getProperty method returns the resolution of the device’s screen in pixels per inch, for example, “240”.
`mobiledesigner.domain.availability:`*`domain_name`* The System.getProperty method returns whether the device can connect to the domain specified by *`domain_name`*. For example, if you want to determine whether the device can connect to www.softwareag.com, use `mobiledesigner.domain.availability:www.softwareag.com`. The return values are one of the following: ■ `true` if the device can connect. ■ `false` if the device cannot connect.
`mobiledesigner.locale` The System.getProperty method returns the language the device is currently configured to use. The language is returned in the following format: *language _COUNTRY* where: ■ *language* is the two-character, lowercase language code defined by ISO 639. ■ *COUNTRY* is the two-character, uppercase country code defined by ISO3166.

Property and Description
Note: In some circumstances, the country code might not be returned, for example, if the device does not grant access to the country information. In these circumstances, only the two-letter language code is returned, for example, “en”.

`mobiledesigner.online.availability`

The System.getProperty method returns whether the device is connected to a network. The returned information is one of the following:

- If the device is *not* connected to a network, the method returns `false`.
- If the device is connected to a network, the return information is in the following format:

  `true:`*`method`*

  For example, if the device is connected to WiFi, the return information is “true:wifi”.

  If the network interface details are not specified, `method` is “unknown”, for example, “true:unknown”.

`mobiledesigner.platform`

The System.getProperty method returns the platform for the device. The platform name that the System.getProperty method returns matches how Mobile Designer lists platforms, for example, “Android” or “iOS”.

If you are running the application in Phoney, the System.getProperty method returns the name of the platform you are simulating in Phoney.

`mobiledesigner.android.display.dpi.class`

Provides a DPI value based on the screen's display class. This will not follow the exact DPI of the device, but will give a value equivalent to a typical device in this class, be it `low`, `medium`, `high`, `xhigh`, etc. density. The values returned are based on the scaling factor given in Android's DisplayMetrics.density as applied to a 160 DPI screen. So, a device with relatively high DPI may have a DisplayMetrics.density value of 2.5, and therefore return a value of 400 (160 *2.5) for the DPI class.

`mobiledesigner.android.display.capabilities`

Will output a String enumerating some of the capabilities for the device. For now, the expected format of this String is *screenlayout_size_category* |*display_is_long* . This String may expand to include more data later, and will continue to use the pipe character ("|") as a separator between fields. The values returned are based on information obtained from the android.content.res.Configuration class. For the *screenlayout_size_category* , current possible return values are

Property and Description
SIZE_SMALL, SIZE_NORMAL, SIZE_LARGE, SIZE_XLARGE and SIZE_UNKNOWN. For the *display_is_long* category, possible return values are LONG_YES, LONG_NO and LONG_UNDEFINED. A typical handset may return SIZE_XLARGE|LONG_YES.

Creating the User Interface

To create the user interface for your mobile application, use the Mobile Designer Native User Interface. For more information, see the *webMethods Mobile Designer Native User Interface Reference*.

Defining Resources for a Mobile Application Project

About the Resource Handler

Each project requires its own resource handler. Software AG recommend that you use the default UniversalResHandler implementation that Mobile Development provides. The goals of the resource handler are to:

- Define all the resources to include with your mobile application, such as texts and graphical assets.
- Set parameters that define the identifiers for resources, text lines, menus, and resource blocks. Mobile Designer includes the parameters in the Parameters.java. For more information about the Parameters class, see "Application and Parameter Classes" on page 50.

You can store single resource files that your project requires, for example image files and text files, in a location within your project's folder. For more information, see "Storing Resource Files for the Project" on page 77. When adding resources with Mobile Development, the resource files are automatically stored in the proper location.

Mobile Development automatically sets up project properties for the resource handler, for example, to specify the name and location of the resource files. However, for debugging and testing purposes, you can set and modify these properties on your own as described in "Setting Project Properties for the Resource Handler" on page 78.

After defining the resources for your project, you can use the resources in your application code. For more information, see "Accessing Resources in Your Application Code" on page 79.

Coding the Resource Handler

Deprecated. It is still possible but not preferable to create the resource handler from the beginning on your own.

Resource Handler Requirements

- Your resource handler must extend com.softwareag.mobile.reshandler.ResourceHandler. This class includes the projectResourceScript method.
- In the resource handler, include all resource handling logic that your project requires in the projectResourceScript method.

  When building your project, Mobile Designer calls the resource handler's projectResourceScript method.

Methods that Mobile Designer Provides for the Resource Handler

Mobile Designer provides the com.softwareag.mobile.reshandler.AntTaskResourceHandler that contains methods you can use in your resource handler.

The com.softwareag.mobile.reshandler.ResourceHandler class, which your project's resource handler must extend, includes the `rh` field that defines a link to AntTaskResourceHandler. As a result, you can easily execute the methods in the AntTaskResourceHandler using the following format, where *`method`* is the AntTaskResourceHandler method you want to invoke:

```
rh.method
```

For example, to use the AntTaskResourceHandleraddFile method to create a resource from a file, use the following:

```
rh.addFile
```

The types of actions you can accomplish using methods provided by the AntTaskResourceHandler include:

- Add resources to the project.
- Set and get IDs for resources, text lines, menus, and resource blocks.

  Note: When the resource handler sets identifiers, Mobile Designer adds corresponding parameters to the Parameters.java class. For more information, see "Setting Parameters in the Resource Handler Code" on page 89.

- Set application-specific parameters. For more information, see "Setting Parameters in the Resource Handler Code" on page 89.

To learn about *all* the methods that Mobile Designer provides for a resource handler, see information about com.softwareag.mobile.reshandler.AntTaskResourceHandler in *webMethods Mobile Designer Java API Reference*.

Setup to Allow the Resource Handler and Application Code to Share Common Code

You can set up your project's resource handler and the application code so that they share common code. For example, the resource handler and application code might use the same set of constant values, or you might have common code that you want to use in both the resource handler and the application code.

To use shared common code, place the shared code in a folder within your project. Then when defining the project.runtime.project.src.path and project.reshandler.src.path Ant paths for your project, include `<pathelement>` tags to the location that contains the shared code. For example, if you placed shared code in the project's src/shared_code_ folder, you might define Ant paths like the following in your project's _defaults_.xml file:

```
<path id="project.runtime.project.src.path">
   <pathelement path="${basedir}/src/core"/>
   <pathelement path="${basedir}/src/shared_code"/>
</path>

<path id="project.reshandler.src.path">
   <pathelement path="${basedir}/reshandler"/>
   <pathelement path="${basedir}/src/shared_code"/>
</path>
```

The _FunctionDemo_ sample application provides an example of this shared code setup.

Sample Resource Handler Code

You can find examples of basic and complex resource handler logic in all the samples applications provided with Mobile Designer.

Accessing Resources in Your Application Code

Your resource handler defines the resources available to your application. For information about how to use the resources in your application code, see "Accessing Resources in Your Application Code" on page 79.

Using Resource Blocks and Resource Packs

You can code your resource handler to put resources into *resource blocks* and *resource packs*.

- **Resource blocks** are a group of resources. For example, you might bundle launch screens into one block, and images into another block.
- **Resource Packs** are bundles of resource blocks.

The use of resource blocks and packs is not required. Reasons to use them might be to save space and/or increase speed. A single larger file compresses better than multiple smaller files. Opening one file can be faster than opening several smaller files. However, with current devices, space and speed are no longer major issues for mobile applications. As a result, using resource bundles and packs for reasons of space and speed is not typically needed.

Another reason you might want to use resource blocks and packs is for better security. When bundling resources, it is more difficult to determine names of resources and more difficult to take malicious actions without having to rebuild the resource packs.

Defining Resource Blocks

To define resource blocks in the resource handler code, execute the startResourceBlock method that is in the com.softwareag.mobile.reshandler.AntTaskResourceHandler class. For information about how to use an AntTaskResourceHandler method in your resource handler, see "Methods that Mobile Designer Provides for the Resource Handler" on page 74.

When you execute the startResourceBlock method, you assign a name to the resource block. You can assign any name to resource blocks that is appropriate for your application. When you execute the startResourceBlock method, the resource block you create becomes the current resource block, and all subsequent resources you add are included in the current block. For example, the following code sample shows how to start a resource block named "IMAGES" and add the file "logo.png" as the resource name "res_logo.png":

```
public class ResHandler extends com.softwareag.mobile.reshandler.ResourceHandler
{
    public void projectResourceScript()
    {
          .
          .
          .
          rh.startResourceBlock ("IMAGES");
          rh.setResourceReadSubdirectory ("graphics"));
          rh.addFile ("res_logo.png", "logo.png");
          .
          .
          .
    }
}
```

To start a new resource block, execute the startResourceBlock method again.

Using resource blocks helps you to control the memory your application uses. When you bundle resources into a block, you can cache the block(s) your application needs based on the application state. In other words, your application can load and unload blocks so that only the resources that are required for a state in your application are loaded and therefore the application is not using valuable memory space for unneeded resources.

Important: If your resource handler bundles resources into blocks, when your application needs to use the resources that are in a block, be sure the application code caches the block into memory before loading its resources.

A best practice is to use the same resource blocks for all the platforms for which you build your application. However, if the devices on which your application runs have varying memory and you are concerned that some devices cannot keep all the resources in memory, you might want to split your resources up in the application bundle for compression, decompression, or run-time memory management. One approach you can use is to bundle the blocks into packs.

Using Resource Packs

Resource packs are bundles of resource blocks. You can define several packs of different combinations of your resource bundles. You might customize packs for each of the devices your application supports. For example, some devices might have the memory capacity to cache all the blocks in memory at application load time, while other devices might only be able to cache one block at a time due to memory limitations.

To define resource packs in the resource handler code, execute the allocateResourceBlockToPack method that is in the AntTaskResourceHandler class. When you execute the allocateResourceBlockToPack method, you identify a resource block and specify the identifier for a resource pack. For example, the following line of code allocates the resource block named "IMAGES" to the pack with identifier “0”:

```
rh.allocateResourceBlockToPack ("IMAGES", 0);
```

At build time when Mobile Designer runs the resource handlers, it creates an individual file for each resource pack that your resource handler defines. When you use resource bundles, your resource handler should define, at a minimum, at least one resource pack that contains all the resource bundles. If a block is not included in a pack, Mobile Designer saves that block’s resources as individual resources in the final binary.

The resource packs you define in your resource handler are transparent to your application code. When Mobile Designer runs the resource handler, it keeps a record of each resource pack along with the blocks that the pack contains. It also keeps track of the resources that are in each block. When your application code loads a resource block into memory, Mobile Designer determines the appropriate pack to load for that block. As a result, you can customize the packs for each device your application supports without being concerned about altering your application code.

Storing Resource Files for the Project

When adding resource files within Mobile Development, they are automatically stored in the proper location.

However, you can continue to manually add resources that your project uses within the *your_project* /resources folder.

Launch Screens for Applications

For iOS, when defining the resources for your application, you should include a launch screen image. A *launch screen* is a static image that a device displays when the user launches the mobile application. The purpose of the image is so that the user can see that the application is starting while the application initializes its initial window and views.

For more information about how to add a launch screen, see *webMethods Mobile Development Help*, *Importing Images for App Icons and Launch Screens*.

Setting Project Properties for the Resource Handler

The following table lists the resource handler project properties. The table specifies the properties you can set to provide details about the resource handler for your project. For instructions about how to set properties, see "Where You Set Properties" on page 84 and "Setting Project Properties" on page 86. For further details about the properties, see "Resource Handler Properties" on page 197.

Property and Description
project.java.reshandler.name Required. Specifies the Java package/class name of the resource handler class.
project.reshandler.src.path Required. Specifies the location of your project's resource handler script and any associated classes.
project.resource.dir.root Required. Specifies the location of the resource files (image files, etc.) for your project.
project.reshandler.additional.libs.path Conditionally required. Specifies the location of classes that the resource handler requires. Required only if your resource handler requires additional classes.
debug.remember.resource.names Optional. Specifies whether you want the Mobile Designer to record the names of the resources included in the build rather than the resource IDs. By default, the resource IDs are recorded, not the resource names.
project.compiled.resources.info.format Optional. Specifies whether you want the _compiled_resources file that Mobile Designer creates when it runs your resource handler to be in XML or text format. For more information, see "Managing Memory for Your Resource Handler and Resources" on page 79. By default, Mobile Designer creates a .txt format file.

Managing Memory for Your Resource Handler and Resources

If your mobile application project's resource compilation is memory and resource intensive, you can increase the amount of memory available for Ant to avoid encountering out-of-memory exceptions during the execution of the resource handler. To increase this memory, use a system property to set the amount of memory. The following shows examples:

- On Windows: `set ANT_OPTS=-Xmx256m`
- On Linux: `export ANT_OPTS=-Xmx256m`

Adjust the value, "256m", used in the example to reflect how much memory your resource handling needs.

You can estimate the amount of memory your application will require when it runs using the information in the _compiled_resources file, which resides in your project's _temp_ folder. Mobile Designer creates this file when it runs your project's resource handler. Use the project.compiled.resources.info.format project property to indicate whether you want the _compiled_resources file to be in XML or text format. The _compiled_resources file contains information on the resources, as well as their resultant resource blocks, resource packs, and file sizes.

Accessing Resources in Your Application Code

To access resources in your application code, Mobile Designer provides the following run-time classes:

Use in your application code to access and manage the audio resources. The AudioHandler class includes the loadSound and unloadSound methods that you can use to load and unload audio resources.

- and

  Use in your application code to access and manage image resources. The ImageBase class includes the getImage, loadImageID, and unloadImageID methods that you can use to load and unload image resources.

  Use in your application code to access and manage text line resources. The TextHandler class includes the getString method to get a text string.

  If you use resource blocks and packs, use this class in your application code to manage the blocks and packs. The ResouceHandler class includes the loadResourceBlock and unloadResourceBlock methods that you can use to load and unload resource blocks.

When using a method to load or unload a resource, you specify the resource's identifier. When Mobile Designer runs your resource handler, it sets parameters for the resources in the com.softwareag.mobile.runtime.Parameters class. For example, if you want to load an

audio resource with the identifier `RESID_STARTUP`, you can use the following method call:

```
AudioHandler.loadSound (Parameters.RESID_STARTUP);
```

If your resource handler bundles resources into resource blocks and packs, in your application code, you must load the resource bundle containing a resource before you load the specific resource. Continuing with the previous example, suppose the `RESID_STARTUP` audio resource is included in the resource block with the identifier `RESBLOCKID_AUDIO`, you can use the following method calls:

```
ResourceHandler loadResourceBlock (Parameters.RESBLOCKID_AUDIO);
   AudioHandler.loadSound (Parameters.RESID_STARTUP);
```

When the application is no longer using the resources in a block, it can use the unloadResourceBlock, which is also in the ResourceHandler class to unload the resource block.

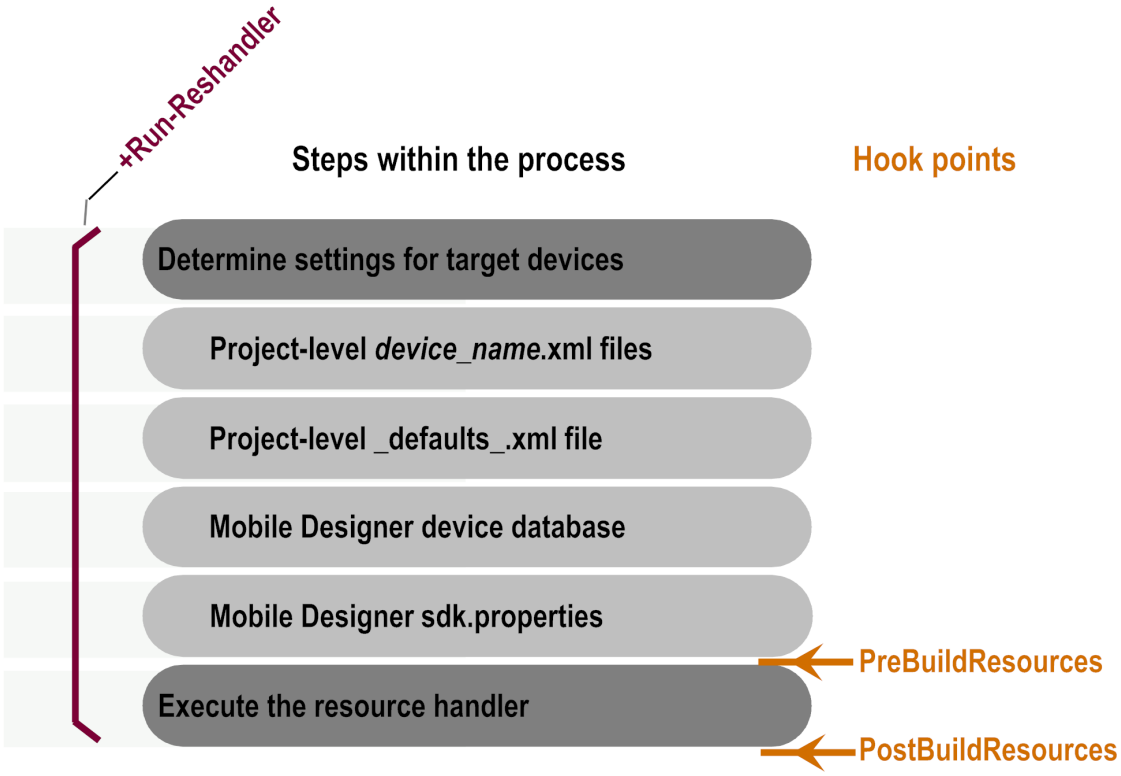
Note: If you are using resource blocks and packs, your application code does not need to load and unload resource packs. Mobile Designer keeps track of the packs required for each resource block. When you load a resource block, Mobile Designer takes care of loading the appropriate pack that contains the resource block.

Compiling Resources Using the +Run-Reshandler Ant Target

Use the +Run-Reshandler Ant target if you want to compile the resources, such as text files and images, for the current device without compiling the source code for the mobile application.

The following diagram shows the steps that Mobile Designer performs when you use the +Run-Reshandler Ant target. See the table below the diagram for a description of the steps. The table also indicates hook points where Mobile Designer can run custom Ant scripts that you provide. For more information about hook points, see "Creating Custom Ant Scripts to Run at Predefined Hook Points" on page 125.

+Run-Reshandler Ant Target



1 Determine settings for target devices

Mobile Designer determines the settings for the device for which it is building the application. It retrieves the settings from the following sources in the order listed.

- Project-level *device_name* .xml files
- Project-level _defaults_.xml file
- Mobile Designer device database
- Mobile Designer sdk.properties file

Mobile Designer uses the first setting it encounters. For example, if Mobile Designer encounters a setting in the project-level target *device_name* .xml file and then again in the project-level _defaults_.xml file, Mobile Designer uses the setting from the target *device_name* .xml file.

For more information about:

- Project-level device files, see "Where You Set Properties" on page 84 and "Setting Project Properties" on page 86

- Mobile Designer device database, see "Devices that a Mobile Application Supports" on page 91
- Mobile Designer sdk.properties file, see "Mobile Designer Configuration Properties (sdk.properties)" on page 18

hook point **PreBuildResources**

If you have created an Ant script to run at the PreBuildResources hook point, Mobile Designer runs the Ant script.

2 **Execute the resource handler**

Mobile Designer runs the resource handler that you created for the project. When running the resource handler, Mobile Designer records all the resources required for your application. Mobile Designer also creates the class. For more information about creating the resource handler, see "About the Resource Handler" on page 73. For more information about the Parameters class, see "Application and Parameter Classes" on page 50.

At this point in the build process, Mobile Designer uses the following project properties:

- project.reshandler.src.path for the location of the project's resource handler script and any associated classes
- project.java.reshandler.name for the name of the resource handler class you created for your project
- project.resource.dir.root for the location of the top-level folder that contains the resources (image files, etc.) for your project
- project.reshandler.additional.libs.path for the location of additional libraries that your project's resource handler requires

For more information about these properties, see "Resource Handler Properties" on page 197.

hook point **PostBuildResources**

If you have created an Ant script to run at the PostBuildResources hook point, Mobile Designer runs the Ant script.

Setting Properties and Parameters for a Mobile Application Project

About Properties and Parameters

Properties and parameters are project settings used when building your project or at run time.

- **Properties** are build-time settings that your project's build script can access when building your project. At build time, the build process can access property settings to determine information, for example, the device for which your application is being built. As a result, you can set properties to manipulate how your application is built for specific platforms and/or specific target devices, including and/or excluding features for the devices your mobile application supports.

  Note: Your application code *cannot* reference property settings at run time. Use parameters for settings that are available to your application's run-time code.

- **Parameters** are run-time attributes that you can use in your application code. For example, at run time your application can access parameters to retrieve information about the resources included in the application to perform such tasks as loading resources. Another example is that your application code can access information about the device on which the application is running to branch the logic based on the needs of that specific device. For more information, see "Using Parameters in Your Application Code" on page 90.

  When Mobile Designer builds a project, it runs the project's resource handler before it compiles your application. The resource handler creates the com.softwareag.mobile.runtime.Parameters class that includes all the parameters.

  The Parameters.java includes:

  - Parameters that Mobile Designer defines and uses.
  - Application-specific parameters related to the resources in your project.

- **System Properties**, like parameters, can be accessed at run time, but in contrast to parameters, they do not appear in the Parameters class. Instead, you can access them using the System.getProperty method. Some system properties define default settings. For example, the system property `ios.webview` specifies which implementation of Webview must be used internally, UIWebview or WKWebview. Other properties can be used for any purpose.

Where You Set Properties

Properties are generally set in the following files. When you set properties, be aware that when Mobile Designer builds a project, it obtains properties from the files in the order listed and uses the first setting it encounters for a property.

- Project-specific target device files (*device_name* .xml) that contain device-specific settings for devices that the project supports. For more information about target device files, see "Devices that a Mobile Application Supports" on page 91 and "Adding a Device to a Project" on page 91.

  Mobile Designer sets some properties in a target device file when you add a device to a project. You can change and/or add additional project properties. For instructions, see "Setting Project Properties" on page 86.

- Project-level _defaults_.xml file that contains default settings for all devices in a project.

  To start a project, you typically clone an existing project. As a result, your project starts with the settings in the cloned project. You can change and/or add additional properties. For more information, see "Setting Project Properties" on page 86.

- If you want to override settings for your mobile application project, set project properties in the project's target device files for the device. For instructions, see "Setting Project Properties" on page 86.

- The Mobile Designer sdk.properties contains default property settings that affect all projects. For more information, see "Updating the sdk.properties File to Configure Mobile Designer" on page 18 and "Mobile Designer Configuration Properties (sdk.properties)" on page 18.

Where You Set System Properties

Default system properties are defined in the md.system.properties file which is located at *your_installation_directory* /Tools/Build/Runtime/v1.0.0.

Important: Do not modify this file as it is overridden by every update.

In the md.system.properties file, you can see which properties are used by Mobile Designer. To change the default settings, you must create a new system.properties file in the root folder of your project. In this file, you can override the default values.

To use a specific system.properties file, specify the path and name of this file in the Ant property `custom.system.properties`.

For example, you can insert the following line in your _defaults_.xml file to use the release.system.properties file from your root project folder: `<property name="custom.system.properties" value="release.system.properties"/>`

Project Properties

The following table lists the properties that must be set for your project. Mobile Development handles these properties. However, for debbuging and testing reasons, you can still specify and modify them on your own. For information about how to set properties, see "Setting Project Properties" on page 86.

Property
project.runtime.project.src.path Required. Specifies the location of the run-time code for your mobile application.
project.runtime.additional.classes.path Conditionally required. Specifies the location of classes required for building your project. Required only if your project requires additional precompiled classes to build the application.
project.runtime.additional.stubs.path Conditionally required. Specifies the location of the stubs required for compilation. Required only if your project requires additional stubs to compile the application's run-time source code.
project.langgroup.group_name Required. Specifies the language(s) that you want your application to support.
project.jarname.format Required. Specifies the file name format that you want Mobile Designer to use when naming your application's final binary.
project.java.reshandler.name Required. Specifies the Java package/class name of the resource handler class you created for your project.
project.reshandler.src.path Required. Specifies the location of your project's resource handler script and any associated classes.
project.resource.dir.root

Property
Required. Specifies the location of the resource files (image files, etc.) for your project.
project.reshandler.additional.libs.path Conditionally required. Specifies the location of classes that the resource handler requires. Required only if your resource handler requires additional classes.
mobiledesigner.buildscript.version Required. Specifies the version of the Mobile Designer build scripting system to use when building your application.
mobiledesigner.runtime.version Required. Specifies the version of the Mobile Designer run-time system to use for your application.
project.java.midlet.name Required. Specifies the name of the root MIDlet/Application class of your project's run-time code. Typically this is the class that extends .
project.jar.name Required. Specifies a text name you want your application to have when installed on a device.

Setting Project Properties

For a list the properties that must be set for a project, see "Project Properties" on page 85.

To set a value or change a predefined value for your project, you can set property values within the project's targets folder.

- Specify properties in the project's _defaults_.xml file apply to the settings to *all* devices that the application supports.
- Specify properties in target device XML files to apply settings to a single device that the application supports. The properties you place in target device XML files override settings in the _default_.xml file.

Note: The properties you specify for the project override settings you make for all projects in the sdk.properties file.

To set project properties

1. In Software AG Designer, in the Project Explorer view, expand the Mobile Designer project for which you want to define properties.
2. In the project's targets folder, double-click the file to which you want to add a property:
   - To apply the setting to *all* the devices the application supports, double-click the _defaults_.xml file.

     project_folder /targets/_defaults_.xml
   - To apply the setting to a single device that your application supports, add the property to the XML file for that device:

     project_folder /targets/*device_file* .xml

     For example, if your application supports IOS_Apple_UniversalRetina and you want to apply the settings only to this device, add the settings to the IOS_Apple_UniversalRetina.xml file.
3. Locate the area of the file where you want to add the property. Refer to comments in the file to find good location.
4. Add the property to the file using the following format:

   ```
   <property name="PropertyName" value="PropertyValue"/>
   ```

   For example:

   ```
   <property name="cross.compiler.extractinners" value="true"/>
   ```
5. After adding properties, save the file.

Where You Define Parameters

You define application-specific parameters in the following files:

- You define application-specific parameters related to the resources in your project in the resource handler code. For more information, see "Setting Parameters in the Resource Handler Code" on page 89.
- You define general application-specific parameters in the following locations:
  - Project's _defaults_.xml and target device files. For more information, see "Setting Parameters in the _defaults_.xml and Target Device Files" on page 88.
  - If you still code your own resource handler, you set application-specific parameters in the resource handler that you created for a project. For more information, see "Setting Parameters in the Resource Handler Code" on page 89.

    You can define any application-specific parameters you might need.

Setting Parameters in the _defaults_.xml and Target Device Files

To add parameters to the project's _defaults_.xml or one of the project's target device files (*device_name* .xml), use the following format:

```
<param name="ParameterName" value="ParameterValue"/>
```

You can also optionally include a comment when setting the parameter:

```
<param name="ParameterName" value="ParameterValue" comment="comment"/>
```

Specifying the Parameter Name

For *ParameterName* specify the name you want to use. The name can be anything that is acceptable as a Java variable.

Specifying the Parameter Value

For *ParameterValue* use one of the formats specified in the following table.

For this type of value...	Use this format to specify the value...
boolean	`boolean={"true" | "false"}` or `bool={"true" | "false"}`
byte	`byte=value`
char	`char=value`
double	`double=value`
float	`float=value`
int	`int=value`
long	`long=value`
short	`short=value`
string	`string=value`

Example

Suppose at run time your application code needs to determine whether to display a launch screen. To set the default for all devices to display the launch screen, you can add a parameter named "DISPLAY_LAUNCH_SCREEN" to the _defaults_.xml file and set its value to "true":

```
<param name="DISPLAY_LAUNCH_SCREEN" bool="true"/>
```

The following shows the resulting parameter declaration in Parameters.java:

```
public static final boolean PARAM_DISPLAY_LAUNCH_SCREEN = true;
```

Then for devices for which you do not want to display the launch screen, you can add the parameter to the target device file for those devices and the parameter value to `false`:

```
<param name="DISPLAY_LAUNCH_SCREEN" bool="false"/>
```

If you build your project for one of the devices for which the launch screen should not be displayed, Mobile Designer reads the target device file, and as a result, uses the following parameter declaration in Parameters.java:

```
public static final boolean PARAM_DISPLAY_LAUNCH_SCREEN = false;
```

Setting Parameters in the Resource Handler Code

If you still code the resource handler for your project, you can add application-specific parameters. Additionally, for each resource you add to the project and to which you assign an ID, an associated parameter is included in the Parameters.java class.

In your resource handler code, to define the parameters, you invoke methods that Mobile Designer provides. The methods are defined in the com.softwareag.mobile.reshandler.AntTaskResourceHandler class. For information about how to use the AntTaskResourceHandler methods in your resource handler, see "Methods that Mobile Designer Provides for the Resource Handler" on page 74.

The following table lists:

- The AntTaskResourceHandler method you use to define a parameter.
- The naming convention that Mobile Designer uses for the parameter names. In the naming conventions, *name* is the name assigned to the parameter and *id* is the identifier assigned to the parameter

Method	Naming convention for added parameter	Description
n/a	PARAM_MD_*name*	Parameters that Mobile Designer defines. These parameters are for internal use. The Mobile Designer run-time classes use these parameters. For a description of the run-time classes, see " Mobile

Method	Naming convention for added parameter	Description
		Designer-Provided Run-Time Classes" on page 49.
setParam	PARAM_*name*	General application-specific parameters that you define.
setResBlockID	RESBLOCKID_*name*	Parameters that define the identifiers for resource blocks that you defined in your project. For more information about resource blocks, see "Using Resource Blocks and Resource Packs" on page 75.
setResID	RESID_*name*	Parameters that define the identifiers for resources (for example, image files, etc.) that you added to your project.
setTextID	TEXTID_*name*	Parameters that define the identifiers for lines of text that you added to your project.

The following list examples:

- If you use the setParam method to add the application-specific parameter "DISPLAY_LAUNCH_SCREEN", Mobile Designer includes the parameter `PARAM_DISPLAY_LAUNCH_SCREEN` in the Parameters.java class.
- If you use the setResID method to assign an audio file the identifier "BEEP", Mobile Designer includes the parameter `RESID_BEEP` in the Parameters.java class.
- If you use the setResBlockID method to assign a resource block the identifier "AUDIO", Mobile Designer includes the parameter `RESBLOCKID_AUDIO` in the Parameters.java class.

Using Parameters in Your Application Code

When creating an application, typically you have common logic that works for all target devices. However, you might require branches in the logic to address the needs of a specific target device. For example, you might need to omit or alter a feature for a target device, or you might need to position an image relative to the screen dimensions for a target device. To accommodate device-specific logic, your application logic can branch based on parameter values that are set using the device profile settings.

To use parameters in your application code, import the com.software.mobile.runtime.Parameters class. You can then easily access parameter

values. For example, if you want to branch logic based on an application-specific "PARAM_DISPLAY_LAUNCH_SCREEN" parameter, you can use a statement like the following:

```
if (Parameters.PARAM_DISPLAY_LAUNCH_SCREEN)
```

You also use parameters that define identifiers for resources and resource blocks to load resources and resource blocks into memory, and unload them from memory, as necessary.

```
ResourceHandler.instance.loadResourceBlock (Parameters.RESBLOCKID_LAUNCHES);

launch_screen_image_id = ImageHandler.instance.loadImageID
                         (Parameters.RESID_LAUNCH_SCREEN_IMAGE, -1, -1, false);

ResourceHandler.instance.unloadResourceBlock (Parameters.RESBLOCKID_LAUNCHES);
```

For more examples about how to use the parameters in application code, review the code provided with sample projects that are provided with Mobile Designer.

Adding Devices to a Mobile Application Project

Devices that a Mobile Application Supports

Mobile Designer has a device database that provides device profiles for many devices. You are encouraged to use the more generic devices and code for multi-resolution apps.

The device profiles are located in the following directory:

Mobile Designer_directory/Devices

In your mobile application project, you reference the subset of the devices your mobile application project will support. The project's target folder contains an XML file for each device that your mobile application supports. Mobile Designer creates the XML file and adds it to the project's target folder when you execute the Add-Handset Ant target to add a device to your project.

Mobile Development will add the devices that you want your application to support to your project when you define launch configurations. For more information, see *webMethods Mobile Development Help*, *Building a Mobile Project*. However, you can continue to add devices using Ant targets as described in "Adding a Device to a Project" on page 91.

Adding a Device to a Project

To add a device to a mobile application project, execute the Add-Handset Ant target from your project. When adding the device, you associate the device with a language or language group.

The Add-Handset Ant target adds an XML file for the device to your project's targets folder. The following is a sample of the target XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
<property name="project.handset.AND_generic_Android3xAPI.langgroups" value="EFIGS"/>
<property name="project.handset.AND_generic_Android3xAPI.mobiledesigner.handsetgroup"
value="AND_generic_Android3xAPI"/>
<target name="-Project.Handset.AND_generic_Android3xAPI">
<!-- properties, parameters and paths -->
<!-- Load the global info for this project and the Mobile Designer          -->
<!-- handset group                                                    -->
<call-mobiledesigner-handset-target handset="${selected.handset}"/>
</target>
</project>
```

To add a device to a project

1. In Software AG Designer, open the project's build.xml file in the Ant view.
   a. In the Project Explorer view, locate the project to which you want to add a device.
   b. Expand the project and drag its build.xml file to the Ant view.
2. In the Ant view, double-click **Add-Handset**.

   Mobile Designer displays an Add Handset dialog.
3. In Add Handset dialog, select the platform on which the device runs from the **Platform Filter** list.
4. Select the manufacturer that makes the device from the **Manufacturer Filter** list.

   When you select a valid platform and manufacturer combination for which Mobile Designer has a device profile, Mobile Designer populates the **Choose your handset** list and displays a default device name in the **Reference Name** field.

   If Mobile Designer does not have a device profile for the platform and manufacturer combination you specified:

   - Ensure you are specifying the information correctly by reviewing the information in the following directory to determine the names of supported devices and platforms:

     Mobile Designer_directory/Devices
5. Select the device you want to add to the project from the **Choose your handset** list.
6. Accept the default name in the **Reference Name** field or update it if you want to use another name.
7. In **Language Groups**, type a semicolon-separated list of language groups to specify the languages that your mobile application will support for the device. You define language groups using the project.langgroup.group_name property.

   For example, if you have defined the language groups `europe` and `asia` and want to specify those language groups, use the following:

```
europe;asia
```

8. Click **Add Handset**.

4 Building Mobile Application Projects Using Jenkins

Getting Started

About the Mobile Suite Jenkins Plugin

Mobile Designer provides a Jenkins plugin to allow for remote-building setups. Jenkins creates an environment for very flexible and configurable build processes, allowing for many different types of tools and setups. This plugin follows on as a replacement to the functionality previously provided with Mobile Administrator. This guide covers important points you should consider when installing and setting up Jenkins, along with the functionality directly provided by Mobile Designer. Some more advanced topics and building setups are also discussed. Advanced functionality and support is available within the Mobile Development perspective of Software AG Designer. The functionality in Software AG Designer is easy to use, and it is recommended to use the Jenkins plugin on a day to day basis within Mobile Development. For more information, see *webMethods Mobile Development Help*.

Jenkins Terminology

If you are not already familiar with Jenkins, the following table explains a few important terms to know when using Jenkins. Further information can be found online.

Term	Explanation
Jenkins Master	The Jenkins master is the term for the computer that manages all Jenkins build activity. It provides a web interface for users to interact with, and it can delegate build activities to other computers (Jenkins slaves). One Jenkins master can be used to manage many slaves.
Jenkins Slave	A computer connected to the Jenkins master. This computer will run build jobs for the master. Slave computers may all be configured the same, or some may be used to provide specific tools (e.g., those that can only run on one operating system, such as Xcode), or different versions of the same tool that will not co-exist on the same computer.
Build Job	A series of build steps executed one after the other in a single workspace in order to do something useful for the developer (e.g., compile an application, generate a test report, etc.).

Term	Explanation
Build Step	This is the smallest unit of work in Jenkins. It comprises a single action (eg., launch Ant/Gradle/Maven with a target, run a batch script, delete a file).
Workspace	A folder in which a Jenkins job runs its build steps. This folder may be on the Jenkins master or one of the Jenkins slaves, and it may be in a different place (or on a different computer) every time the job runs.
Pipeline	A collection of Jenkins jobs that are tied together to perform a larger workflow (e.g., compile > run tests > generate reports). Unlike build steps in a job, jobs in a pipeline can be executing simultaneously.
Build Artifact	The end product of a build job (e.g., an .apk or .app file for building a mobile project).

Requirements

The Mobile Suite Jenkins plugin requires a Jenkins server running at least version 2.7.3 of Jenkins. It has been tested up to version 2.121.1 (the latest long-term support release at the time of writing). The plugin also requires the Jenkins Ant plugin to be installed. Versions 1.5-1.8 are acceptable and have been tested. Jenkins should automatically install this dependency as part of the installation of the Mobile Suite plugin.

Computers acting as Jenkins slaves used for building mobile applications need to be running MacOS El Capitan (10.11), Sierra (10.12) or High Sierra (10.13). Later versions of MacOS are also expected to work.

All Jenkins slave computers must have Software AG Designer with the Mobile Development perspective and Mobile Designer installed and configured, along with any platform SDKs (Android Studio, Xcode) that you want to use.

Note: Single computer configurations are technically possible, if all requirements for the Jenkins slave and the Jenkins server can be met. There are additional requirements and configuration steps for single-computer setups, many of which are covered in this documentation. However, it is not recommended to use single-computer configurations. It is less secure, scales poorly with additional users, and will make the Jenkins web UI less responsive during building.

Important: When installing Mobile Designer, Mobile Development and Platform SDKs on a Jenkins build slave, be aware that the user account doing the installation may not have the same rights and permissions as Jenkins does when

building your application. It may be necessary to install files into "common" locations, and/or to adjust the access properties of files after installation.

Installing and Configuring Jenkins

Installing Jenkins

If you do a new Jenkins install, follow the instructions at "https://jenkins.io/" to download and install your server. Software AG recommend that you choose a version of Jenkins that is part of the stable long-term support branch.

Mobile Designer provides an installation bundle that you must upload to your Jenkins server.

Note: You may need to restart your Jenkins service as part of this process. If your build server is constantly in use, you can check the Jenkins documentation on how to stop the server from accepting new builds (the *Quiet Down* mode), see *Prepare for Shutdown* in the *Manage Jenkins* section.

To install the installation bundle

1. In a web browser, connect and log in to your local Jenkins server (with administrative privileges, if required).
2. Click **Manage Jenkins**, and then click **Manage Plugins**.
3. Select the **Advanced** tab, then scroll down to the **Upload Plugin** section.
4. Click **Browse** and locate the *your_mobile_designer_install* /plugins/Jenkins/v1.0.0/mobile-suite-jenkins-plugin.hpi installation bundle on your machine. Click **Upload**.

The Jenkins server installs the plugin.

Note: You may need to restart your Jenkins server, especially if you are upgrading the plugin from a previous install on an older version of Jenkins. Jenkins will often provide a check box that can be selected as part of the plugin install progress page. You can also use your normal procedure to restart the server. Ensure that all currently running build jobs finish before restarting the server.

Single-Computer Setup for iOS

When setting up a Jenkins master with the intention of building for iOS on the same computer, the Jenkins user that is created will not have a default keychain to start with, and this will break the build process. You have two options: You can either change the user that the Jenkins server runs as (see the Jenkins documentation for this), or you can create the default keychain for the Jenkins user.

Note: When creating .p12 files for a single-computer setup like this, keep in mind that the Jenkins user does not have access to the associated keys you have on other user accounts. These must be included in the .p12 files you export, or placed in the JenkinsDefault keychain by hand.

Creating the Default Keychain for the Jenkins User

To create a default keychain

1. Log in to your computer with a user name and password that have administrator rights.
2. Open a terminal under /Applications/Utilities/Terminal.
3. Enter the following:

```
sudo su Jenkins
your_password
cd ~/Library
mkdir Keychains
cd Keychains
security create-keychain JenkinsDefault.keychain
the_new_password
security default-keychain -s JenkinsDefault.keychain
exit
sudo passwd Jenkins
the_password_for_the_JenkinsDefault.keychain
```

4. Reboot your computer.

Configuring Your Server After Installation

Under the **Manage Jenkins** section, the Mobile Suite Jenkins plugin will create two extra sections. They are used to manage Android and iOS signing keys, and iOS provisioning profiles.

Adding an Ant Installation Through Global Tools

If you are setting up a new server, or have never configured Apache Ant before, you must configure this through the **Global Tools** section.

To add an Ant installation

1. On your Jenkins homepage, select **Manage Jenkins > Global Tool Configuration**.
2. Click **Ant Installations...** to expand the **Global Tool Configuration** section.
3. Click **Add Ant**.
4. In the **Name** field, enter a memorable name (eg., Ant_1_10_1).
5. Ensure that the **Install Automatically** check box is selected.

6. Under **Install from Apache**, select an appropriate version of Ant from the drop-down list for use with Mobile Designer. 1.9.x versions greater than 1.9.4 and 1.10.x versions are expected to be compatible.
7. Click **Save**.

Managing Signing Keys

In order to ensure security for signing credentials, the files stored on the server are encrypted using the Jenkins master node's key. This key is randomly generated during install and will be unique to your server.

Important: Do not use files held on the Jenkins server as your only copies of your private keys. Once they are uploaded to the server, you cannot get them back in a usable form.

Adding an Android Keystore

To add an Android keystore

1. On your Jenkins homepage, select **Manage Jenkins > Mobile Signing Key Configuration**. This page is used to manage iOS private keys and certificates, stored in .p12 files, and Android keystores.
2. At the top of the page, under **Upload Android or iOS Signing Keys**, click **Browse**.
3. Select the Android keystore file to upload, and provide the password used to unlock it.
4. Select **Android** from the platform drop-down list, and then click **Upload**.

   The server will indicate if the file has been uploaded correctly, and it should appear in the **Current Android Keys** section below.
5. In the **Current Android Keys** section, add a meaningful description for this keystore.
6. Provide the **key passwords** for each key alias in the keystore.
7. At the bottom of the **Current Android Keys** section, click **Save**.

Exporting iOS Certificates and Private Keys to .p12 Files

The Jenkins plugin works with .p12 (Personal Information Exchange) files. These password-protected files contain the certificates and private keys that can be used to sign your build. .p12 files can be created by exporting certificates that are already installed on a Mac.

To export certificates and private keys to .p12 files

1. Open the **Keychain Access** application from /Applications/Utilities/Keychain Access.

2. Locate the certificates you want to export. The view can be filtered by selecting the **Certificates** or **MyCertificates** category.

   Tip: Certificates for signing iOS builds will likely start with "iPhone Developer:", "iOS Distribution:", or similar.

3. Each certificate you want to export will have a private key associated with it (indicated with a small grey triangle to the left of the certificate's icon). Click this triangle to reveal the key.
4. Press SHIFT+CLICK or COMMAND+CLICK to select all the certificates you want to export and their private keys.
5. Right-click the selection, and select **Export *n* Items**, where *n* is the number of items to export.
6. Select an appropriate name and location for the file, and make sure the file format is **Personal Information Exchange (.p12)**. Then click **Save**.

You will be prompted for a password to lock this file with. Make a note of this password as you will need it again when the file is imported to your Jenkins server.

Adding an iOS .p12 File

To add an iOS .p12 file

1. On your Jenkins homepage, select **Manage Jenkins > Mobile Signing Key Configuration**.
2. At the top of the page, under **Upload Android or iOS Signing Keys**, click **Browse**.
3. Select the iOS .p12 file to upload, and provide the password used to unlock it.
4. Select **iOS** from the platform drop-down list, and then click **Upload**.

   The server will indicate if the file has been uploaded correctly, and it should appear in the **Current iOS Keys** section below.
5. In the **Current iOS Keys** section, add a meaningful description for this key.
6. At the bottom of the **Current iOS Keys** section, click **Save**.

Uploading Provisioning Profiles

To upload a provisioning profile

1. On your Jenkins homepage, select **Manage Jenkins > Mobile Provisioning Profiles iOS**.
2. At the top of the page, under **Upload iOS Provisioning Profiles**, click **Browse**.
3. Select the provisioning profile file to upload, and then click **Upload**.

The server will indicate if the file has been uploaded correctly, and it should appear in the **Current Provisioning Profiles** section below.

Advanced Settings

About

Jenkins is highly configurable, and the Mobile Suite plugin allows advanced users to create build jobs and pipelines that work within their existing processes and infrastructure.

New Build Steps Provided by the Mobile Suite Plugin

The Mobile Suite Jenkins plugin provides four build steps that can be added to Jenkins jobs. A small description of each is provided here. More help and information can be found through the Jenkins web UI itself when interacting with the build steps in a job.

- **Generate Mobile Designer Sources** This build step is the equivalent of the menu option **Generate Source Code > Application Model and API** in the Mobile Development perspective of Software AG Designer. Using the application model, it prepares all resources needed and creates the contents of the gen folder.
- **Decompress Mobile Designer Project** This build step is designed to take a Mobile Suite project bundle (usually with the name "project.bundle.zip") that was passed as a parameter to the Jenkins job and decompress it. Optionally, it can check the project bundle for metadata about the project and remove any old copies of that folder from the workspace before decompressing it.
- **Auto-Build Mobile Designer Project** This build step uses the metadata that comes from a Mobile Suite project bundle to find out which build was specified by the user in Software AG Designer. It will then build them.
- **Build Mobile Designer Project** This build step allows for more direct control over the parameters used to build the mobile application. You can use it if you want to build your application after checking out from source control.

  You can specify the handset target, version number, language, etc. This build step can also be useful if you want to build multiple handset targets in the same job, as this build step may be repeated as many times as required.

Creating New Build Jobs

The automatic creation of build jobs through Jenkins with Mobile Development will be adequate for most use cases. However, you have a few additional options that may be more appropriate for your development cycle.

Using the Jenkins-Create-Job Ant Task

You can use the Jenkins-Create-Job Ant task from any mobile project that supports the Jenkins plugin.

You must supply values for the following Ant properties:

Property	Description
`jenkins.server`	Required. The root page of the server to connect to, eg: "http://127.0.0.1:8080".
`jenkins.job.name`	Required. The name of the Jenkins job to create. If the name is already taken, the process will not override it.
`jenkins.project.root.folder`	Required. The name of the project's folder, e.g., "_NativeUIDemoNew_", "Cocktails".
`jenkins.username`	Optional. The user name used to connect to the Jenkins server. This user must have permission to create Jenkins jobs.
`jenkins.password`	Optional. The password used to connect to the Jenkins server. If `jenkins.username` is not defined, this property will be ignored.
`jenkins.sag.install.location`	Optional. If defined, this will instruct Jenkins to look in this location for the Software AG install to use (e.g., "/Applications/SoftwareAG103"). If this is not defined, then the default install location will be used.
`jenkins.sign.ios`	Optional. Signs any iOS builds that this Jenkins job does with a key. If set to `true`, then `jenkins.ios.bundle.id` must be specified.
`jenkins.ios.bundle.id`	Optional. Defines the full iOS bundle ID for this application. Wildcards are *not* allowed here.
`jenkins.set.android.package`	Optional. If set to `true`, the Android package is overriden, and `jenkins.android.package` must be specified.

Property	Description
`jenkins.android.package`	Optional. Defines a java package to create your Android build in (e.g., "com.softwareag.mobile.someproject").
`jenkins.needs.xss.protection`	Optional. Normally, Mobile Designer automatically detects if your server uses Cross Site Request Forgery Protection. However, this may not be possible in all cases. Set to `true` if your server uses Cross Site Request Forgery Protection, but Mobile Designer cannot detect it.
`jenkins.with.autogeneration.step`	Optional. If set to `false`, the created build job will omit the **Generate Mobile Designer Sources** build step and assume that they are to be uploaded in the project bundle.
`jenkins.force.build.node.to`	Optional. Forces the Jenkins job to run only on certain build nodes. Specify the name of a Jenkins node or a label pointing to one (or more) build nodes. The default is `""`.

Building Mobile Applications From Source Control

You may want to build your application directly from source control, without user intervention, for example, as a common "nightly" build for testing, or when making a clean deployment to the App Store. In these cases, a build job can be created manually to suit your needs.

Note: As requirements and infrastructure can vary greatly between organisations, this section is not a direct step-by-step set of instructions of what must be done to get your application built. It provides as many common details and useful hints as possible so that you can work with your existing infrastructure to get a result that works well for you.

First, check out your source into a folder into the workspace, setting things up so that the project is held directly in a folder named after the project. (e.g., for a project called "MyMobileProject", you want to have *Jenkins_workspace_root* /MyMobileProject with build.xml, src/, model/, etc. inside.). This step will vary a lot depending on your source-control system, how you connect to it, and which Jenkins plugin(s) you are using.

Tip: If your source control system sets files to read-only unless they are marked as open for edit (e.g., Perforce), then you may need to unlock some files for edit and revert them after the build process. To minimise this, it is recommended

that the gen, _temp_, .launch and Builds folders, along with the .classpath file are kept out of source control, as these folders will often change during the build and/or activate processes. As long as this rule is adhered to, read-only files should not be a problem with most mobile projects.

After the project source is checked out, in most cases you will want to generate the mobile sources, using the model to create the gen folder and all resources, source, etc. inside. This is done using the **Generate Mobile Designer Sources** build step. Enter the project's name (e.g., "MyMobileProject") and optionally specify the location of your Software AG install on the slave computer that you want to build with. If no value is given, Jenkins will try the default location.

After source code is generated, the next build step is **Build Mobile Designer Project**. This step allows you to specify the handset target, version, language, and so on. This step may be repeated as many times as required to create the builds that are needed.

Tip: Creating more than one build in the same build job can be convenient. However, be aware that if one build fails, the Jenkins job will stop at that point. It can often be better to start by creating one Jenkins job, then copy it and make small changes to get other builds.

When all builds are finished, it is important to archive the built files in a post-build action. In most cases, you will have to capture the contents of the *your_project* /Builds folder.

Using the Jenkins-Multi-Build Ant Task

You can start a Jenkins remote multi-build process by calling the Jenkins-Multi-Build Ant task from any mobile project that supports the Jenkins plugin.

The Jenkins-Multi-Build Ant task has similar properties as the +Target-Build Ant task, with a few additions. You must supply values for the following Ant properties:

Property	Description
`handset`	Required. The name of the target to build (e.g., "Android_generic_AndroidWVGA800"). These correspond to the file names in your project's targets folder.
`langgroup`	Required. The language group to build for. Most mobile projects developed with the Mobile Development perspective of Software AG Designer will use "I18N" here.
`target`	Required. The type of build to create. For Android builds, this can only be `release` or

Property	Description
	`debug`. For iOS builds, this can be `appstore`, `enterprise`, `ad hoc`, `sim release`, `dev release`, or `xcode project`.
`version`	Required. The version number of the application to build, e.g. "1.3.2".
`jenkins.server`	Required. The root page of the server to connect to, eg: "http://127.0.0.1:8080".
`jenkins.job.name`	Required. The name of the Jenkins job to execute to create this build.
`jenkins.username`	Optional. The user name used to connect to the Jenkins server. This user must have permission to create Jenkins jobs.
`jenkins.password`	Optional. The password used to connect to the Jenkins server.
`jenkins.bundle.zip.includes`	Optional. A semicolon delimited list of files and folders to include in the bundle passed to the server. The default value is `""`, which will upload everything that is not excluded or ignored.
`jenkins.bundle.zip.excludes`	Optional. A semicolon delimited list of files and folders to exclude from the bundle passed to the server. The default is `_temp_;bin;Builds`.
`jenkins.bundle.zip.ignores`	Optional. A semicolon delimited list of files and folders to ignore when building the bundle for the server. This list is intended for use to exclude source control metadata files (e.g., .svn folders). The default value is `.svn`.
`jenkins.follow.console`	Optional. If set to `true`, the build process will not detach from the Jenkins server after the build has started. Console output from the Jenkins build job will be piped through to the local console. The default is `true`.

Property	Description
`jenkins.needs.xss.protection`	Optional. Normally, Mobile Designer automatically detects if your server uses Cross Site Request Forgery Protection. However, this may not be possible in all cases. Set to `true` if your server uses Cross Site Request Forgery Protection, but Mobile Designer cannot detect it.
`jenkins.skip.code.generation`	Optional. If set to `true`, the created build job will omit the **Generate Mobile Designer Sources** The default is `false`.

5 Building and Compiling Mobile Application Projects Using Ant Targets

Build Process Overview

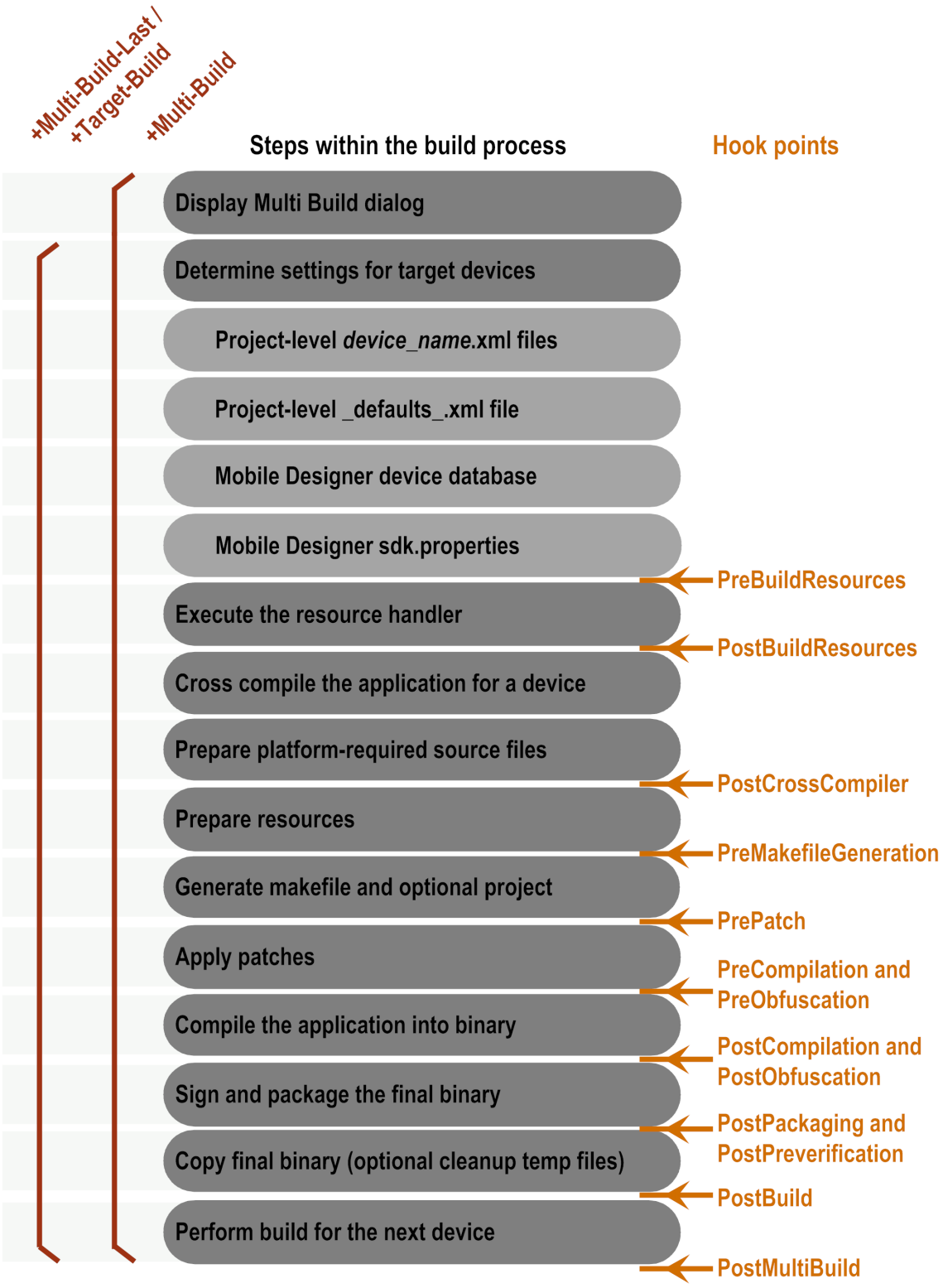
Build Ant Target Summary

The following diagram provides summary information about the process Mobile Designer performs to build a mobile application project. It shows:

- The names of the Ant targets you can use to build a mobile application project.
- The steps Mobile Designer performs in the build process. For details about each step, see "Steps in the Multi-Build Process" on page 112.
- The hook points in the build process where Mobile Designer can run custom Ant scripts that you provide. For more information about hook points, see "Creating Custom Ant Scripts to Run at Predefined Hook Points" on page 125.

Note: For instructions on how to perform a build of your mobile application project, see "Building Mobile Applications" on page 117.

Multi-Build Ant Targets



Steps in the Multi-Build Process

This section describes the steps that Mobile Designer performs when you run the +Multi-Build, +Multi-Build-Last, or +Target-Build Ant targets to build a mobile application project.

Note: You can use Mobile Development to execute a local multi-build or a remote Jenkins build. For more information, see *webMethods Mobile Development Help*, *Building a Mobile Project*.

The table also indicates hook points where Mobile Designer can run custom Ant scripts that you provide. For more information about hook points, see "Creating Custom Ant Scripts to Run at Predefined Hook Points" on page 125.

Note: For instructions on how to perform a build of your mobile application project, see "Building Mobile Applications" on page 117.

1 Display the Multi Build dialog

Mobile Designer displays the Multi Build dialog.

Note: Mobile Designer only displays the Multi Build dialog when you run the +Multi-Build Ant target.

2 Determine settings for target devices

Mobile Designer determines the settings for the device for which it is building the application. It retrieves the settings from the following sources in the order listed.

- Project-level *device_name* .xml files
- Project-level _defaults_.xml file
- Mobile Designer device database
- Mobile Designer sdk.properties file

Mobile Designer uses the first setting it encounters. For example, if Mobile Designer encounters a setting in the project-level target *device_name* .xml file and then again in the project-level _defaults_.xml file, Mobile Designer uses the setting from the target *device_name* .xml file.

For more information about:

- Project-level device files, see "Where You Set Properties" on page 84 and "Setting Project Properties" on page 86
- Mobile Designer device database, see "Devices that a Mobile Application Supports" on page 91

- Mobile Designer sdk.properties file, see "Mobile Designer Configuration Properties (sdk.properties)" on page 18

hook point **PreBuildResources**

If you have created an Ant script to run at the PreBuildResources hook point, Mobile Designer runs the Ant script.

3 **Execute the resource handler**

Mobile Designer runs the resource handler that you created for the project. When running the resource handler, Mobile Designer records all the resources required for your application. Mobile Designer also creates the class. For more information about creating the resource handler, see "Defining Resources for a Mobile Application Project" on page 73. For more information about the Parameters class, see "Application and Parameter Classes" on page 50.

At this point in the build process, Mobile Designer uses the following project properties:

- project.reshandler.src.path for the location of the project's resource handler script and any associated classes
- project.java.reshandler.name for the name of the resource handler class you created for your project.
- project.resource.dir.root for the location of the top-level folder that contains the resources (image files, etc.) for your project
- project.reshandler.additional.libs.path for the location of additional libraries that your project's resource handler requires

For more information about these properties, see "Resource Handler Properties" on page 197.

hook point **PostBuildResources**

If you have created an Ant script to run at the PostBuildResources hook point, Mobile Designer runs the Ant script.

4 **Cross compile the application for a device**

Mobile Designer copies your project's application code for the target device into the project's _temp__src_ folder. At this point in the build process, Mobile Designer uses the following project properties:

- project.runtime.project.src.path for the location of your project's application code
- project.runtime.additional.classes.path for the location of additional precompiled classes that your application requires

- project.runtime.additional.stubs.path for the location of additional stubs to use when compiling your application code

For more information about these properties, see "Run-Time Code Compilation Properties" on page 203.

Mobile Designer converts the mobile application source code into a neutral language, for example, C++ or Java. The neutral language that Mobile Designer uses is the required neutral language for the target device. You can set the following code conversion properties to customize how Mobile Designer converts the application source code:

- cpp.no.selfprotect for whether to add a safeguard to every method call
- cross.compiler.extractinners for whether to extract inner classes included in the Java source code
- cross.compiler.nodatestamp for whether to include timestamps at the top of every generated C++ (CPP) and H file
- cross.compiler.render.selfprotect for whether to add an extra line at the top of every non-static method
- java.parser.retain.comments for whether to retain comments when compiling

For more information about these properties, see "Code Conversion Properties" on page 167.

The neutral language code that Mobile Designer generates from converting your application code is easy-to-read and, by default, retains comments from the original Java source code.

Note: When Mobile Designer generates C++ code for C++-based platforms (for example, iOS), the C++ code it generates will be the same format for all C++-based platforms. The generated C++ code is uniform, ANSI-compatible, and compiles against all target architectures (such as x86, MIPS, and ARM processors).

5 Prepare platform-required source files

Mobile Designer gathers other source files that it requires to make the final binary. For example, additional platform-specific files might be needed for the final binary.

hook point PostCrossCompiler

If you have created an Ant script to run at the PostCrossCompiler hook point, Mobile Designer runs the Ant script.

6 Prepare resources

Mobile Designer gathers the resources that the project requires and prepares them for building the final binary. These includes, for example, the resources defined by your project's resource handler, NativeUI resources, and resources that the specific platform might require.

hook point **PreMakefileGeneration**

If you have created an Ant script to run at the PreMakefileGeneration hook point, Mobile Designer runs the Ant script.

7 **Generate makefile and optional project**

Mobile Designer creates the makefile that defines everything that is required to create the final binary.

Additionally, if appropriate for the target platform, Mobile Designer creates the platform-specific project. For example, when building an application that runs on the iOS platform, Mobile Designer creates an Xcode project or when building an application that runs on the Android platform, it creates an Eclipse project.

You can set the cross-compiler properties that affect this step in the build process:

- project.handset.custom.stubfolder for whether to override the makefile and project template files that are provided with Mobile Designer with makefile and project template files that you supply
- Several properties that customize the generation of the makefile. See "Makefile Additions" on page 175.

hook point **PrePatch**

If you have created an Ant script to run at the PrePatch hook point, Mobile Designer runs the Ant script.

8 **Apply patches**

If you created patches, Mobile Designer applies the patches to the cross-compiled code. For more information about when to use patches and how to create patches, see "Creating Patch Files to Apply to the Cross-Compiled Code" on page 129.

Note: If you set the project.handset.skip.compilation property to `true`, Mobile Designer skips the following actions.

hook point **PreCompilation and PreObfuscation**

If you have created Ant script(s) to run at the PreCompilation and/or PreObfuscation hook points, Mobile Designer runs the Ant script(s).

9 Compile the application into binary

Mobile Designer executes the makefile it created in a previous step to create binary for the target device. During this step, Mobile Designer requires the platform-specific SDKs that you installed when setting up the platform. For more information, see "Setting Up Platforms" on page 27.

hook point PostCompilation and PreVerification

If you have created Ant script(s) to run at the PostCompilation and/or PreVerification hook points, Mobile Designer runs the Ant script(s).

10 Sign and package the final binary

Mobile Designer packages the output of the build so that is ready for installation on the target device. The packaging process varies from platform to platform. For example, for Android builds, the output is in an application package (apk) file.

Additionally, if required by the target platform, Mobile Designer signs the final build.

hook point PostPackaging and PostVerification

If you have created Ant script(s) to run at the PostPackaging and/or PostVerification hook points, Mobile Designer runs the Ant script(s).

11 Copy final binary (optional cleanup temp files)

When Mobile Designer builds your project, it adds a Builds folder to your project folder. Within the Builds folder, Mobile Designer creates a folder for the device for which it is building your application:

project /Builds/*x.y.z* /*device*

In the folder location above:

- *project* is the name of your project.
- *x.y,z* is the version number you specified for the build.
- *device* is the name of the device for which Mobile Designer is building your application.

The *device* folder contains a _temp_ folder. Mobile Designer stores temporary files in the _temp_ directory while it is building the application. After it creates the final binary, Mobile Designer copies the final binary from within the _temp_ folder into the *device* folder.

Mobile Designer then optionally performs cleanup. If you clear the **Retain output build files** check box in the Multi Build dialog when you started the build, Mobile Designer cleans up the temporary files by deleting the _temp_ folder.

Note: If you set the project.handset.skip.compilation property to `true`, Mobile Designer resumes processing with the following actions.

hook point **PostBuild**

If you have created an Ant script to run at the PostBuild hook point, Mobile Designer runs the Ant script.

12 **Perform build for the next device**

If you select multiple devices/language combinations in the Multi Build dialog to have Mobile Designer build your application for multiple device/language combinations, Mobile Designer restarts the build process at step 2 for the next device/language to build.

hook point **PostMultiBuild**

If you have created an Ant script to run at the PostMultiBuild hook point, Mobile Designer runs the Ant script.

Building Mobile Applications

About Building a Mobile Application Project

Before you can build your project, you must ensure you have completed the required prerequisite tasks. For more information, see "Before You Can Build a Mobile Application Project" on page 118.

To build your project, including compiling your project into binary and packaging your application for target devices, use the following Ant Targets:

- +Multi-Build to build your project for multiple device/language combinations. For more information, see "Building a Project for Multiple Target Devices" on page 120.
- +Multi-Build-Last to rebuild your project for the last target device. For more information, see "Building a Project for the Last Target Devices" on page 121.
- +Target-Build to build your project from the command line. For more information, see "Building a Project from the Command Line" on page 121.

If you want, you can use Mobile Designer to convert your mobile code into a neutral language (for example, C++ or Java), but then use your own platform-specific, native

tools to compile your project into binary and package the application for the target device. For more information, see "Using Native Tools to Create the Final Binary" on page 123.

For details about the actions that Mobile Designer performs to build your project, see "Build Process Overview" on page 110.

Before You Can Build a Mobile Application Project

Before you build your mobile application project, be sure you have completed the following tasks:

- Perform platform-specific setup on your machine

  You must have the proper setup for the target platform(s). The setup requires that you have items such as compilers and SDK installed on your machine. For example, if you want to build an Android application, you must have the Android SDK installed on your machine.

  In addition to installing platform-specific tools, you must also set properties in the Mobile Designer sdk.properties file to indicate where the SDKs, compilers, and other tools are located.

  For information about how to set up your platform, see "Setting Up Platforms" on page 27.

- Set required project properties

  Mobile Designer requires that you set some project properties for your project. For example, you must set the `project.runtime.project.src.path` property to specify the location of your project's run-time code. For a list of the required project properties, see "Setting Project Properties" on page 86.

  For information about iOS-specific settings, see "Setting Properties to Build for an iOS Device" on page 118.

  For information about Android-specific settings, see "Setting Properties to Build for an Android Device" on page 119.

Setting Properties to Build for an iOS Device

To make AppStore, AdHoc, or Dev builds, you must provide an appropriate provisioning profile. The following properties must be specified in your _defaults_.xml for the different builds:

- **AppStore**

```
<property name="ios.provisioning.profile.appstore"
 value="path_to_appstore_provisioning_profile"/>
```

- **Enterprise** (Select **AppStore** in the Multi Build dialog.)

```
<property name="ios.provisioning.profile.appstore"
 value="path_to_appstore_provisioning_profile"/>
```

```
<property name="ios.appstore.method" value="enterprise"/>
```

- **AdHoc**

```
<property name="ios.provisioning.profile.adhoc"
 value="path_to_adhoc_provisioning_profile"/>
```

- **Dev**

```
<property name="ios.provisioning.profile.appstore"
 value="path_to_appstore_provisioning_profile"/>
```

You can either specify the path to the provisioning profile, or you can specify the name, ID, or AppID of the provisioning profile. In the latter case, the search pattern you specified is applied to all provisioning profiles installed on your machine. For a successful build, exactly one match should be found.

You can check your search pattern using an Ant task. The following example shows an ID search pattern:

```
ant check -Dpprofile=4dc3a740-b922-4fe2-8529-595eecddb6bc
```

If a provisioning profile with this ID exists, the returned code looks similar to the following example:

```
[checkFeature] Searching for provisioning profiles in /Users/your_user_name/
 Library/MobileDevice/Provisioning Profiles
[checkFeature]   Path       : /Users/your_user_name/Library/MobileDevice/
 Provisioning Profiles/4dc3a740-b922-4fe2-8529-595eecddb6bc.mobileprovision
[checkFeature]   AppID      : U6GS52HNZ5.com.softwareag.mobile.nativeuidemonew
[checkFeature]   ID         : 4dc3a740-b922-4fe2-8529-595eecddb6bc
[checkFeature]   Name       : nativeuidemonew.adhoc.profile
[checkFeature]   Method     : ad-hoc
[checkFeature]   Expires on : 2019-03-02T09:38:20Z
[checkFeature]   Identity   : iPhone Distribution: Software AG
```

To perform a build, the signing identity must be installed on your system. If it is not installed yet, follow the instructions as described in "Importing the Signing Environment from Another Macintosh" on page 35 or "Creating a New Signing Environment" on page 37.

Alternatively, if you use Xcode 8 and later, you can use the auto signing feature of Xcode by defining the following properties:

```
<property name="ios.xcode.managed.signing" value="true"/>
<property name="ios.xcode.managed.signing.team.uid" value="your_team_id"/>
```

For the team ID, see "https://developer.apple.com/account/#/membership".

Setting Properties to Build for an Android Device

An optional property file can be provided to sign for an Android device. If it exists, it should point to a text file stored under *project_root* /android_signing.properties that contains the following four properties in standard `key=value` Java properties format (see the format used for sdk.properties). This file can be kept out of source control to protect passwords and keyfiles.

- `android.release.keystore.file` Location of the keystore file. It can either be relative or absolute to the project's root directory. Use "/" for path separators, even

on Windows. This file can be kept out of source control to protect passwords and keyfiles.

- `android.release.keystore.password` Passphrase used to unlock the keystore.
- `android.release.key.alias` Alias of the key for signing this build.
- `android.release.key.password` Password used with this key alias.

Building a Project for Multiple Target Devices

To build your mobile application project, you use the +Multi-Build Ant target. The +Multi-Build Ant target allows you to build your project for multiple device/language combinations.

For information about the actions that Mobile Designer performs to build your project, see "Build Process Overview" on page 110.

Note: Before you build your project, be sure you have completed the required setup. See "Before You Can Build a Mobile Application Project" on page 118.

To build a project for multiple targets

1. In Software AG Designer, open the project you want to build.
2. Click the Project Explorer view, expand the project, and drag the build.xml file to the Ant view.
3. In the Ant view, double-click **+Multi-Build** to launch the Multi Build dialog.
4. In the Multi Build dialog, select the device/language combinations for which you want to build the project.
5. In the **Version** field, type the version number for the build.
6. Select or clear the **Retain output build files** check box.
   - Select the check box to retain the files from the build.

     Mobile Designer retains the cross-compiled code it generated from your original Java code, along with any project (for example, Xcode project for iOS) it might have generated.

     Choose to retain the files if you need to create a patch for your project, if you want to save the output for subsequent testing, or if you want to use native platform tools to compile to create the final binary for your project.
   - Clear the check box if you do not need the files from the build.
7. Click **Multi Build**.

Building a Project for the Last Target Devices

If you want to rebuild your project for the last target devices, use the +Multi-Build-Last Ant task. Mobile Designer rebuilds the project for all the devices you selected the last time you used the Multi-Build dialog.

For information about the actions that Mobile Designer performs to build your project, see "Build Process Overview" on page 110.

Note: Before you build your project, be sure you have completed the required setup. See "Before You Can Build a Mobile Application Project" on page 118.

To build a project for the last target

1. In Software AG Designer, open the project you want to build.
2. Click the Project Explorer view, expand the project, and drag the build.xml file to the Ant view.
3. In the Ant view, double-click **+Multi-Build-Last**.

   Mobile Designer does not display a dialog. Mobile Designer rebuilds the project for the last target device that was built, using the settings from the last build, including any setting you might have made in a custom JPanel.

Building a Project from the Command Line

You can build your project from the command line using the +Target-Build Ant target. When using the +Target-Build Ant target, you can build for only a single device/language combination at a time. To execute the Ant target, from the command line navigate into the project's folder.

For information about the actions that Mobile Designer performs to build your project, see "Build Process Overview" on page 110.

Note: Before you build your project, be sure you have completed the required setup. See "Before You Can Build a Mobile Application Project" on page 118.

Syntax

```
version=x.y.z [retain={true | false}] handset=device langgroup=language
platform=platform target=executable_type +Target-Build
```

Options

Option	Description
`version=x.y.z`	Required. Specifies the version number for the application your are building.
`[retain={true | false}]`	Optional. Specifies whether you want Mobile Designer to retain the cross-compiled code it generated from your original Java code, along with any platform-specific project it might have generated The default is `true`.
`handset=device`	Required. Specifies the device for which you want Mobile Designer to build the application.
`langgroup=language`	Required. Specifies the language(s) for which you want Mobile Designer to build the application.
`platform=platform`	Required. Specifies the platform for which you want Mobile Designer to build the application.
`target=executable_type`	Required. Specifies the type of executable you are building. For example, you might specify `release` or `debug`.
`pprofile`	Required. Defines the path to the provisioning profile. The build type (eg., AppStore, Enterprise, AdHoc, or Dev) is deduced automatically. Alternatively, you can specify the name, ID, or AppId of the provisioning profile, see "Setting Properties to Build for an iOS Device" on page 118. **Note:** The appropriate signing identity must be installed on your system and be able to perform the application signing.
`keychain`	Optional. Defines the path to the keychain.
`buildnumber`	Optional. Specifies the specific build number.

Option	Description
`extraXcodeArgs`	Optional. Defines extra parameters for an Xcode build by generating an Xcode archive.

Example

The following code sample shows how to build version 2.0.3 of your project for the IOS_Apple_iPhone5 device, which runs on the iOS platform, for the language group EFIGS to create an executable version for the Apple App Store.

```
ant +Target-Build -Dversion=2.0.3 -Dbuildnumber=99 -Dretain=true
-Dhandset=IOS_Apple_iPhone5 -Dlanggroup=EFIGS -Dplatform=IOS
-Dtarget=appstore -Dpprofile="~/provisioning_profiles/appstore.mobileprovision"
```

Because `retain` is set to `true`, Mobile Designer retains the cross-compiled code it generated from your original Java code, along with any project (for example, Xcode project for iOS) it might have generated.

Using Native Tools to Create the Final Binary

If you want to use platform-specific, native tools to create the final binary and package your application, you can use the Mobile Designer+Multi-Build or +Target-Build Ant target to build your project, but have Mobile Designer skip compiling your project into binary and packaging your application for the target device. You can move the cross-compiled code and platform-specific project that Mobile Designer creates to the platform-specific tool to create the final binary.

For information about the actions that Mobile Designer performs to build your project, including the actions that Mobile Designer skips when you perform the following procedure, see "Build Process Overview" on page 110.

Note: Before you build your project, be sure you have completed the required setup. See "Before You Can Build a Mobile Application Project" on page 118.

To use native tools to create the final binary for your project

1. Set the project.handset.skip.compilation property to `true` to indicate that you want Mobile Designer to skip compiling your project into binary and packaging your application for the target device. For instructions for how to set a property, see "Setting Project Properties" on page 86.
2. Start the build using one of the following Ant targets:
   - +Multi-Build Ant target. For instructions, see "Building a Project for Multiple Target Devices" on page 120.

     Be sure you select the **Retain output build files** in the Multi Build dialog.

- +Target-Build Ant target. For instructions, see "Building a Project from the Command Line" on page 121.

  Be sure you set the `retain` option to `true`.

3. After the build is complete, locate the cross-compiled code and platform-specific project.

   When Mobile Designer builds your project, it adds a Builds folder to your project folder. Within the Builds folder, Mobile Designer creates a _temp_ folder for each device. The cross-compiled code and the platform-specific project reside within the _temp_ folder:

   project /Builds/*x.y.z* /*device* /_temp_

   In the folder location above:

   - *project* is the name of your project.
   - *x.y,z* is the version number you specified for the build.
   - *device* is the name of a device for which Mobile Designer built your application.

   You can find the cross-compiled code and the platform-specific project in the _temp_/_*language* _ folder. If you created a patch file for the application, you can find the patched versions in the _temp_/_*language* _edit_ folder. For example, for an Android build, you can find the cross-compiled code in the _temp_/_java_ folder, and if a patch was applied, the patched version of the code is in the _temp_/_java_edit_ folder.

4. Copy the files you need to create the final binary to the platform-specific, native tool. Then use the native tool to create the final binary.

Generating Javadocs for a Project

Use the Generate-Javadoc Ant target to generate javadocs for a project. The javadocs that the Generate-Javadoc Ant target generates are primarily javadocs for the customer's project codebase. The javadocs also include Mobile Designer run-time methods and constants.

When you use the Generate-Javadoc Ant target, Mobile Designer adds a _temp_ folder to your main project folder that contains the generated files.

To generate javadocs for a project

1. In Software AG Designer, in the Project Explorer view, locate the project for which you want to generate javadocs.
2. Expand the project and drag its build.xml file to the Ant view.
3. In the Ant view, double-click **Generate-Javadoc**.

   Mobile Designer displays the Activate Handset dialog.

4. In the Activate Handset dialog, select the device for which you want generate javadocs and the language group. Then click **Activate Handset**.

Mobile Designer places the results of the Generate-Javadoc Ant target in your project's _temp_ folder.

Customizing the Build Process

About Customizing the Build Process

Mobile Designer provides the following features that you can use to customize the standard build process:

- **Custom JPanels.** Deprecated. (The feature will not be supported as of Mobile Designer version 10.5.)
- **Hook points.** You can create custom Ant scripts that you want Mobile Designer to run during the build process. Mobile Designer defines various points, called *hook points*, in the build process where it can run an Ant script that you provide. For more information, see "Creating Custom Ant Scripts to Run at Predefined Hook Points" on page 125.
- **Patch files.** You can create patch files that Mobile Designer applies to the cross-compiled code. Use patch files to correct issues that might prevent the cross-compiled code from compiling successfully. Additionally, some times the code might cross-compile successfully, but the resulting code might not execute as expected. You might be able to correct the issue with a patch file. For more information, see "Creating Patch Files to Apply to the Cross-Compiled Code" on page 129.

Creating Custom Ant Scripts to Run at Predefined Hook Points

To customize the standard build process that Mobile Designer performs, you can create custom Ant scripts that Mobile Designer runs at various predefined points, called *hook points*. For example, to automatically upload a build to an FTP server once the build is created, you can use the PostBuild Hook Point, which Mobile Designer runs after it completes a build.

Mobile Designer provides several predefined hook points in the build process where you can inject a custom Ant script to perform custom actions. For example, you can have Mobile Designer run a custom Ant script before it runs the resource handler, or you can have Mobile Designer run a custom Ant script after it runs the resource handler. For more information about the hook points you can use and when each occur in the

build process, see "Hook Point Reference" on page 126 and "Ant Target Summary" on page 205.

To run a custom Ant script at a hook point:

- You must add the associated hook point property to your project's _defaults_.xml file. You use the property to provide the name of the custom Ant script that you want Mobile Designer to run. For information about the properties, see "Hook Point Properties" on page 190.
- You must create the custom Ant script. You can include the Ant script in your _defaults_.xml file directly after the property declaration.

  Ant properties, parameters, and paths set in your custom Ant scripts do not flow through into the remaining build process. Your custom Ant scripts are separate Ant target calls that branch off outside the normal build process flow.

Caution: Although invoking custom Ant scripts at the predefined hook points adds flexibility to the build process, be aware that Mobile Designer does *not* perform safety checking on custom Ant scripts before executing them. You should thoroughly test your custom Ant script's functionality before integrating it into Mobile Designer.

Note: Use of hook points is optional. You can use none, one, or multiple hook points. Mobile Designer only attempts to run a custom Ant script at a hook point if your project's _defaults_.xml file contains a hook point property that provides the name of a custom Ant script.

For an example that illustrates how you use hook points, see the NativeUI PDF Demo sample project.

Hook Point Reference

The following sections describe the hook points that you can use for a project. Each section lists:

- When the hook point occurs in the build process
- Property that you specify in your project's _defaults_.xml file to provide the name of the custom Ant script you want Mobile Designer to run at the hook point
- Names of the Mobile Designer Ant targets that use the hook point

To see a diagram that shows where the hook points are in the build process, see "Ant Target Summary" on page 205.

PreBuildResources Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script before it runs your project's resource handler.

Property	project.hookpoint.target.prebuildresources
Ant targets	+Run-Reshandler +Multi-Build +Multi-Build-Last +Target-Build ++Activate-Handset + +Re-Activate-Handset ++Run-Phoney-With-Re-Activation

PostBuildResources Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script after it runs your project's resource handler.

Property	project.hookpoint.target.postbuildresources
Ant targets	+Run-Reshandler +Multi-Build +Multi-Build-Last +Target-Build ++Activate-Handset + +Re-Activate-Handset ++Run-Phoney-With-Re-Activation

PostCrossCompiler Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script after it converts your project's source to the build's target language.

Note: Mobile Designer only calls this hook point for cross-compiled builds.

Property	project.hookpoint.target.postcrosscompiler
Ant targets	+Multi-Build +Multi-Build-Last +Target-Build

PreMakefileGeneration Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script before it generates the makefile and associated Microsoft Visual Studio or Apple Xcode project for a build.

Note: Mobile Designer only calls this hook point for cross-compiled builds.

Property	project.hookpoint.target.premakefilegeneration
Ant targets	+Multi-Build +Multi-Build-Last +Target-Build

PrePatch Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script before it applies the patches to your code.

Note: Mobile Designer only calls this hook point for cross-compiled builds.

Property project.hookpoint.target.prepatch

Ant targets +Multi-Build +Multi-Build-Last +Target-Build

PreCompilation Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script before it performs platform-specific compilation for each build.

Property project.hookpoint.target.precompilation

Ant targets +Multi-Build +Multi-Build-Last +Target-Build ++Activate-Handset ++Re-Activate-Handset ++Run-Phoney-With-Re-Activation

PostCompilation Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script after it performs platform-specific compilation for each build.

Property project.hookpoint.target.postcompilation

Ant targets +Multi-Build +Multi-Build-Last +Target-Build

PostPackaging Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script after it generates the platform-specific build bundle. After executing this hook point, Mobile Designer performs code signing, if any, that is appropriate to the platform being built.

Note: Mobile Designer only calls this hook point for cross-compiled builds.

Property project.hookpoint.target.postpackaging

Ant targets +Multi-Build +Multi-Build-Last +Target-Build

PostBuild Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script after it packages and signs the build.

Property	project.hookpoint.target.postbuild
Ant targets	+Multi-Build +Multi-Build-Last +Target-Build

PostMultiBuild Hook Point

When you use this hook point, Mobile Designer invokes your custom Ant script after it creates all the builds you selected in the Multi Build dialog.

Property	project.hookpoint.target.postmultibuild
Ant targets	+Multi-Build +Multi-Build-Last +Target-Build

Creating Patch Files to Apply to the Cross-Compiled Code

A *patch file* is a standard difference (diff) file that you can create. Use patch files when you need to make changes to the code that the Mobile Designer cross compiler creates. For example:

- You might need to correct issues that prevent the code from compiling.
- You might need to correct issues when the code cross compiles correctly, but the resulting code does not execute as expected.

  For example, in your source Java code, you use multiple post increments, such as `i++` in one line of code:

  ```
  int rgb = (data[i++] << 16) + (data[i++] << 8) + (data[i++]);
  ```

  This line of code can cross-compile to C++ fine, but cannot be guaranteed to execute as Java does. In some C++ compilers this type of code might result in "undefined behavior." In this case, even though the code will convert to C++ and then compile successfully, you might find the run-time execution is incorrect. For these types of situations, it is usually a best practice to write a more compliant variant in your Java code. Continuing this example, you might change the original Java code to something like the following:

  ```
  int rgb = (data[i] << 16) + (data[i + 1] << 8) + (data[i + 2]);
  ```

You create the patch file one time, and Mobile Designer automatically applies the fix to all future builds, even if you continue to make changes to your application's original Java code. For instructions about how to create a patch and where to store the patch file, see "Creating a Patch" on page 130.

When Mobile Designer builds your project, it adds a Builds folder to your project folder. Within the Builds folder, there are folders for each device for which your application was built.

project /Builds/*x.y.z* /*device*

In the folder location above:

- *project* is the name of your project.
- *x.y,z* is the version number you specified for the build.
- *device* is the name of a device for which Mobile Designer built your application.

The *device* folder contains a _temp_/_*language* _ folder that contains the version of the cross-compiled code *without* a patch applied. If a patch file exists, Mobile Designer patches the code and stores the resulting patched version of your project in the _temp_/_*language* _edit_ folder. For example, for Android, Mobile Designer generates a _java_edit_ folder along side the _java_ folder.

Creating a Patch

Mobile Designer provides patch and diff tools with sample scripts so that you can create patch files. For an example of how to use the patch and diff tools, see the Function Demo sample project.

To create a patch file

1. Build your project using one of the following Ant targets:
   - +Multi-Build Ant target. For instructions, see "Building a Project for Multiple Target Devices" on page 120.

     Be sure you select the **Retain output build files** in the Multi Build dialog.
   - +Target-Build Ant target. For instructions, see "Building a Project from the Command Line" on page 121.

     Be sure you set the `retain` option to `true`.
2. Locate your project's Build folder, which Mobile Designer creates during the build process.
3. In the Build folder, locate the cross-compiled version of your code.

   Specifically, look for your code in the following folder:

   project /Builds/*x.y.z* /*device* /_temp_/_*language* _edit_

   In the folder location above:

   - *project* is the name of your project.
   - *x.y,z* is the version number you specified for the build.
   - *device* is the name of a device for which Mobile Designer built your application.
   - *language* is the language into which your application code was cross compiled, for example, java.

4. Review the cross-compiled code and make any minor code adjustments needed to correct the issues you are encountering so that your code compiles as expected on the target platform.
5. Run the patch_maker script that Mobile Designer provides in the following project folder:

   project /Builds/*x.y.z* /*device* /_temp_

   When Mobile Designer builds your project, it places versions of the cross-compiled code in both of the following project folders:

   - _*language* _ folder
   - _*language* _edit_ folder

   The version of the cross-compiled code in the _*language* _edit_ folder includes any patch that you might have already applied.

   When you run the patch_maker script, Mobile Designer generates a diff file with the differences between the two folders.
6. Save the diff file that the patch_maker script created in the project's main folder using the names specified in the following table.

Platform	Patch Name
Android	`android_java.diff`
iOS	`ios_cpp.diff`

Note: If you want to give your diff file an alternate name, for example, if two targets for the same platform require different diff files, you can use a custom name for a diff file. You still save the diff file to the project's main folder. However, if you name your diff file differently from the names specified in the table above, you must add the `patch.name` property to the target device file in the project's targets folder. Set the value of the `patch.name` property to the name of the diff file to use for the device.

6 Installing and Testing Mobile Applications

Using Phoney for Debugging Your Mobile Application

About Using Phoney to Debug Mobile Applications

The Mobile Designer utility, *Phoney*, is a phone simulator that is not platform-specific. You can test your application by running it in Phoney.

If you use the Mobile Designer NativeUI library to create the user interface for your mobile application, the user interface is rendered using Phoney skins that Mobile Designer provides. Mobile Designer provides some platform-specific Phoney skins that attempt to mimic the look-and-feel of a platform. For platforms for which Mobile Designer does *not* provide a platform-specific Phoney skin, Mobile Designer provides a general graphical skin for the user interface. For more information about Phoney skins, see *webMethods Mobile Designer Native User Interface Reference*.

You run Phoney from Mobile Development. For instructions, see *webMethods Mobile Development Help*, *Testing a Mobile Project with Phoney*. Alternatively, you can run Phoney from the command line. For more information, see "Running Phoney from the Command Line" on page 138.

You can also still use the ++Run-Phoney, ++Run-Phoney-With-Activation, or ++Run-Phoney-With-Re-Activation Ant targets to run a mobile application in the Phoney simulator. For information about the actions Mobile Designer takes when you run Phoney, see "Phoney Ant Target Summary" on page 135 and "Steps Performed for Phoney Ant Targets" on page 136.

If a mobile application accesses services via the Internet and you want to use SSL to secure the communications between Phoney and the service, install certificates on Phoney. For instructions, see "Installing Certificates on Phoney" on page 143.

You can use the Phoney Metrics panel to get an estimation of an application's memory and thread usage to help determine whether you might encounter issues with memory and thread usage when running the application on physical devices. For more information, see "Using Phoney to Monitor an Application's Memory and Thread Usage" on page 143.

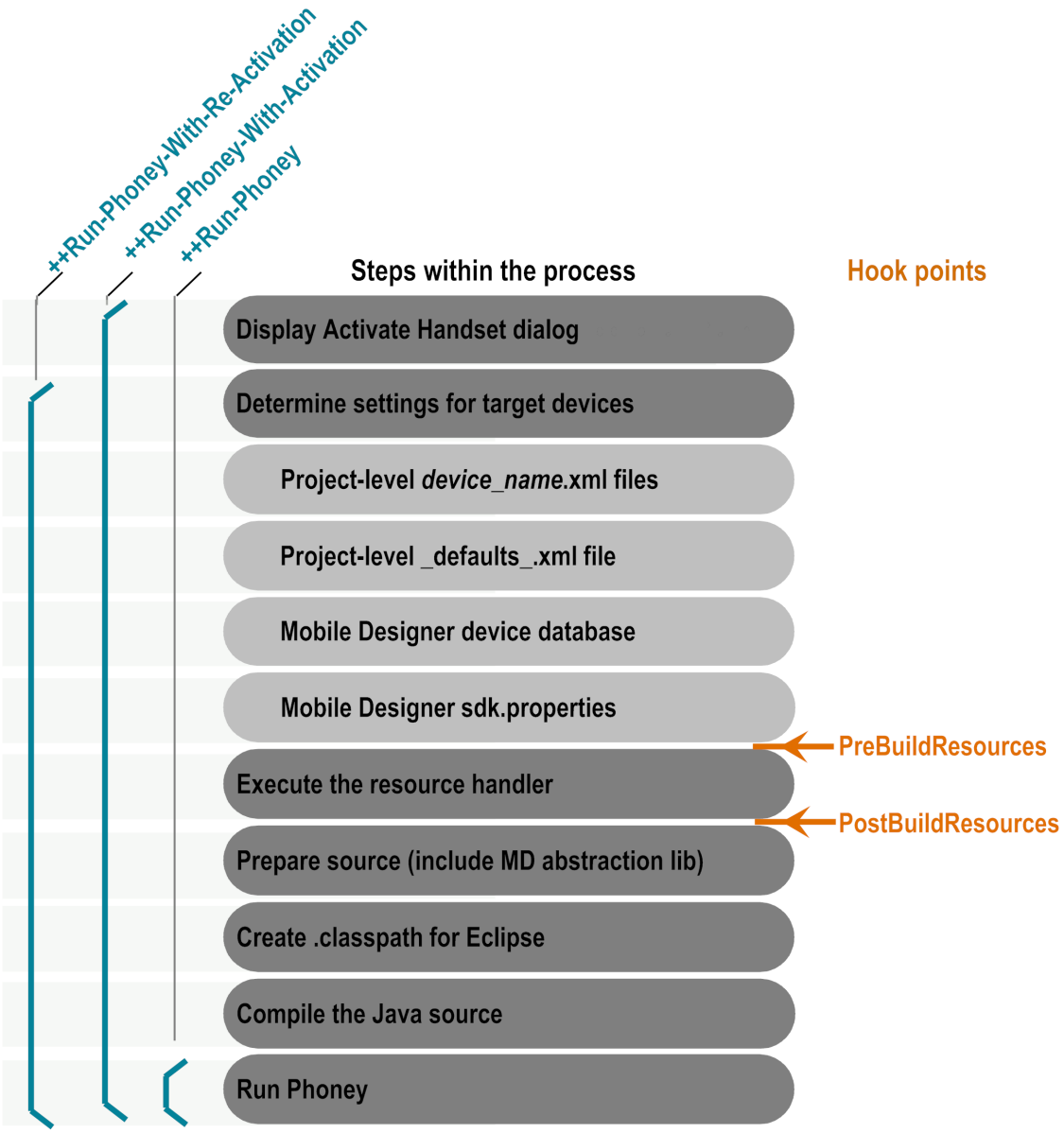
Note: You can also test and debug by installing a mobile application on to a physical device or a platform-specific emulator or simulator. See the providers' documentation for the latest instructions on how to install applications and use emulators or simulators that they provide. This documentation includes limited instructions for some platforms. For more information, see "Installing Applications on Devices" on page 151.

Phoney Ant Target Summary

The following diagram provides summary information about the process Mobile Designer performs when you use Phoney Ant targets. It shows:

- The names of the Ant targets you can use to run Phoney.
- The steps Mobile Designer performs. For details about each step, see "Steps Performed for Phoney Ant Targets" on page 136.
- The hook points in the process where Mobile Designer can run custom Ant scripts that you provide. For more information about hook points, see "Creating Custom Ant Scripts to Run at Predefined Hook Points" on page 125.

Note: For instructions on how to run Phoney, see *webMethods Mobile Development Help, Testing a Mobile Project with Phoney* and "Running Phoney from the Command Line" on page 138.



Steps Performed for Phoney Ant Targets

This section describes the steps that Mobile Designer performs when you run the ++Run-Phoney, ++Run-Phoney-With-Activation, and ++Run-Phoney-With-Re-Activation Ant targets to run a mobile application in the Phoney simulator.

Note: For instructions on how to run Phoney, see *webMethods Mobile Development Help, Testing a Mobile Project with Phoney* and "Running Phoney from the Command Line" on page 138.

++Run-Phoney Ant Target

When you use the ++Run-Phoney Ant target, Mobile Designer performs only the Run Phoney step.

The Run Phoney step starts Phoney for the last activated device. You can specify Phoney startup options using the phoney.base.params property.

++Run-Phoney-With-Activation and ++Run-Phoney-With-Re-Activation Ant Targets

Use the ++Run-Phoney-With-Activation Ant target to activate a device, then run Phoney. Use the ++Run-Phoney-With-Re-Activation Ant target to reactivate the last device, then run Phoney.

When you use the ++Run-Phoney-With-Activation or ++Run-Phoney-With-Re-Activation Ant target, Mobile Designer performs the steps in the following table. The table also indicates hook points where Mobile Designer can run custom Ant scripts that you provide. For more information about hook points, see "Creating Custom Ant Scripts to Run at Predefined Hook Points" on page 125.

The first four steps and hook points are for reactivating the last device. For details about these steps, see "Steps Performed to Activate Handsets" on page 148.

Step	Description
1	**Display the Activate Handset dialog** Mobile Designer displays the Activate Handset dialog. **Note:** Mobile Designer only displays the Activate Handset dialog when you run the ++Run-Phoney-With-Activation Ant target.
2	Determine setting for target devices
hook point	Optional. Run a custom Ant script for the PreBuildResources hook point
3	Execute the resource handler
hook point	Optional. Run a custom Ant script for the PostBuildResources hook point
4	Prepare source (include Mobile Designer abstraction lib)
5	Create .classpath for Eclipse
6	Compiles the Java source

Step	Description
7	Run Phoney Mobile Designer starts Phoney for the last activated device. You can specify Phoney startup options using the phoney.base.params property.

Running Phoney from the Command Line

You can run Phoney from the command line using any of the Phoney Ant targets. To execute the Ant target, from the command line navigate into the project's folder.

Syntax

```
[options] ant_target
```

For *ant_target*, specify one of the Phoney Ant targets, for example, `++Run-Phoney`. The options you can use are listed in the table below.

Phoney Startup Options

Note: If you want to run Phoney from Software AG Designer, you can specify the startup options in the phoney.base.params property.

Option and Description
{--accelerate_images | -ai} Enables image acceleration, speeding up 32-bit images.
{--device | -fd} *device_name* Forces Phoney to emulate the dimensions and keys of the specified device. For *device_name*, specify the name you selected from the **Choose your handset** list in the Add Handset dialog when you added the device to the project.
{--dimensions | -d} *width***x***height* Specifies the canvas dimensions. The default is 176x220 pixels. **Note:** If you also specify the `{--device | -fd}` option, the dimensions you specify with the `{--dimensions | -d}` option override the dimension specified with the `{--device | -fd}` option. Other settings specified with the `{--device | -fd}` option still apply.
{--fc_pim_roots | -rt} *simulated_drives*

Option and Description
Specifies one or more simulated drives that Phoney uses when executing FileConnection classes. To specify a drive use the format: `fake_root_name,path` For example, to simulate the D:\ drive in C:\fake\d, use the following `D:\,C:\fake\d` Use a semicolon-separated list to specify multiple simulated drives. For example, to simulate the D:\ drive in C:\fake\d and the E:\drive in C:\fake\e, use the following `D:\,C:\fake\d;E:\,C:\fake\e` The default is the following: `C:\,%FCDIR%/c/` For the default, `%FCDIR%` maps to the following directory: *Mobile Designer_directory*\Tools\Phoney\FC_PIM_fake_roots
{--fixed_frame_rate | -ff} Specifies that you want Phoney to create AVI files using fixed frame rate. If you do not specify this option, Phoney uses a variable rate.
{--frame_rate | -fr} *rate* Specifies that you want Phoney to record everything that is displayed in the Phoney window in an AVI file. Specify *rate* to indicate the frame rate (frames per second) to use to record the movie. The default frame rate value is 25fps. Using this option allows you to record a movie of your application, for example, to use as a demonstration video.
{--invert | -i} Inverts the number pad so that it behaves like a phone keypad.
{--jad_param | -jp} *parameters* Specifies Java Application Descriptor (JAD) parameters. If a JAD file is present, the parameters you specify override the parameters in the JAD file. You can specify the `{--jad_param | -jp}` startup option multiple times.
{--location_position | -lp} *latitude;longitude;altitude* Specifies the default position returned by the Location API. The default is "51:30:26;-0:7:39;24" (London).

Option and Description
{--max_scale | -ms} Sets the scale of the current screen to the maximum that is allowable given the current screen resolution.
{--metrics_mem_limit_kb | -mm} *megabytes* Specifies the maximum number of kilobytes to use for memory before issuing warnings. Phoney signals a warning if an application's memory usage is approaching or exceeding configured limit. The default is 51200 KB (50 MB). For more information, see "Using Phoney to Monitor an Application's Memory and Thread Usage" on page 143. **Note:** It is recommended that you do not change the default. If you need to change it, do so for specific devices rather than the entire project.
{--metrics_thread_limit | -mt} *number* Specifies the maximum number of threads to use before issuing warnings. Phoney signals a warning if an application's thread usage is approaching or exceeding configured limit. The default is 15 threads. For more information, see "Using Phoney to Monitor an Application's Memory and Thread Usage" on page 143. **Note:** It is recommended that you do not change the default. If you need to change it, do so for specific devices rather than the entire project.
{--no_connectivity | -nc} Start Phoney up with all virtual network connections disabled. This can be used to simulate a device in a no-coverage area, or in Airplane mode.
{--no_fake_fc_pim_roots | -fk} Disables the behavior of the `{--fc_pim_roots | -rt}` startup option. **Caution:** When you use `{--no_fake_fc_pim_roots | -fk}`, Phoney lists your computer's root devices directly. As a result, your mobile application has direct access to the root drives, for example, C:\.
{--no_file_connection | -fc} Disables the FileConnection portion of FC-PIM. Use this option when you want to simulate devices that do not support the FileConnection API.
{--no_menus | -nm} Disables the menu bar.

Option and Description
{--no_pim | -np} Disables the reading of the Phoney Contacts file.
{--no_settings | -ns} Specifies that you want Phoney to only use the command-line options for user settings and ignore settings in user settings files. By default, Phoney retains settings from previous sessions in a properties file so they can be used again. Using this property ensures that Phoney does not use previously saved settings.
{--open_dialog | -od} Shows the file open dialog so that you can have Phoney load a JAD or JAR file at startup.
{--open_screenshot | -os} Indicates that you want Phoney to automatically open any screen shot you take using F1. The screen shot is open using the system-defined application for viewing PNG images.
{--pim_root | -pr} Specifies where to locate the Phoney Contacts file. The default is the following: `%PIMDIR%` For the default, `%PIMDIR%` maps to the following directory: *Mobile Designer_directory*\Tools\Phoney\FC_PIM_Contacts
{--properties | -p} *`filename`* Specifies you want Phoney to use device settings in the specified properties file. **Important:** Be sure to specify this setting so that Phoney automatically represents the activated device.
{--redirect_output | -ro} Redirects all stdout and stderr text to an internal buffer that you can then view from the **Phoney > Console** menu item.
{--repaint_sleep | -rs} *`milliseconds`*

Option and Description
Specifies the number of milliseconds the MIDP repaint calls sleeps, allowing time for the paint thread to service the queue.
{--rms_files | -r} Specifies that you want to store RMS data in files. Phoney uses RMS files for the RecordStore class read/write processes. If you do not use file-based RMS, the save/load operations do not persist between Phoney sessions.
{--scale | -s} *number* Specifies the window scale multiplier. The default is 1. **Note:** While running Phoney, you can change the screen scale by pressing the '+' or '-' key to increase or decrease the scale.
{--single_keypress | -sk} Disables multiple simultaneous key presses. By default, multiple simultaneous key presses are enabled.
{--slow_rms | -sr} *milliseconds* Specifies that you want to emulate a record store that performs slowly. Specify the number of milliseconds you want the RecordStore class to use to perform an add or set operation. **Note:** This is mostly useful for legacy J2ME devices that sometimes had issues writing to the RecordStore, causing a lag when trying to save too frequently or not allowing enough time for the write operations to complete. By simulating the time that the device takes, you can determine how the application will perform on devices that exhibit this issue.
{--softkeys | -k} Forces a system soft-key area for LCDUI soft keys.
{--start_landscape | -sl} Starts Phoney up and immediately rotates the device by 90 degrees (normally landscape mode for most devices).
{--sound | -so} Enables sound playback using the JDK 1.5 Java Sound API.

Option and Description
{--verbose | -v} Specifies that you want verbose output.
{--verbose_mmapi | -vm} Specifies that you want verbose debug information from the MMAPI code.

Installing Certificates on Phoney

Phoney makes use of the J2SE keychain. As a result, you can use X.509 v1, v2, and v3 certificates and PKCS #7-formatted certificate chains consisting of certificates of that type.

Note: The J2SE keychain details vary based on the VM you have installed. For more information about the Oracle Java 7 keychain, see "http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html".

To install custom SSL certificates on Phoney

- Use keytool to install the certificate.

  The keytool is located in %JAVA_HOME%/bin.

  Example:

```
keytool -importcert -file %KEY_FILE_FOLDER%\%KEY_FILE%
-alias %CERTIFICATE_ALIAS% -keystore "%JAVA_HOME%\jre\lib\security\cacerts
```

Using Phoney to Monitor an Application's Memory and Thread Usage

You can use Phoney to run applications on a simulated, generic device. When using Phoney to run an application, you can view memory and thread usage information for the simulated device in the Metrics panel. Use the metrics to determine:

- Estimation of how much memory and threads an application uses.
- When an application's memory and thread use is at its highest.

The metrics can help determine whether you might encounter issues with memory and thread usage when running the application on physical devices.

Additionally, Phoney displays warnings if an application's memory and/or thread use exceeds configured thresholds.

Note: When running the application on a physical device, the metrics might differ somewhat from the values in the Metrics panel. For example, when running the application on a physical device, the memory usage might be a little less because the format of the application resources, such as, data, images, and sound, might be in a format better suited for the physical device. Native Mobile Designer libraries used on physical devices might also differ slightly in their usage of threads.

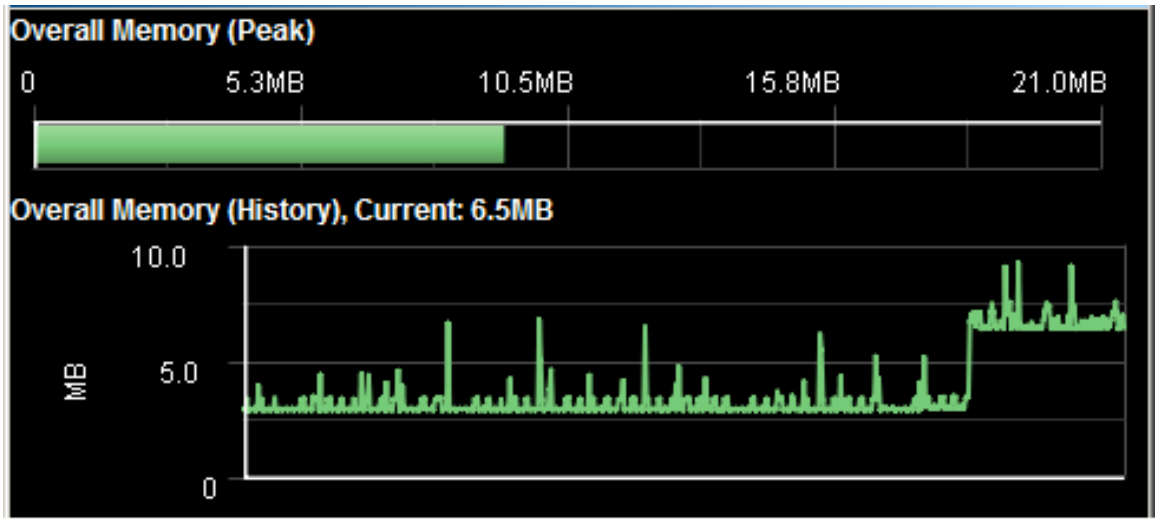
Also note that other applications accessing sound and video hardware can cause the JVM in which Phoney is running to create spurious threads. For example, web browsers displaying video, Flash, or other rich content might cause the JVM in which Phoney is running to create spurious threads. You can safely ignore these additional threads.

Metrics Panel

You can open the Metrics panel from Phoney by selecting **Phoney > App Metrics**, or by pressing CTRL+M on the keyboard. The Metrics panel displays the following information:

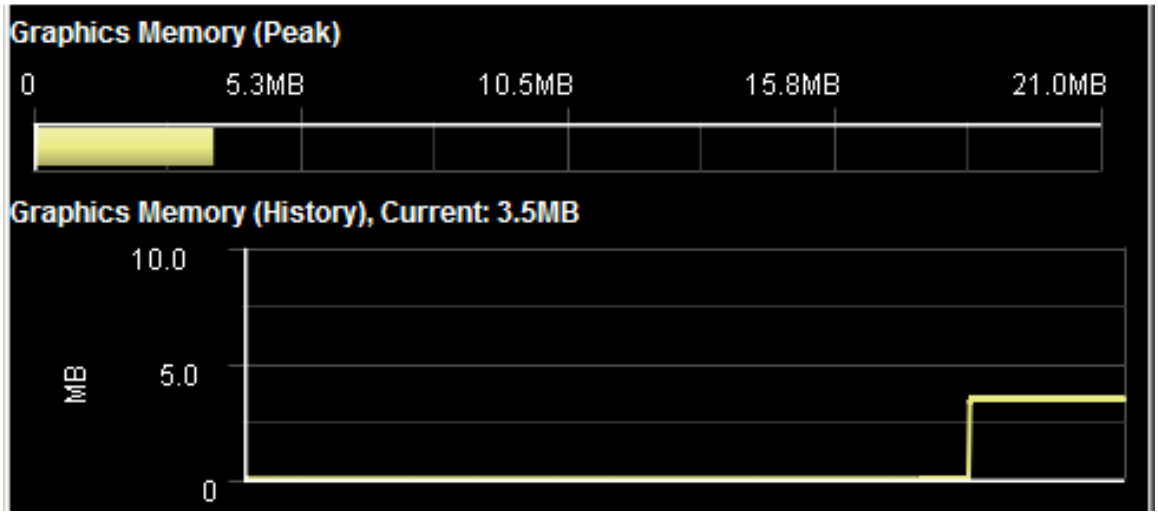
- **Overall memory usage of an application**

  The Peak bar shows the highest recorded use of memory, as a proportion of the current JVM's total size. The History graph shows the last 60 seconds of memory usage.



- **Memory usage for Image classes within an application**

  Similar to the overall memory metrics, these metrics display a Peak bar that shows the highest recorded use of memory for Image classes within the application. The History graph shows the last 60 seconds of memory usage.



Because both the overall memory usage and the usage for Image classes use the same scale, you can determine the proportion of the overall memory that the application uses for Image classes.

- **Thread usage**

  For thread usage, the Metrics panel lists the number of threads that the application has created. The History graph shows the last 60 seconds of thread usage.

Note: Even if the Metrics panel is closed, Phoney continues to monitor memory and thread usage and displays warnings if memory and/or threshold use exceeds configured thresholds.

Setting Warning Thresholds and How Warnings are Displayed

You can configure limits for when you want Phoney to signal a warning if an application's memory and/or thread usage is approaching or exceeding configured limits. To configure the limits, you specify options when you start Phoney. If you do not specify the startup options, Phoney uses default values. You can specify the startup options:

- On the command line when you start Phoney.

- When starting Phoney via Software AG Designer. Set the options in one of the following locations:
  - On the **Arguments** tab inside the Run Configurations window.
  - Using the phoney.base.params property in the project's targets/_defaults_.xml file.

When an application goes over the configured limits, Phoney displays a blinking icon on the bottom-left of the screen. If the Metrics Panel is open, the background behind the graphs also blinks red. The warnings blink for approximately 60 seconds after the application's memory and/or thread usage reduces to the configured limits.

Note: Even if the Metrics panel is closed, Phoney monitors memory and thread usage and displays warning icons.

The following table lists the startup option you use to configure the limits, the default values for the limits, and provides information about when Phoney displays warning.

Memory limit	**Startup option**	`{--metrics_mem_limit_kb | -mm}`
	Default value	50 MB
	Warning	Blinking **RAM** warning icon - **When blinking slowly**, the application's usage is over 90% of the memory limit. - **When blinking rapidly**, the application's usage is over the memory limit.
Thread limit	**Startup option**	`{--metrics_thread_limit | -mt}`
	Default value	15 threads
	Warning	Blinking **CPU** warning icon - **When blinking slowly**, the application's usage is over 80% of the thread limit. - **When blinking rapidly**, the application's usage is over the thread limit.

The following example shows startup options to set the memory usage limit to 20 MB and the thread usage limit to 5 threads.

```
--metrics_thread_limit 5 --metrics_mem_limit_kb 20480
```

Alternatively, you can use the short form of the options:

```
--mt 5 --mm 20480
```

Activating Devices

About Activating Devices

Mobile Development will add devices that you want your application to support to your project when you define launch configurations. For more information, see *webMethods Mobile Development Help*, *Building a Mobile Project*.

However, you can continue to add devices using Ant targets as described in "Adding Devices to a Mobile Application Project" on page 91. When testing how your application runs on one of the devices, you need the run-time settings to be set for a specific device. To get the correct settings for a device, you can activate it. For example, if you want to test your application using Phoney, you might first activate the device for which you want to test, then run Phoney to test your application.

When you activate a device, Mobile Designer

- Hot-swaps in the required Mobile Designer run-time classes. For more information, see " Mobile Designer-Provided Run-Time Classes" on page 49 and "Run-Time Classes Properties" on page 201.
- Sets up the project compile path appropriately by creating the .classpath file that provides information about how to compile the Java code.
- Runs the project's resource handler so that the class contains settings for the device you activate.
- Compiles the code.

For more information about the steps Mobile Designer takes when you activate a device, see "Activate Devices Ant Summary" on page 147 and "Steps Performed to Activate Handsets" on page 148.

To activate a device, you use the ++Activate-Handset or ++Re-Activate-Handset Ant targets. For instructions, see "Activating a Device" on page 150.

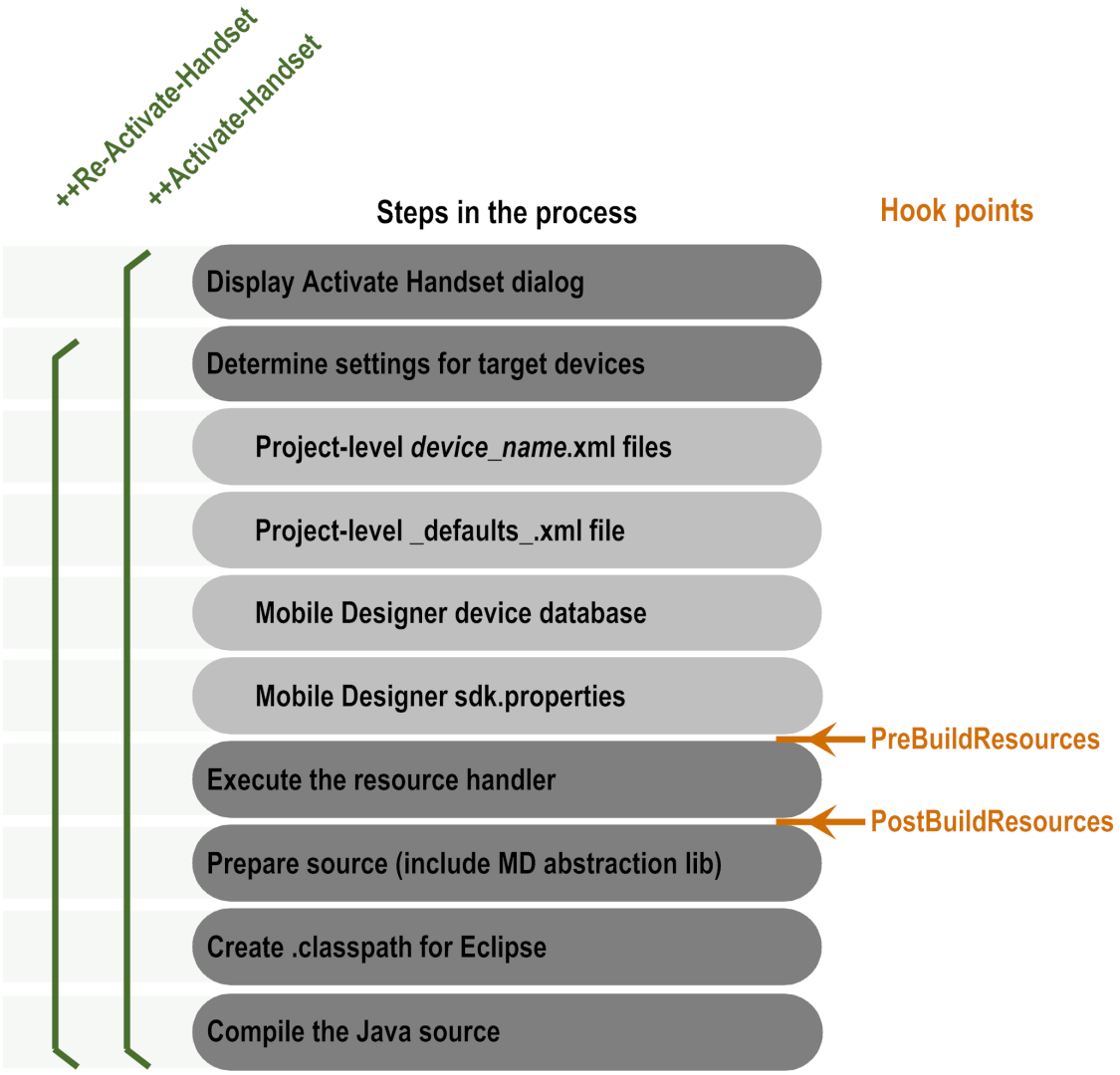
Activate Devices Ant Summary

The following diagram provides summary information about the process Mobile Designer performs when activating a device. It shows:

- The names of the Ant targets you can use to activate devices.
- The steps Mobile Designer performs. For details about each step, see "Steps Performed to Activate Handsets" on page 148.

- The hook points in the process where Mobile Designer can run custom Ant scripts that you provide. For more information about hook points, see "Creating Custom Ant Scripts to Run at Predefined Hook Points" on page 125.

Note: For instructions on how to activate a device, see "Activating a Device" on page 150.



Steps Performed to Activate Handsets

This section describes the steps that Mobile Designer performs when you run the ++Activate-Handset or ++Re-Activate-Handset Ant targets to activate a device.

The table also indicates hook points where Mobile Designer can run custom Ant scripts that you provide. For more information about hook points, see "Creating Custom Ant Scripts to Run at Predefined Hook Points" on page 125.

Note: For instructions on how to activate a device, see "Activating a Device" on page 150.

1 **Display the Activate Handset dialog**

Mobile Designer displays the Activate Handset dialog.

Note: Mobile Designer only displays the Activate Handset dialog when you run the ++Activate-Handset Ant target.

2 **Determine settings for target devices**

Mobile Designer determines the settings for the device you are activating. It retrieves the settings from the following sources in the order listed.

- Project-level *device_name* .xml files
- Project-level _defaults_.xml file
- Mobile Designer device database
- Mobile Designer sdk.properties file

Mobile Designer uses the first setting it encounters. For example, if Mobile Designer encounters a setting in the project-level target *device_name* .xml file and then again in the project-level _defaults_.xml file, Mobile Designer uses the setting from the target *device_name* .xml file.

For more information about:

- Project-level device files, see "Where You Set Properties" on page 84 and "Setting Project Properties" on page 86
- Mobile Designer device database, see "Devices that a Mobile Application Supports" on page 91
- Mobile Designer sdk.properties file, see "Mobile Designer Configuration Properties (sdk.properties)" on page 18

hook point **PreBuildResources**

If you have created an Ant script to run at the PreBuildResources hook point, Mobile Designer runs the Ant script.

3 **Execute the resource handler**

Mobile Designer runs the resource handler that you created for the project. When running the resource handler, Mobile Designer records all the resources required for your application. Mobile Designer also creates

the class. For more information about creating the resource handler, see "Defining Resources for a Mobile Application Project" on page 73. For more information about the Parameters class, see "Application and Parameter Classes" on page 50.

At this point in the build process, Mobile Designer uses the following project properties:

- project.reshandler.src.path for the location of the project's resource handler script and any associated classes
- project.java.reshandler.name for the name of the resource handler class you created for your project.
- project.resource.dir.root for the location of the top-level folder that contains the resources (image files, etc.) for your project
- project.reshandler.additional.libs.path for the location of additional libraries that your project's resource handler requires

For more information about these properties, see "Resource Handler Properties" on page 197.

hook point **PostBuildResources**

If you have created an Ant script to run at the PostBuildResources hook point, Mobile Designer runs the Ant script.

4 **Prepare source (include Mobile Designer abstraction lib)**

Mobile Designer prepares the source by including the relevant run-time classes. For more information, see " Mobile Designer-Provided Run-Time Classes" on page 49 and "Run-Time Classes Properties" on page 201.

5 **Create .classpath for Eclipse**

Mobile Designer creates a classpath file in the project's root folder to reference the stubs that the device's run-time code and that the resource handler requires. The Mobile Designer project is refreshed so that the classpath changes are propagated through to the codebase.

6 **Compile the Java source**

Activating a Device

To activate a device, you use one of the following Ant targets:

- ++Activate-Handset to specify the device and the language combination that you want to activate.

- ++Re-Activate-Handset to activate the last device that was activated.

Note: When you activate a device, Mobile Designer also creates a debug or run configuration. You can find the created .launch file in the project's _temp_ folder.

To activate a device

1. In Software AG Designer, click the Project Explorer view, expand the Mobile Designer project, and drag the build.xml file to the Ant view.
2. In the Ant view, double-click the Ant target you want to use:
   - Double-click **++Activate-Handset** if you want to specify a device/language combination.

     Mobile Designer displays an Activate Handset dialog. Continue with the next step to complete the procedure.
   - Double-click **++Re-Activate-Handset** to re-activate the last device.

     Mobile Designer does not display a dialog. It re-activates the last device and does not require you to perform further actions.
3. In Activate Handset dialog, select the device you want to activate from **Choose your handset** list.
4. Select the language group for which you want to test from **Choose your language group** list.
5. Click **Activate Handset**.

Installing Applications on Devices

About Installing Applications on Devices

After you use webMethods Mobile Designer to build an application, you can install the final binary on the target devices. This documentation describes how to install applications on the following commonly-used platforms:

- Android
- iOS

The procedures in this documentation do *not* cover all possible setups and scenarios. Refer to the device provider's web pages for further details.

Installing Applications on Android Devices

To install Android applications, you can use the following methods to install applications on emulated or physical devices:

- Install an Android application package (APK) file using the Android Debug Bridge (ADB)
- Install an Android application package (APK) file using the built-in emulator of Mobile Development

Note: If you need to install a certificate, this documentation provides information about installing certificates on Android 4.0 and later devices. See "Installing Certificates on Android 4.0 and Later Physical Devices" on page 156. For additional information, see the device provider's web pages.

Installing an APK File to an Emulated or Physical Device Using the Android Debug Bridge

To upload an APK file to an Android device, use the ADB command-line tool. The ADB tool is located in the platform-tools directory of your Android SDK install. See the Android developer website for more information about the ADB tool.

Tip: To make accessing the Android tools from the command line easier, add the platform-tools and tools folder of your SDK install to your default path.

Note: You can use the following procedure for both virtual Android devices and physical Android devices.

To install an APK file using the ADB tool

1. Ensure that your Android device is fully booted.
2. If you are using a physical device, connect the device to your computer and ensure that you have all required drivers installed.

   Note: For physical Android devices, you might need to enable debugging. To do so, launch the Settings application, and select **Applications > Development > USB Debugging**.

3. Open a command prompt and execute one of the following, where *apkFilePath* is the path to the APK file and *filename* is the name of your APK file.

   a. **For Windows:**

   ```
   cd apkFilePath
   C:\android-sdk-windows\platform-tools\adb install filename.apk
   ```

b. **For Macintosh**:

```
cd apkFilePath
/android-sdk-macosx/platform-tools/adb install filename.apk
```

The ADB tool interacts with the device and installs your build.

4. Launch your application from the device and test it.

Note: From the command line, use `adb logcat` to monitor output from your device. This can be useful when debugging your application.

Installing an APK File to an Emulated or Physical Device Using the Built-in Emulator of Mobile Development

To install an application to an emulated or physical device in Mobile Development, follow the instructions as described in *webMethods Mobile Development Help*, *Installing an Application on Android Using the Emulator*.

Installing Applications on iOS Devices

To install iOS applications, you can use the following methods:

- **To install to a simulator**, you can install using Apple Xcode.
- **To install to a physical device**, you can:
  - Install using the Apple Xcode IDE
  - Install an Ad-Hoc Build Using iTunes

Note: If you need to install a certificate, this documentation provides information about installing certificates on iOS devices. See "Installing Certificates on iOS Physical Devices" on page 156. For additional information, see the device provider's web pages.

Installing to a Simulated or Physical Device Using the Apple Xcode IDE

This method is useful when installing during the development process and when you want to install to a simulated device.

To install to a simulated or physical device using Xcode

1. When building your project using the Mobile Designer +Multi-Build Ant task, be sure to select the **Retain output build files** check box.

When the build completes successfully, Mobile Designer retains a folder named _temp in the same folder as your build. The temp/_cpp_edit_ folder is an Xcode project (the .xcodeproj file) that you can use to access your cross-compiled source.

2. Double-click the .xcodeproj file.

   Xcode starts.

3. Select the device to which you want to install the application:

   - **To install to a simulated device,** on the screen set the scheme to the simulated device to which you want to install.
   - **To install to a physical device**, attach the iOS device to your computer.

4. Click **Run**.

   Xcode compiles your application, signs it with the development certificate, and installs it on the selected simulated or physical device.

Installing an Ad-Hoc Build to a Physical Device Using iTunes

Use the following procedure to install an ad-hoc build to a physical iOS device.

To install an Ad-Hoc build using iTunes

1. Gather the build files you need for installation and make them available on the machine running iTunes.

   a. Locate the application bundle in the project Builds folder.

      After running the Mobile Designer+Multi-Build Ant task to create an ad-hoc device build, Mobile Designer places the application bundle in this location.

   b. Compress the application bundle on the Macintosh where you created the build.

      If you want to distribute the build, for example, via an email message or upload it to a server, you must compress the files before moving them. Distributing an uncompressed application bundle can cause issues, for example, missing symbolic links and/or permissions set for the build.

      Important: It is recommended that you use a program such as 7Zip, WinZip, or WinRar for the decompression. Using the built-in **Extract All** function that comes with Windows can damage the build when it is extracted.

   c. Locate the ad-hoc mobile provision file used when signing the application.

      Tip: This is the file specified in the `ios.provisioning.profile.adhoc` property in the _defaults_.xml of your project.

   d. Copy the compressed application bundle and the ad-hoc mobile provision file (.mobileprovision file) to the computer you want to use for the installation.

Place the files in an easy-to-find location, for example, the desktop.

2. Optionally, if the application already exists on the physical device, delete the application if you want to perform a clean installation.
3. Connect the physical iOS device to your computer.
4. Start iTunes.
5. Optionally, install the ad-hoc mobile provision file by dragging the .mobileprovision file into **Library > Applications** in iTunes.

   Important: You must install the .mobileprovision file if:

   - This is the first time installing this application on the device.
   - The .mobileprovision file has changed since the last time you installed the application.

6. Install the application bundle.
   a. Extract the files from the compressed application bundle.
      - On Windows, the application is in a folder with the name *application_name* .app
      - On Macintosh, the application is in a single file with a name that matches the application name.
   b. Drag the application bundle (that is, the Windows folder or Macintosh file) into **Library > Applications** in iTunes.
7. Select **Library > Applications** to verify that the new application is present.
8. Under **Devices**, select the iOS device to which you want to install the application.
9. Select the **Applications** tab.
10. Select **Sync Applications** and that the application you want to install is selected.
11. Select the **Summary** tab.

    iTunes installs the application.

Installing Custom SSL Certificates on Devices

If an application accesses services via the Internet and you want to use SSL to secure the communications between the device and the service, install certificates on the device. This documentation describes the supported certificate types and how to install certificates for the following platforms:

- Android 4.0 and later devices
- iOS physical devices

The procedures in this documentation do *not* cover all possible setups and scenario. Refer to the device provider's web pages for further details.

Installing Certificates on Android 4.0 and Later Physical Devices

Android 4.0 (Ice Cream Sandwich) and later supports DER-encoded X.509 certificates saved in files with a .crt or .cer file extension.

Note: If you do not have a valid certificate installed, you will see the error **javax.net.ssl.SSLHandshakeException: java.security.cert.CertPathValidatorException: Trust anchor for certification path not found.**

To install custom SSL certificates on Android 4.0 and later physical devices

1. If your certificate file has a .der or other extension, change it to .crt or .cer.
2. Use the Android Debug Bridge (ADB) tool to send a copy of your certificate to the SD card on the device. To do so, with the ADB running, execute the following command, where *certificate* is the name of your certificate:

   ```
   adb push certificate.crt /sdcard/
   ```

3. On the Android device, select **Settings > Location & Security Settings > Set up screen lock** to ensure you have a password or screen lock for the device.
4. On the device, select **Settings > Personal > Security > Credential Storage > Install from SD Card** to install the custom certificate.

Installing Certificates on iOS Physical Devices

Apple iOS supports DER or Base64-encoded X.509 certificates saved in files with a .crt, .cer, .der or .pem file extension.

Note: If you do not have a valid certificate installed, you might see an error on the console **MD: ERROR: Can't reach url "..." and a java.io.IOException**.

Important: You can only use the following procedure to install custom certificates on a physical iOS device. The procedure will not work to install certificates to the keychain that the iOS simulator uses.

To install custom SSL certificates on iOS physical devices

1. Send the certificate to your device by doing one of the following:
   - Send an email message with the certificate as an attachment. On the device, open the email attachment.
   - Put the certificate on an accessible server. On the device, download the certificate from Safari.

   When the device receives the certificate, it opens an Install Profile dialog.

2. In the Install Profile dialog, click **Install**.
3. Double-check the root certificate to ensure it is the one you want to install.
4. Click **Install** in the warning dialog.
5. Enter your passcode if you have one set up on your device.

On an iOS device, you can view and remove installed certificates selecting **Settings > General > Profiles**.

A Project Properties Reference

Note: You are not required to define values for all the properties described in this reference because Mobile Development defines values for most of the properties. However, if you need to set a value or change a predefined value for your project, you can. For instructions, see "Setting Project Properties" on page 86.

Android Project Properties

The Android properties customize how Mobile Designer build applications for Android devices.

project.android.sdk.version.override

Overrides the default Android SDK that Mobile Designer uses for the project.

Platforms Android

Value API number of the SDK you want to use.

Default `21`

For more information about the sdk.properties, see "Configuring Mobile Designer for the Android SDK" on page 30.

Build Results Properties

The build results properties customize how Mobile Designer creates the final binary for the project.

packager

Specifies the packager that you want Mobile Designer to use to create the JAR for each J2ME device that your project supports.

Note: Mobile Designer does not use this property for platforms that do not use JAR files, for example, iOS and Android.

If you are building your application for a platform that does use JAR files, be sure to set the property to a packager that creates JARs that the devices in your project can run. Some devices might not be able to execute JARs created by some packagers.

Platforms All

Value Specify one of the following values:

Value	Meaning
`jar`	Mobile Designer uses the built-in JAR packager that is part of the JDK.
`zip`	Mobile Designer provides compression with WinZip.
`7zip`	Mobile Designer provides compression with 7Zip. To create the JAR Mobile Designer uses the following command-line parameters when executing 7Zip: `${packager.7zip.args1} "filename" * ${packager.7zip.args2}` The `packager.7zip.args1` and `packager.7zip.args2` properties are defined in the bs-defaults.xml file, which is in the following location: *Mobile Designer_directory*/Tools/Build/BuildScript/v1.0.0/bs-defaults.xml The following shows the default property settings in the bs-defaults.xml file: `<property name="packager.7zip.args1" value="a -r -tzip"/>` `<property name="packager.7zip.args2" value="-mx=9 -mfb=255"/>`
`kzip`	Mobile Designer provides compression with Kzip. To create the JAR Mobile Designer uses the following command-line parameters when executing Kzip: `${packager.kzip.args} "filename" *` The `packager.kzip.args` property is defined in the bs-defaults.xml file. The following shows the default property setting in the bs-defaults.xml file: `<property name="packager.kzip.args" value="-r -zl121"/>`

Default `kzip`

If you do not have the KZip packager installed, at run time Mobile Designer uses 7Zip.

project.handset.skip.compilation

Specifies whether you want Mobile Designer to skip the build step. You can have Mobile Designer skip the build step if you want to use native platform tools to compile and create the final binary.

Platforms All

Value

- `true` to skip the build.

  Mobile Designer performs all the resource handling and cross-compiling (for platforms that require it), but does *not* compile the final binary.

  Important! When you set this property to `true`, be sure to select **Retain output build files** in the Multi Build dialog. When you select **Retain output build files**, Mobile Designer retains the cross-compiled code it generated from your original mobile code, along with any project (for example, Xcode project for iOS) it might have generated. You can then use the retained files and the native platform tools to compile to create the final binary.

- `false` to have Mobile Designer perform the build step to compile and create the final binary.

  Mobile Designer performs all the resource handling, cross-compiling, *and* compiles the final binary.

Default `false`

project.jarname.format

Specifies the file name format that you want Mobile Designer to use when creating the final binary for a build of the project.

Platforms All

Value File name format that you define by specifying one or more of the parameters listed below. At run time, Mobile Designer replaces the parameters with the values listed below.

- **@PROJECT@**

  is replaced with: `${project.jar.name}`

- **@LANGPROJECT@**

is replaced with: `${project.jar.name.selected-langgroup}`, if it exists, otherwise replace with the value of `${project.jar.name}`

- **@HANDSET@**

  is replaced with: `${mobiledesigner.handset.devicegroup.output.filename}`

- **@LANGGROUP@**

  is replaced with: Selected language group

- **@VMAJOR@**

  is replaced with: X part of the project version number (X.y.z) that you specify in the Multi Build dialog when building the project.

- **@VMINOR@**

  is replaced with: Y part of the project version number (x.Y.z) that you specify in the Multi Build dialog when building the project.

- **@VMICRO@**

  is replaced with: Z part of the project version number (x.y.Z) that you specify in the Multi Build dialog when building the project.

Example:

```
<property name="project.jarname.format"
          value="@PROJECT@@HANDSET@@LANGGROUP@"/>
```

Note: The resulting file name must contain only valid characters are alphanumeric (A-Z, a-z, 0-9), period (.), and hyphen (-).

Default None.

Note: You must specify this property for a project.

project.jarname.format.override.*device_name*

Overrides the JAR file name format value specified in the `project.jarname.format` property for a specific device.

When specifying the property, replace *`device_name`* with the name of the device for which you want to override the JAR file name. You can find the device name in the **Handset** field of the Multi Build dialog. For example, to specify the property for the Apple iPhone 5 phone, use the following property, where the `IOS_Apple_iPhone5` portion of the property name is the device name for the Apple iPhone 5 phone:

```
project.jarname.format.override.IOS_Apple_iPhone5
```

Platforms All

Value See the value for the project.jarname.format property

Default No default.

Mobile Designer uses the value of `project.jarname.format` for all devices that you do not specifically override using the `project.jarname.format.override.`*`device_name`* property.

project.multibuild.built.handset.list

Contains a semicolon-separated list of all the devices that Mobile Designer built when building the project. Mobile Designer sets this property.

Platforms All

Value Semicolon-separated list of all the builds that Mobile Designer created

Default n/a

project.output._temp_.folder

Overrides the default name for the project folder that contains the output data from a build of the project.

Platforms All

Value Folder name that you define by specifying the one or more of the parameters listed below. At run time, Mobile Designer replaces the parameters with the values listed below.

Parameter	Replace with the value of
`@LANGGROUP@`	`${selected.langgroup}`
`@PLATFORM@`	`${selected.platform}`
`@TARGET@`	`${selected.target}`

Example:

If you want Mobile Designer to store the output data in a project folder named "_temp_EFIGS_j2me-jar_release_" for an English, French, Italian, German, Spanish (EFIGS) release J2ME build, specify the following:

```
<property name="project.output._temp_.folder"
```

```
          value="_temp_@LANGGROUP@_@PLATFORM@_@TARGET@_"/>
```

Default project _temp_ folder

If you do not set this property, Mobile Designer stores the output and compilation directories in a project _temp_ folder alongside the compiled binary.

Build Script Properties

The build script properties are properties you are required to put in your project's build.xml file.

additional.device.profiles.dir.root

Specifies the location of a folder that contains custom device profiles for your project.

You might want to create your own custom device profiles, but you do not want to include them in Mobile Designer device database. In this situation, set up a separate folder to contain the your additional, custom device profiles, which must use the same XML format that Mobile Designer uses for the device profiles it provides.

To use the custom device profiles for a project, include this property in the project's build.xml file. You must place it in the build.xml file before the following line that imports the targets.xml:

```
<importxmldirectory location="${basedir}/targets"/>
```

Platforms All

Value Directory that contains custom device profiles

For example:

```
<property name="additional.device.profiles.dir.root"
          value="${basedir}/device_info"/>
```

Default None.

Note: Mobile Designer does not use custom profiles if you do not include this property.

mobiledesigner.buildscript.version

Specifies the version of the Mobile Designer build scripting system to use when building your application. Define this property in the project's build.xml file.

Platforms All

Value `v1.0.0`

> **Note:** Currently the only supported value is `v1.0.0`.This property is for backward compatibility in the event Mobile Designer uses a different build scripting system in the future.

Default None.

> **Note:** You must specify this property for a project.

mobiledesigner.runtime.version

Specifies the version of the Mobile Designer run-time code to use for your application. Define this property in the project's build.xml file.

Mobile Designer

Platforms All

Value `v1.0.0`

> **Note:** Currently the only supported value is `v1.0.0`.This property is for backward compatibility in the event Mobile Designer uses a different run-time code in the future.

Default None.

> **Note:** You must specify this property for a project.

project.jar.name

Specifies a text name you want your application to have when installed on a device. Define this property in the project's build.xml file.

Platforms All

Value Name for your project. Valid characters are alphanumeric (A-Z, a-z, 0-9), period (.), and hyphen (-).

For example:

```
<property name="project.jar.name" value="HelloWorld"/>
```

Default None.

> **Note:** You must specify this property for a project.

project.java.midlet.name

Specifies the name of the root MIDlet/Application class of your project's run-time code. . Define this property in the project's build.xml file.

Platforms All

Value For example:

```
<property name="project.java.midlet.name"
          value="com.softwareag.mobile.helloworld.MyApplication"/>
```

Default None.

Note: You must specify this property for a project.

Code Conversion Properties

The code conversion properties configure how Mobile Designer creates the generated C++ or Java code for the project.

java.parser.retain.comments

Specifies whether to retain comments when compiling to Java or C++.

Platforms All

Values

- `true` retains comments when compiling to Java or C++.

  Mobile Designer attempts to keep comments connected to the line where the comments are relevant, typically the line of code that follows the comment. However, due to the fundamental difference between Java, C# and C++, some comments might get misplaced or lost.

  Note: If a project has been built without comments and the property is set to `true`, the cached parsed-Java files do not contain comments to use. To resolve this issue, clear out the project's _temp__bcdfs_cache_ folder.

- `false` removes comments when compiling to Java or C++.

Default `true`

cpp.no.selfprotect

Indicates whether to add a safeguard to every method call.

Platforms All

Values
- `true` indicates that you do *not* want to use the safeguard.

  When set to `false`, the cross compiler might inject the macro line, but it does nothing.
- `false` adds the safeguard to every method call using the injected macro line.

  Caution: Adding the injected macro line to every method call adds overhead to all methods and increases the output size.

Default `false`

cross.compiler.extractinners

Indicates whether to extract inner classes included in the Java source code.

Platforms All

Values
- `true` extracts the inner classes.

  Note: The examination required to perform this action increases the processor time.
- `false` does not extract the inner classes.

Default `true`

cross.compiler.nodatestamp

Specifies whether to exclude timestamps at the top of every generated C++ (CPP) and H file.

Platforms All

Values
- `true` indicates that timestamps are *not* included.

  If you are storing the files in a source control system, omitting this header information reduces the file differences that occur with each cross compilation.

- `false` indicates that timestamps are included.

Default `false`

cross.compiler.render.selfprotect

Indicates whether to add an extra line at the top of every non-static method to prevent self-destruction of the object during the method call.

Platforms All

Values
- `true` indicates that the extra line is added.

  This is part of a fix needed to prevent object self-deletion on the rare occasion that an object nulls all references to itself. The injected line is a macro whose actual contents you can remove by the value of the `cpp.no.selfprotect` property.
- `false` indicates that the extra line is *not* added.

Default `true`

Cross-Compiler Properties

The webMethods Mobile Designer cross-compiler libraries contain code to support the Mobile Information Device Profile (MIDP) and Connected Limited Device Configuration (CLDC) standards and Java functionality. However, your application might not require the entire library, or you might want to override default values. In this case, you can use the cross-compiler properties to customize your use of the cross-compiler library, for example, to disable areas of the libraries or to override default functionality.

The following lists the types of cross-compiler project properties you can set.

Debugging

project.handset.log.debug.filename

Provides the file name to use for the debug log file in which to log debug information when the application uses the debug function call.

> **Note:** To have debug statements written to an output file, you must set the project.handset.log.debug.to.file property to `true`.

Platforms iOS

Value File name for the debug log file

Default `LOG_debug.txt`

project.handset.log.debug.to.file

Specifies whether you want the application to write debug statements to an output file when debugging the application through the Mobile Designer Multi Build dialog box.

Platforms iOS

Value
- `true` to write debug statements to an output file.

  By default, the output file name is LOG_debug.txt. You can specify an alternate name using the "project.handset.log.debug.filename" on page 170 property.
- `false` if you do not want the application to write debug statements to an output file.

Default `false`

Extra Libraries and Custom Code

project.handset.hook.startup

Specifies whether you want Mobile Designer to invoke the startup hook, which is a native function, when the application is starting before the J2ME initialization occurs.

When creating the code to invoke, implement the following function in your native code or patch in the created StubInfo.cpp file:

```
void hookStartUp(void);
```

Caution: Mobile Designer invokes the native code before setting up the J2ME environment, for example, before initializing static data or generating system output streams. Do *not* include code related to the J2ME library or to the mobile application in the native code.

Platforms All

Value
- `true` to invoke native code before the J2ME application code is started.

  Mobile Designer invokes hookStartUp() at the beginning of the application in these locations:
  - In iOS at the start of applicationDidFinishLaunching
  - On all other platforms at the start of main or its equivalent
- `false` if you do not want Mobile Designer to invoke native code before the J2ME application code is started.

Default `false`

project.handset.push.notifications

Specifies whether the application receives push notifications.

Note: Use of this property requires the Wireless Messaging API (WMA) library for J2ME and the project.handset.uses.WMA property set to `true`.

Platforms Android
iOS

Value
- `true` if the application receives push notifications.
- `false` if the application does not receive push notifications.

Default `false`

project.handset.uses.camera

Specifies whether the application uses Mobile Media API (MMAPI) to access the camera functionality.

Platforms All

Value
- `true` if the application uses MMAPI to access the camera.

- `false` if the application does not use MMAPI to access the camera.

Default `false`

project.handset.uses.Database

Specifies whether your mobile application uses the com.softwareag.mobile.runtime.database classes, which Mobile Designer provides. For more information, see "Run-Time Database Classes" on page 51.

Platforms All

Value
- `true` if your application uses the database classes.
- `false` if your application does *not* use the database classes.

Default `false`

Note: If you specify a value for the mobilesupportclient.runtime.dir property, the value of this property is automatically set to `true` because the Mobile Support Client library requires the com.softwareag.mobile.runtime.database classes.

project.handset.uses.FCPIM

Specifies whether your application uses the Personal Information Management (PIM) library for J2ME.

Platforms Android
iOS

Value
- `true` if your application uses the PIM library.

  When you set this property to `true`, necessary permissions or requests are added to native builds.
- `false` if your application does *not* use the PIM library.

  Note: If your applications uses the PIM library, but you set this property to `false` (or it defaults to `false`), your application will fail to compile.

Default `false`

project.handset.uses.Location

Specifies whether an application requires the Location API library for J2ME to compile and that you want Mobile Designer to initialize static values for the library.

Platforms All

Value
- `true` if an application uses the Location library.
- `false` if an application does not use the Location library. Specifying `false` reduces the final size of your project's binaries.

> **Important:** If an application uses the Location library, and you set the property to `false`, Mobile Designer cannot compile the application.

Default `true`

project.handset.uses.Sensors

Specifies whether an application requires the Mobile Sensor API library for J2ME to compile and that you want Mobile Designer to initialize static values for the library.

Platforms All

Value
- `true` if the application uses the Sensors library.
- `false` if the application does not use the Sensors library. Specifying `false` reduces the final size of your project's binaries.

> **Important:** If the application uses the Sensors library, and you set the property to `false`, Mobile Designer cannot compile the application.

Default `true`

project.handset.uses.WebServices

Specifies whether an application requires the Web Service API library.

> **Note:** The JSON library is also part of the Web Service API library.

Platforms All

Value
- `true` if the application uses the Web Service API library.

- `false` if the application does not use the Web Service API library.

Default `false`

project.handset.uses.WMA

Specifies whether an application requires the Wireless Messaging API (WMA) library for J2ME to compile and that you want Mobile Designer to initialize static values for the library. WMA enables sending Short Message Service (SMS), Multimedia Messaging Service (MMS), and Cell Broadcast Service (CBS) formats.

Platforms All

Value
- `true` if the application uses the WMA library.
- `false` if the application does not use the WMA library. Specifying `false` reduces the final size of your project's binaries.

> **Important:** If the application uses the WMA library, and you set the property to `false`, Mobile Designer cannot compile the application.

Default `true`

Java Classes

cpp.no.extraexceptions

Specifies whether you want the application to perform the Java-required `NullPointer` and `ArrayIndexOutOfBounds` exception checks for every array access.

Platforms All

Value
- `true` if an application code-base is safe and does not need Java-required `NullPointer` and `ArrayIndexOutOfBounds` exception checks.
- `false` if you want an application to perform the Java-required `NullPointer` and `ArrayIndexOutOfBounds` exception checks.

Default `false`

project.javac.encoding

Specifies the character encoding used in your application's Java source code.

Note: Mobile Designer also supports other Java supported character sets such as UTF-8 and SJIS for the Shift-JIS, Japanese character set.

Platforms All

Value Character encoding

Default `8859_1` for ISO 8859-1, Latin Alphabet No. 1

Makefile Additions

Note: Makefiles are now deprecated and will be removed in a later version of Mobile Designer. At this point, Mobile Designer will instead make use of the generated project file to build the mobile application (i.e. Xcode for iOS builds).

Use the Makefile Additions properties to link to third-party native-platform libraries and/or to include more complex native alterations to your applications.

Note: Unless indicated otherwise, Mobile Designer duplicates your settings in the corresponding Visual Studio or Xcode project.

project.cpp.additional.compiler.options

Specifies extra compiler flags or settings that you want Mobile Designer to include when compiling an application.

Note: Mobile Designer does not include the flags and settings in the Visual Studio or Xcode project.

Platforms All

Value Flags and settings as required by the specified makefile for the target platform

Default An empty String

project.cpp.additional.defines

Specifies `#define` statements that you want Mobile Designer to use when compiling an application.

Platforms All

Value A semicolon-separated list of `#define` statements

Default An empty String

project.cpp.additional.includes.path

Identifies the name of an Ant path that includes a list of folders or files that you want Mobile Designer to reference for all C++ `#include` statements in a project.

Platforms All

Value Name of the Ant path

Default An empty String

project.cpp.additional.libs.path

Identifies the name of an Ant path that includes a list of folders containing additional libraries used in a project.

Platforms All

Value Name of the Ant path

Default null

project.cpp.additional.libs.post

Identifies libraries to add to the end of the referenced library list in the makefile. The libraries you specify with this property are placed after the default libraries that Mobile Designer automatically includes.

Platforms All

Value A semicolon-separated list of libraries

Do not include file extensions or makefile-specific encoding. Mobile Designer will set the appropriate value for a platform. For example, if you set `project.cpp.additional.libs.post` to `mylib1;mylib2`, Mobile Designer might change the value to "-lmylib1 –lmylib2" on one platform, and "mylib1.lib mylib2.lib" on a different platform.

Default An empty String

project.cpp.additional.libs.pre

Identifies libraries to add to the beginning of the referenced library list in the makefile. The libraries you specify with this property are placed before the default libraries that Mobile Designer automatically includes.

Platforms All

Value A semicolon-separated list of libraries

When specifying the libraries, do not include file extensions or makefile-specific encoding.Mobile Designer will set the appropriate value for a platform. For example, if you set `project.cpp.additional.libs.pre` to `mylib1;mylib2`, Mobile Designer might change the value to "-lmylib1 –lmylib2" on one platform, and "mylib1.lib mylib2.lib" on a different platform.

Default An empty String

project.cpp.additional.linker.options

Specifies linker flags or settings that you want Mobile Designer to include when generating the final binary and linking the compiled code with the defined external C++ libraries.

Note: Mobile Designer does not include the flags and settings that you specify in the Microsoft Visual Studio or Apple Xcode project.

Platforms All

Value Flags and settings as required by the specified makefile for the target platform

Default An empty String

project.handset.custom.stubfolder

Specifies a stub makefile and/or project template to override those that Mobile Designer supplies.

Platforms iOS

Value Path to the copy of the stub makefile and/or project template you want to use

Default An empty String

Optimization

project.in.code.manifest

Specifies whether you want Mobile Designer to delete the META-INF folder, which determines how your application can access the manifest properties.

Platforms All

Value
- `true` to keep the META-INF folder. The application can manually access the manifest.mf file using the following path:

  /META-INF/MANIFEST.MF
- `false` if you want Mobile Designer to delete the META-INF folder. The application can access the manifest properties using the MIDlet.getAppProperty method.

Default `true`

project.runtime.http.connection.timeout.ms

Specifies the number of milliseconds an application waits for an HTTP or HTTPS connection before assuming a timeout has occurred.

Platforms Android
iOS

Value Numeric value

Default The default value is based on the platform.

Platform	Default
Android iOS	`15000`

Screen and Display Handling

project.runtime.statusbar.visible

Specifies whether you want the application to leave room so that a device's status bar displays. The status bar is where a device displays information, such as battery strength, signal strength, and the time.

Platforms: Android, iOS

Value:
- `true` if you want the status bar visible.
- `false` if you do not want the status bar visible.

Default: Depends on whether the application uses the Native user interface library (com.softwareag.mobile.runtime.nui) for its user interface:
- If an application uses the NativeUI library, the default is `true`.
- If an application does *not* use the NativeUI library, the default is `false`.

Threading

project.handset.thread.stacksize

Specifies the number of bytes to allocate for the stack when the application creates a thread.

Platforms: All

Value: Numeric value

Default: The default is based on the platform:

Platform	Default

iOS	`524288`
Other platforms	`32768`

Android

android.apk

Specifies a file name that Mobile Designer uses to override the name for the Android application package file.

Platforms	Android
Value	File name for the Android application package file
Default	File name as specified by the Multi Build dialog

android.backkey.valid

Specifies the action to take when the user presses the Back key, that is, whether to handle the Back key like all other keys or to terminate the application.

Platforms	Android
Value	■ `true` to use the Back key like other keys. ■ `false` to terminate the application.
Default	`true`

android.manifest.permissions

Specifies permissions that you want Mobile Designer to include in the AndroidManifest.xml file.

Platforms	Android
Value	A semicolon-separated list of permissions
Default	An empty String

android.manifest.xml

Specifies an override for the default AndroidManifest.xml file that Mobile Designer generates.

Platforms Android

Value The path to an alternate XML file to use instead of the default AndroidManifest.xml file

Default An empty String

android.min.sdk.version

Specifies the value that you want Mobile Designer to insert into the AndroidManifest.xml file for the `minSdkVersion` setting, which defines the Android application programming interface (API) level.

Platforms Android

Value Value for `minSdkVersion`

Default `4`

android.orientation.forced

Specifies whether you want the application to force the screen orientation to portrait or landscape, or whether you want the application to allow the screen to automatically rotate based on the device's orientation.

Platforms Android

Value
- `portrait` to force the screen display to use portrait mode.
- `landscape` to force the screen display to use landscape mode.
- An empty string to allow the screen to automatically rotate.

Default An empty String

android.package

Specifies the name of the Java package that Mobile Designer uses for Android builds.

Platforms Android

Value Name of the Java package to use for Android builds

Default com.softwareag.mobile.runtime

android.package.name

Specifies a String to override the name for the Android application package (APK) file.

Platforms Android

Value String to use for the APK file name

Default Value of the `android.package` property with the selected.jarname appended to it

For example, if `android.package` is `com.softwareag.mobileruntime` and selected.jarname is `myproject`, the APK name would default to `com.softwareag.mobileruntime.myproject`.

android.studio.gradle.android.plugin.version

Specifies the version of the Gradle plugin to use when building Android projects. If this value is changed, it may also require a change in the version of Gradle that is used. This can be set with the additional property `android.studio.gradle.distribution.url`.

Platforms Android

Value An appropriate version number (e.g. `1.3.2`).

Default `2.2.2`

android.studio.gradle.distribution.url

Indicates the URL of the Gradle version to use to build this Android Studio project. Different versions of the Android builder for Gradle (set with `android.studio.gradle.android.plugin.version`) require different versions of Gradle. If the value is unset, then the value used in the project's `gradle/wrapper/gradle.properties` file is the default value specified by the current install of Android Studio.

Platforms Android

Value An appropriate URL.

Default `unset`. At the time of writing, this leads to Android Studio indicating a value of `https://services.gradle.org/distributions/gradle-2.14.1-all.zip`.

android.studio.selected.buildtools.version

This property will automatically point to the highest version of Android build tools provided with the Android SDK. It will be autodetected at compile-time.

Platforms Android

Value Name of any valid folder in the `build-tools` folder delivered with the Android SDK. Usually of the form `25.0.1` or `android-22.0.0`.

Default Will be set to the highest value available, taking the target sdk version into account.

android.studio.support.dependencies.root.dir

Gives the relative path from the root of the Android SDK installation to the Android support repository.

Platforms Android

Value Relative path pointing to the `support` folder.

Default `extras/android/m2repository/com/android/support`

android.studio.selected.dependency.version.appcompatv7

This property will automatically point to the highest version of the v7 app compatibility library installed in the Android SDK. It will be autodetected at compile-time, taking the `android.target.sdk.version` property into account. So, if `android.target.sdk.version` is `23`, possible values for `android.studio.selected.dependency.version.appcompatv7` will be of the form `23.x.y`.

Platforms Android

Value Name of any valid version folder in the `appcompat-v7` repository delivered with the Android SDK. Usually of the form `23.4.0`.

Default Will be set to the highest value available, taking the target sdk version into account.

android.studio.selected.dependency.version.design

This property will automatically point to the highest version of the design support library installed in the Android SDK. It will be autodetected at compile-time, taking the `android.target.sdk.version` property into account. So, if `android.target.sdk.version` is `23`, possible values for `android.studio.selected.dependency.version.design` will be of the form `23.x.y`.

Platforms Android

Value Name of any valid version folder in the `design support` repository delivered with the Android SDK. Usually of the form `23.4.0`.

Default Will be set to the highest value available, taking the target sdk version into account.

project.additional.libs.path

Specifies the name of an Ant path that includes a list of folders or direct file locations where required native-shared libraries are located.

Platforms Android

Value Name of an Ant path

Default `project.cpp.additional.libs.path`

project.android.sdk.version.override

Specifies the version of the SDK that you want to use for compiling.

Platforms Android

Value Version of the version of the SDK you want to use for compiling

Default `21`

android.push.additional.icon.pre21api

If set, nominates an additional icon that will be copied to the application as res/raw/pushiconpre21.png and used as the in-application notification icon for incoming Push Messages for devices with device API earlier than 21. If this parameter is unset (default), then the the icon set in "android.push.additional.icon" or the application's main icon will be used instead.

Platforms Android

Value Icon `res/raw/pushiconpre21.png`

Default not set

android.push.message.key

Determines the most important key within an incoming Push Message. This is the one that is returned to the application as the "primary message text" through MessageConnectionHandler.receivedTextMessage().

Platforms Android

Value Primary message text

Default `collapse_key`

android.push.additional.icon

If set, nominates an additional icon that will be copied to the application as res/raw/pushicon.png and used as the in-application notification icon for incoming Push Messages. If this parameter is unset, then the icon set in "android.push.additional.icon.pre21api" or the application's main icon will be used instead.

Platforms Android

Value Icon `res/raw/pushicon.png`

Default not set

android.push.icon.reference

This should not normally need to be set by the user. It will point to the icon used for Push Notifications within the application. Setting android.push.additional.icon will

automatically change this value to "R.raw.pushicon". Advanced usage may include the use of ternary operators or method calls to specify an int value from R.java.

Platforms Android

Value R.raw.pushicon

Default `R.mipmap.icon`

iOS

ios.app.delegate.name

Specifies the name of the application delegate that you want the application to use.

Platforms iOS

Value Name of an alternative delegate application

Default `xyzApp`

ios.background.music

Specifies whether the application allows users to continue playing their audio in the background while the application is running.

Platforms iOS

Value

- `true` to allow the user to play music, which is not a part of the application, in the background while the application is running.

  Specifying `true` also ensures that the audio supports the iPhone's Ring/Silent switch setting.
- `false` if you do not want to allow a user to play music while the application is running.

Default `false`

ios.deployment.target

Specifies the minimum iOS firmware version that the application supports.

Platforms iOS

Value Version number

When specifying a version, use the number only, such as `6.0`.

Default `6.0`

ios.info.plist.output.format

Specifies the format you want Mobile Designer to use for the final packaged Info.plist.

Platforms iOS

Value
- `binary` to maintain the Info.plist file in binary format.
- `xml` to maintain the Info.plist file in XML format.

Default `binary`

ios.retained.png.list

Specifies a list of portable network graphic (PNG) files that you want Mobile Designer to retain a .png file rather than generated _png files.

Platforms iOS

Value A semicolon-separated list of PNG files

Default An empty String

ios.sdk.version

Specifies the version of the iOS SDK that you want Mobile Designer to use when compiling the application.

Platforms iOS

Value Version number of the iOS SDK

Default Latest version of the SDK

Mobile Designer looks for the installed iOS SDK versions and selects the version with the highest version number

ios.use.retina.display

Specifies whether the application supports the high-resolution retina display of an iOS device.

Platforms iOS

Value
- `true` if the application supports the high-resolution retina display.
- `false` if the application does not support the high-resolution retina display.

Default `false`

Cross-Product Integration Properties

The cross-product integration properties configure how Mobile Designer works with other products in the webMethods product suite.

mobilesupportclient.runtime.dir

Specifies whether to include the Mobile Support Client library in a mobile project. The methods in this library facilitate data synchronization between mobile devices and back-end databases by initiating synchronization requests with webMethods Mobile Support.

For more information about webMethods Mobile Support, see *Developing Data Synchronization Solutions with webMethods Mobile Support*. For more information about the Mobile Support Client library, see *webMethods Mobile Support Client Java API Reference*.

Note: The Mobile Support Client library requires the com.softwareag.mobile.runtime.database classes. As a result, when you set a value for the `mobilesupportclient.runtime.dir` property, the project.handset.uses.Database is automatically set to `true` to indicate that the mobile application uses the com.softwareag.mobile.runtime.database classes.

Platforms All

Values Root directory where the Mobile Support Client library mdlibrary.properties file and src folder reside.

Example:

```
<property name="mobilesupportclient.runtime.dir"
          value="c:/SoftwareAG/Mobile/SupportClient" />
```

Default None.

Device-Specific Properties

The device-specific properties define information for specific devices in your project. Store these properties in your project's targets folder in the target device file for the device to which the property pertains, *device_name* .xml.

project.handset.*device_name* .langgroups

Specifies one or more language groups that indicate the language(s) that your mobile application supports for the device indicated in the property name.

When specifying the property, replace `device_name` in the property name with the name of a specific device. The value for a specific device is the value that you selected from the **Choose your handset** list in the Add Handset dialog when you added the device to the project. For example, for an Apple iPhone 5 phone, the property name is:

```
project.handset.IOS_Apple_iPhone5.langgroup
```

When you added the device to the project, Mobile Designer automatically added this property to the target device file in your project's targets folder.

project_folder /targets/*device_file* .xml

Mobile Designer set the value of the property to the language groups you specified in the **Language Groups** field of the Add Handset dialog. For more information, see "Adding a Device to a Project" on page 91.

You can update the property if you want to add, change, or remove language codes. You can specify the language groups you define with the project.langgroup.group_name property.

Platforms All

Value One or more language groups. To specify multiple language groups, create a semicolon-separated list.

Example

Suppose you use the project.langgroup.group_name property to define the following language groups:

```
<property name="project.langgroup.AMERICAN" value="en;fr;es"/>
<property name="project.langgroup.EUROPEAN"
value="en;fr;it;de;es"/>
<property name="project.langgroup.ASIAN" value="zh;ja"/>
```

To specify that the application supports the American and European language groups for the IOS_Apple_iPhone5 device, use the following:

```
<property name="project.handset.IOS_Apple_iPhone5.langgroup"
          value="AMERICAN;EUROPEAN"/>
```

Default No default.

This property must have a value.

project.manifest

Specifies the location of the manifest.mf file for a device.

Platforms All

Value Absolute path to the manifest file.

Default The default is based on the specific device and is set in the device's device profile. You can find the device profiles in the following location:

Mobile Designer_directory/Devices

Hook Point Properties

The hook point properties provide the names of Ant targets that you create and that you want Mobile Designer to execute when it is executing Ant targets that are provided with Mobile Designer, for example, the +Multi-Build Ant target. Using hookpoints allows you to customize Mobile Designer processes. For more information about hookpoints, see "Creating Custom Ant Scripts to Run at Predefined Hook Points" on page 125.

project.hookpoint.target.prebuildresources

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *before* it executes the Resource Handler while processing the executing the +Multi-Build, +Multi-Build-Last, +Activate-Handset, +Re-Activate-Handset, or +Run-Phoney-With-Re-Activation Ant tasks.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

project.hookpoint.target.postbuildresources

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *after* it executes the Resource Handler while processing the +Multi-Build, +Multi-Build-Last, +Activate-Handset, +Re-Activate-Handset, or +Run-Phoney-With-Re-Activation Ant task.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

project.hookpoint.target.precompilation

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *before* it performs platform-specific compilation for each build while processing the +Multi-Build or +Multi-Build-Last Ant task.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

project.hookpoint.target.postcompilation

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *after* it performs platform-specific compilation for each build while processing the +Multi-Build or +Multi-Build-Last Ant task.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

project.hookpoint.target.postcrosscompiler

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *after* the cross compiler has converted the project's source to your build's target language.

Note: Mobile Designer only calls this hook point for cross-compiled builds.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

project.hookpoint.target.premakefilegeneration

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *before* it generates the makefile and platform-specific project. Examples of platform-specific projects are an Apple Xcode project for the iOS platform, or an Eclipse project for platforms like Android.

Note: Mobile Designer only calls this hook point for cross-compiled builds.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

project.hookpoint.target.prepatch

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *after* it applies the patches to your code.

Note: Mobile Designer only calls this hook point for cross-compiled builds.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

project.hookpoint.target.postpackaging

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *after* it generates a platform-specific build bundle.

The output of the packaging process varies from platform to platform. For example, for Android builds, the output is in an application package (apk) file.

After Mobile Designer executes the Ant target specified by the `project.hookpoint.target.postpackaging` property, if appropriate for the platform, Mobile Designer performs any code signing.

Note: Mobile Designer only calls this hook point for cross-compiled builds.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

project.hookpoint.target.postbuild

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *after* it packages and signs a build.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

project.hookpoint.target.postmultibuild

Specifies an Ant target that you created. Mobile Designer calls the Ant target you specify *after* it creates all the builds you selected in the Multi-Build dialog.

Platforms All

Value Name of an Ant Target you defined

Default None. If you do not include this property, Mobile Designer does not invoke a hook point.

Multi-Build Selection Properties

The Multi-Build selection properties contain values that you selected when building the project. Some of the properties are also set when you activate a device in the project.

Mobile Designer sets these properties based on the selections you make in the Multi Build or Activate Handset dialogs. The properties are *not* set in either the _defaults_.xml file or a target device file. The properties are not saved to any file. As a result, you cannot override the settings.

You can use the properties in your Ant scripts so that at run time it can determine information about the current build Mobile Designer is processing or the current device being activated.

selected.handset

Specifies the device for which Mobile Designer is building the application or the device Mobile Designer is activating. In other words, the value is the name of the device you selected in the Multi Build or Activate Handset dialog, for example, `IOS_Apple_iPhone5`. Mobile Designer sets this property.

Platforms All

Value Device you selected in the Multi Build or Activate Handset dialog

Default n/a

selected.jarname

Specifies the name of the JAR file that is listed in the **Filename** field of the Multi Build dialog, for example, `NativeUIDemo_iPhone5_DE`. This is the JAR file name that Mobile Designer uses for the build. Mobile Designer sets this property.

Platforms All

Value JAR file name Mobile Designer is using for the build of the project

Default n/a

selected.langgroup

Specifies the language you selected in the Multi Build or Activate Handset dialog. This is the language for which Mobile Designer is building the application or activating a device. Mobile Designer sets this property.

Platforms All

Value Language you selected in the Multi Build or Activate Handset dialog

Default n/a

selected.platform

Specifies the name of the platform that is listed in the **Platform** field of the Multi Build dialog, for example, `ios-app`. This is the platform for which Mobile Designer is creating the build. Mobile Designer sets this property.

Platforms All

Value Platform for which Mobile Designer is creating the build of the project

Default n/a

selected.target

Specifies the type of executable being built. Mobile Designer sets this property using the value from the **Target** field of the Multi Build dialog, for example, `release`.

Platforms All

Value Type of executable listed in the **Target** field in the Multi Build dialog

Default n/a

selected.version

Specifies the version number for the application that you specified in the Multi Build dialog, for example, `1.0.0`. Mobile Designer sets this property.

Platforms All

Value Version number you specified in the Multi Build dialog

Default n/a

Phoney Properties

The Phoney properties customize the use of Phoney for the project.

phoney.base.params

Specifies Phoney startup options. The startup options you specify with this property are used when you start Phoney via Software AG Designer.

Platforms All

Value One or more of the startup options. For a list of the startup options, see "Phoney Startup Options" on page 138.

Default `-r -i -s 1 -rs 1 -so -ai -p ${project.temp.dir.root}/_build_info_.txt`

Project Language Properties

The project definition properties define settings for your project.

project.langgroup.*group_name*

Defines a language group that specifies one or more languages that your application supports. When specifying the property, replace *group_name* with the name you want to give the language group. Set the value of the property to one or more language codes that indicate the language(s) in the group. You can use this property multiple times to define multiple language groups.

For example, you might want to set up three language groups to define languages for different territories, an American territory, European territory, and Asian territory. To do so, specify the following:

```
<property name="project.langgroup.AMERICAN" value="en;fr;es"/>
<property name="project.langgroup.EUROPEAN" value="en;fr;it;de;es"/>
<property name="project.langgroup.ASIAN" value="zh;ja"/>
```

When displaying the Multi Build dialog for the project, Mobile Designer lists each language group. You can then select from those language groups to identify the languages for which you want to create a build.

When specifying the project.handset.device_name.langgroups property to identify the language groups that a specific device supports, you set the value to one or more language groups you define using this `project.langgroup.`*group_name* property.

Platforms All

Value One or more of the following language codes to define the languages in the group. To specify multiple languages, use a semicolon-separated list.

Default None.

Note: You must specify this property for a project.

Resource Handler Properties

The resource handler properties provide information about the project's resource handler. For more information about how you define the resource handler for a project, see "Defining Resources for a Mobile Application Project" on page 73.

debug.remember.resource.names

Specifies whether you want the Mobile Designer to record the names of the resources included in the build.

Platforms All

Value
- `true` if you want to record the names of the resources.

  When the property is set to `true`, the Mobile Designer run-time debug output references these names directly, instead of the ID numbers.

  Setting the property to `true` results in extra code and a larger data pool resulting from all the resource names that are stored in the final binary.
- `false` if you do *not* want to record the names of the resources.

  You should set the property to `false` when preparing a release build.

Default `false`

project.audio.file.extensions

Specifies the file extension required for the audio files that a device supports.

Platforms All

Value File extension for the audio, for example `.mp3`, `.wav`, or `.mid`.

Default The default is based on the specific device and is set in the device's device profile. You can find the device profiles in the following location:

Mobile Designer_directory/Devices

project.audio.spec

Specifies the style of audio that a device supports.

Platforms All

Value Style of audio, for example `mp3`, `wav`, or `midi`.

Default The default is based on the specific device and is set in the device's device profile. You can find the device profiles in the following location:

Mobile Designer_directory/Devices

project.compiled.resources.info.format

Specifies the file format that you want Mobile Designer to use for the _compiled_resources file. Mobile Designer saves this file in a project's _temp_ directory.

Note: Set this property in the project's _default_.xml file, and *not* in a specific target device file, *target_name* .xml.

Platforms All

Value
- `txt` if you want Mobile Designer to create a text format compiled_resources file (_temp__compiled_resources.txt).
- `xml` if you want Mobile Designer to create an XML format compiled_resources file (_temp__compiled_resources.xml).

Default `txt`

project.jar.midlet.icon.spec

Specifies the icon(s) to use for the application's MIDlet-icon for a specific device.

Platforms All

Value Name of the portable network graphic (.png) file(s) to use for the application's MIDlet-icon.

Use this property to include the required icon(s) so that the icon for your application when it is installed on the target devices meets your company and/or application branding requirements. A best practice is to name the icons so that the name includes the size of the icon, for example icon-64x64-24bit.png. Store the image files in a subfolder that your resource handler can easily reference.

Default The default is based on the specific device and is set in the device's device profile. You can find the device profiles in the following location:

Mobile Designer_directory/Devices

In the device profiles that Mobile Designer provides, the provided icons are Software AG and Mobile Designer icons.

project.java.reshandler.name

Specifies the Java package/class name of the resource handler class you created for your project. For more information about creating a resource handler for your project, see "Coding the Resource Handler" on page 74.

Platforms All

Value Name the Resource Handler class name.

Default None.

Note: You must specify this property for a project.

project.reshandler.additional.libs.path

Specifies the Ant path to additional libraries that your project's resource handler requires. This is not an Ant property. Use the following format to specify the Ant path:

```
<path id="project.reshandler.additional.libs.path">
          <pathelement path="path"/>
</path>
```

For example:

```
<path id="project.reshandler.additional.libs.path">
          <pathelement path="${basedir}/reshandler/libs"/>
</path>
```

Platforms All

Value Path to the additional libraries for the project's resource handler

Default None.

Note: If you use additional libraries for the resource handler, you must specify this Ant path for a project.

project.reshandler.src.path

Specifies the path to the project's resource handler script and any associated classes. In other words, the path to the Java class you specify with the project.java.reshandler.name property.

This is not an Ant property. Use the following format to specify the Ant path:

```
<path id="project.reshandler.src.path">
          <pathelement path="path"/>
</path>
```

For example:

```
<path id="project.reshandler.additional.libs.path">
          <pathelement path="${basedir}/reshandler"/>
</path>
```

Platforms All

Value Path to the Java class specified by the project.java.reshandler.name property

Default None.

Note: You must specify this Ant path for a project.

project.resource.dir.root

Specifies the path to the top-level folder that contains the resources (image files, etc.) for your project. For example, you might have a resources folder that contains subfolders with the resources:

```
MyProject
    resources
        graphics
        icons
        text
```

For this example, set the property to point to the top-level folder, "${basedir}/resources". For more information about how to specify resources for your project, see "Setting Project Properties for the Resource Handler" on page 78.

Platforms All

Value Path to the top-level folder that contains the project resource files.

Default None.

Note: You must specify this property for a project.

Run-Time Classes Properties

The run-time classes properties customize how the project uses the run-time classes that Mobile Designer provides.

mobiledesigner.runtime.core.class.camera

Overrides the value that Mobile Designer sets for how mobile application uses a device's camera.

Note: The com.softwareag.mobile.runtime.media.CameraHandler class uses this property. For more information, see "Run-Time Media Classes" on page 52.

Platforms All

Value
- `none` if your application does not use the camera, or specific devices that your project supports do not support a camera.
- `jsr135` specifies your application uses the Mobile Media API (MMAPI) package for J2ME devices that support the camera.

Default The default is based on the specific device and is set in the device's device profile. You can find the device profiles in the following location:

Mobile Designer_directory/Devices

mobiledesigner.runtime.core.class.comms.messageconnection

Overrides the value that Mobile Designer sets for whether the application uses SMS messaging.

Note: The com.softwareag.mobile.runtime.comms.MessageConnectionHandler class uses this property. For more information, see "Run-Time Comms Classes" on page 51.

Platforms All

Value
- `none` if your application or specific devices that your project supports are not support SMS messaging.
- `wma` if your mobile application is using the J2ME Wireless Messaging API for SMS messaging.

Default The default is based on the specific device and is set in the device's device profile. You can find the device profiles in the following location:

Mobile Designer_directory/Devices

mobiledesigner.runtime.core.class.datatypes

Overrides the value that Mobile Designer sets to determine which primitive data types can be read when parsing your resources with the run-time com.softwareag.mobile.runtime.storage.ResourceDataTypes class.

Note: The com.softwareag.mobile.runtime.storage.ResourceDataTypes class uses this property. For more information, see "Run-Time Storage Classes" on page 54.

Platforms All

Value
- `cldc11` specifies your mobile application uses CLDC 1.1.

  CLDC 1.1 supports integer numbers, float and double.

Default The default is based on the specific device and is set in the device's device profile. You can find the device profiles in the following location:

Mobile Designer_directory/Devices

mobiledesigner.runtime.core.class.serialize

Overrides the value that Mobile Designer sets for com.softwareag.mobile.runtime.serialize.Serializer class.

Note: The com.softwareag.mobile.runtime.serialize.Serializer class uses this property. For more information, see "Run-Time Serializer Class" on page 53.

Platforms All

Value
- `cldc11` specifies your mobile application uses CLDC 1.1.

  CLDC 1.1 supports integer numbers, float and double.

Default The default is based on the specific device and is set in the device's device profile. You can find the device profiles in the following location:

Mobile Designer_directory/Devices

Run-Time Code Compilation Properties

The run-time code compilation properties customize how Mobile Designer compiles your project.

project.runtime.additional.classes.path

Specifies the Ant path to additional precompiled classes to include when building the project. This is not an Ant property. Use the following format to specify the Ant path:

```
<path id="project.runtime.additional.classes.path">
          <pathelement path="path"/>
</path>
```

For example:

```
<path id="project.runtime.additional.classes.path">
          <pathelement path="${basedir}/src/classes"/>
</path>
```

Platforms All

Value Path to the folder that contains the additional precompiled classes.

Default None.

> **Note:** If you use additional precompiled classes, you must specify this Ant path for a project.

project.runtime.additional.stubs.path

Specifies the Ant path to additional stubs that you want Mobile Designer to use when compiling the run-time source code. This is not an Ant property. Use the following format to specify the Ant path:

```
<path id="project.runtime.additional.stubs.path">
          <pathelement path="path"/>
</path>
```

For example:

```
<path id="project.runtime.additional.stubs.path">
          <pathelement path="${basedir}/stubs"/>
</path>
```

> **Important:** This path is used in combination with default classpath entries that Mobile Designer defines. If you set this Ant path, be careful to avoid any duplicate class clashes.

Platforms All

Value Path to the folder that contains the stubs.

Default None.

Note: If you want to use additional stubs, you must specify this Ant path for a project.

project.runtime.project.src.path

Specifies the Ant path to the run-time code to include in the build of the project. This is not an Ant property. Use the following format to specify the Ant path:

```
<path id="project.runtime.project.src.path">
          <pathelement path="path"/>
</path>
```

For example:

```
<path id="project.runtime.project.src.path">
          <pathelement path="${basedir}/src"/>
</path>
```

Platforms All

Value Path to the folder that contains the run-time code.

Default None.

Note: You must specify this Ant path for a project.

B Ant Target Summary

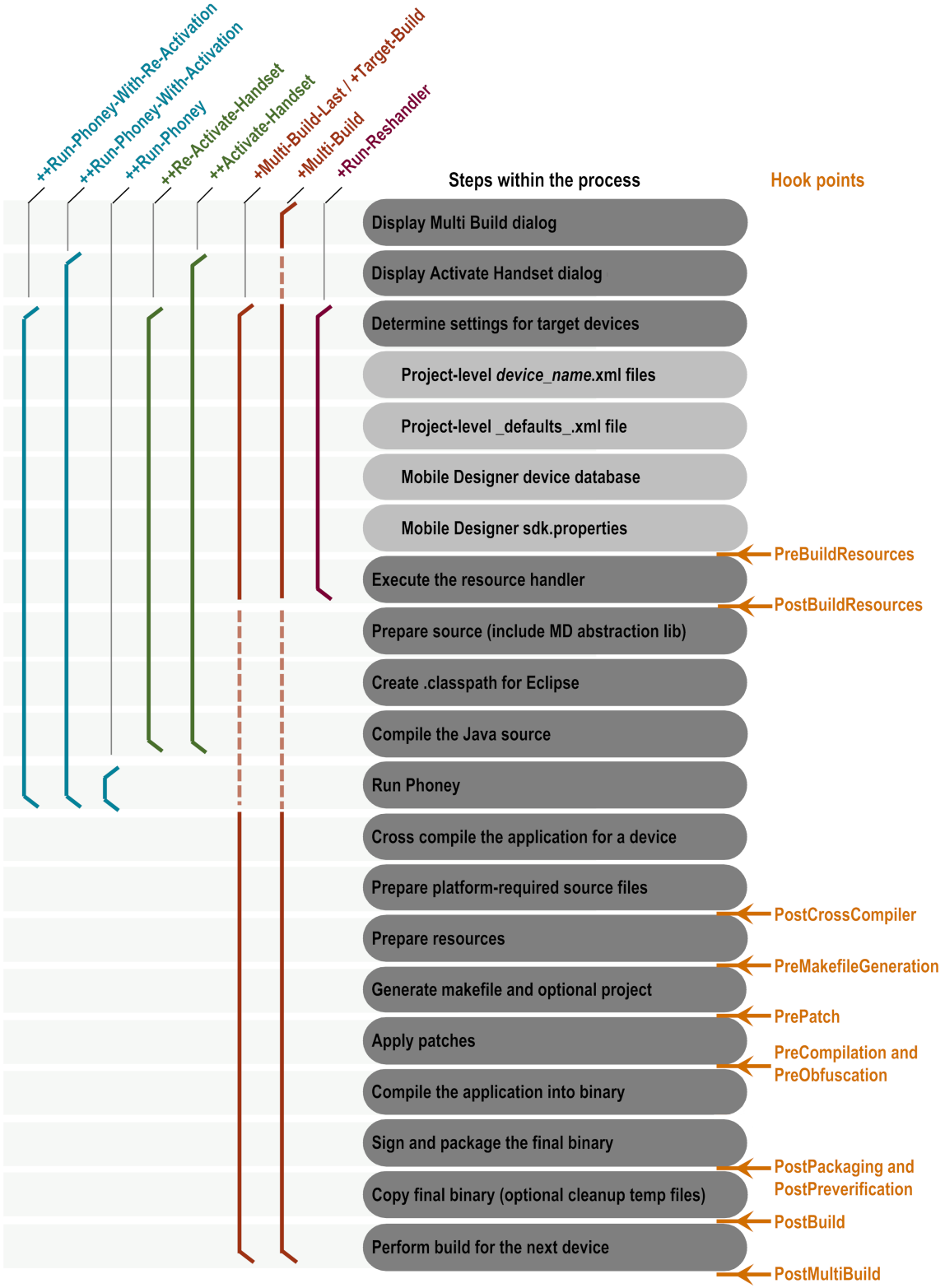
Ant Target Summary

This section provides a diagram that shows the Ant targets you can use to compile resources, build a mobile application project, activate a device, and use Phoney.

For each Ant target, the diagram indicate the steps Mobile Designer performs for an Ant target. For example, for the ++Run-Phoney Ant target, Mobile Designer only performs the "Run Phoney" step. Dashed lines indicate steps that Mobile Designer does not perform. For example, for the +Multi-Build Ant target, there is a dashed line for the "Display Activate Handset dialog" step because Mobile Designer does not perform this step when running the +Multi-Build Ant target. For more details about the steps, see one of the following:

- For details about compiling resources only, see "Compiling Resources Using the +Run-Reshandler Ant Target" on page 80.
- For details about the build process, see "Steps in the Multi-Build Process" on page 112.
- For details about the process to activate a device, see "Steps Performed to Activate Handsets" on page 148.
- For details about the actions taken when you run Phoney, see "Steps Performed for Phoney Ant Targets" on page 136.

Use this diagram to compare the actions performed for each Ant target.


++Run-Phoney-With-Re-Activation
++Run-Phoney-With-Activation
++Run-Phoney
++Re-Activate-Handset
++Activate-Handset
+Multi-Build-Last / +Target-Build
+Multi-Build
+Run-Reshandler
Steps within the process
Hook points
Display Multi Build dialog
Display Activate Handset dialog
Determine settings for target devices
Project-level device_name.xml files
Project-level _defaults_.xml file
Mobile Designer device database
Mobile Designer sdk.properties
PreBuildResources
Execute the resource handler
PostBuildResources
Prepare source (include MD abstraction lib)
Create .classpath for Eclipse
Compile the Java source
Run Phoney
Cross compile the application for a device
Prepare platform-required source files
PostCrossCompiler
Prepare resources
PreMakefileGeneration
Generate makefile and optional project
PrePatch
Apply patches
PreCompilation and
PreObfuscation
Compile the application into binary
Sign and package the final binary
PostPackaging and
PostPreverification
Copy final binary (optional cleanup temp files)
PostBuild
Perform build for the next device
PostMultiBuild