Maui 1.2
Developer's Guide

Author: **Dave Lapointe (dave@bitmovers.com)**
Contributors:

Version: **0.0.1**
Date: **2001.09.24**

Abstract: **This is a starting document for Maui application developers. It describes everything a developer needs to know about Maui.**

Table of Contents

1.0 Introduction

This is a document for Maui application developers. It contains everything you will need to get started building Maui applications. It also provides details on the more advanced capabilities of Maui.

The intended audience for this document is anyone who is developing or designing Maui applications. There is no specific requirement for the reader to be proficient in Java. However, knowledge of Java will help in understanding some of the contents and examples.

Also, if you have a Java environment (and an IDE) at your immediate disposal, then it may help to try out some of the examples.

1.1 Software Environment

- Java 2 SDK Environment (1.2 or greater)
- Optionally, an Integrated Development Environment. Maui was built using CodeWarrior (http://www.metrowerks.com) on Macs, and deployed on Linux. Although any IDE that supports Java will work, the examples were built using CodeWarrior. Some examples include the SDK's command line based syntax.

2.0 Overview

Following is a simple overview Maui, including folder descriptions and the Maui component environment.

2.1 Maui Folder Hierarchy and Files

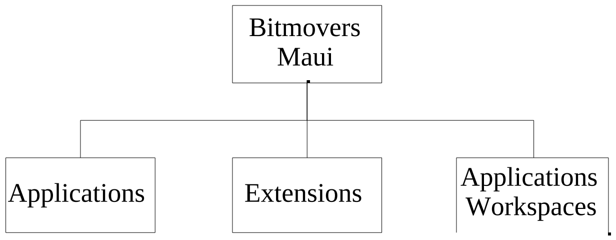

Figure 1 – Maui Folder Hierarchy

Folder	**Description**
BitmoversMaui	This is the main folder of the hierarchy. The Maui runtime (MauiRuntime.jar), plus supporting files are contained here.
Applications	This folder contains Maui applications. This folder can contain sub-folders. Maui applications are contained in .jar files.
Extensions	This folder contains jar files for libraries that are requirements for some Maui applications. For example, for a Maui application using the Java Mail API, the mail.jar file would be stored here. (Alternatively, it is possible to store the extension jar files in $JAVA_HOME/jre/lib/ext.)
Application Workspace	This is a hierarchy of folders that provides private workspaces for Maui applications. Each folder can be used exclusively by its respective Maui application. If the folders don't already exist, each application's workspace will be created by Maui when the application is loaded. The folder hierarchy is derived from the application's package and name (eg. for an application named "com.my.env.FooBar" a workspace folder hierarchy of the form "ApplicationsWorkspaces/com/my/env/FooBar" will be created. The name of the application's workspace can be retrieved through the property "maui.user.dir" from within the MauiApplication.

2.2 Maui Overview

The following diagram provides an overall perspective on where Maui and Maui applications fit in the request/response chain. To prevent clutter, this diagram is not comprehensive. Rather, it focuses only on the components involved in handling client requests and responses.

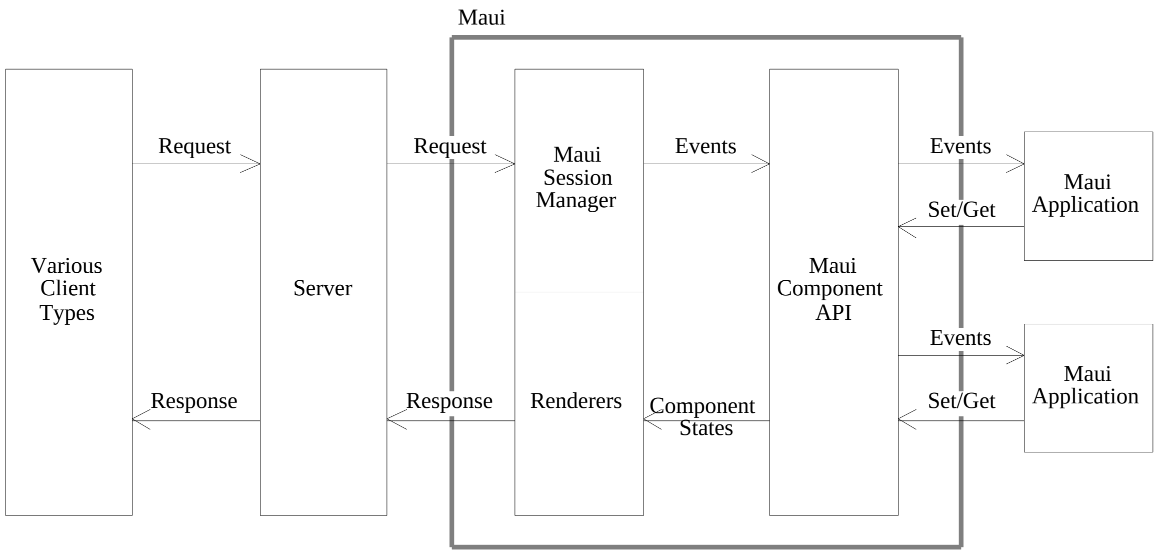

Figure 2 – Maui Overview

Item	Description
Various Client Types	These are the client environments that are communicating with a server environment. At present, Maui supports many types of HTML and WML browsers. It also supports Palm WebClipping clients.
Request and Response	These are the HTTP based communications. HTTP Requests are passed to a web server and forwarded to Maui. And Maui renders HTTP based Response messages.
Server	This is any HTTP server. Because Maui includes an HTTP server, it is not necessary to use an external server in order to run Maui. Maui can also function as a Servlet. This is documented in the Maui Servlet Support document.
Maui Session Manager	This subsystem is responsible for many steps: - Creating Maui applications - Converting requests to events - Forwarding events to application components - Driving the renderers
Events	These are the event representations of the HTTP Request. All events are derived from the MauiEvent class.
Maui Component API	This is the API that is used by Maui applications. Maui components are created through the API. Many

	components are "live" and are capable of receiving events, and publishing them to event listeners in the respective Maui applications.
Maui Applications	These are the various Maui applications. This can be any application that is contained in a jar file within the Applications folder hierarchy.
Set/Get	Maui applications extract component states using standard "get" methods, and change component states using standard "set" methods. Note that state is maintained in the components, and that the Maui application is not required to track any state information.
Component States	This is the component state information that is retrieved by the renderer libraries, and is used to represent components to the various clients.
Renderers	Based on the client type, a renderer is located and bound to each renderable Maui component. The renderer is responsible for generating a component representation that is appropriate for the client. The renderer selection process involves matching the best renderer available for a particular client type. This can include such things as distinguishing between browser versions or browser types (eg. IE 5 vs. Communicator 4.7). The Session Manager is responsible for driving the renderers, and accumulating the response to be sent to the client.

2.3 Maui Environment

With some minor differences, the Maui environment is structured much like Java's AWT or Swing environments. Maui has "widget" like components (eg. Buttons, Labels, TextFields), different types of container components (eg. Frames, Panels, Tabbed Panes), and layout managers. Anyone who has developed AWT or Swing based applications will have little difficulty transferring the concepts to the Maui environment.

2.3.1 Application Structure

The following diagram shows the basic structure of a Maui application:

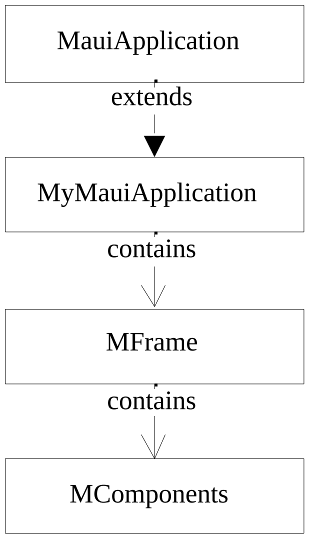

Figure 3 – Maui Application Structure

Item	Description
MauiApplication	This is the class from which all Maui applications must extend. Since a Maui application extends the MauiApplication class, it does not require a "main" method. Rather it is instantiated and activated through the Maui runtime environment. This is the main point of contrast with AWT or Swing based applications… they must have a "main" method, and are invoked through the Java VM.
MyMauiApplication	This is the Maui application class. It can be any name. Care must be taken that is not the same name as another Maui application in the Applications folder. To avoid this possible ambiguity, it is recommended that all applications have complete package names. The MauiApplication class extension (eg.

	MyMauiApplication) functions mostly as the "model" layer in a Model-View-Controller type application. It is responsible for creating the initial As such, it has no GUI representation. The UI is defined through the contained MFrame.
MFrame	This is the primary display container. This contains the actual UI components, including other Maui component containers. Although a Maui application can work with any number of MFrames, it can contain only one MFrame at a time. In other words, only one MFrame can be displayed at a time. However, the current MFrame can be replaced with any other MFrame at any time… the next time a render occurs, the current MFrame, and its contents, will be rendered.
MComponents	These are all other components. They are contained within each MFrame and fully represent to the client one view into the Maui application. The contents of any MFrame can change at any time, and the new view will be presented to the client the next time a render occurs. Note that this means that the UI is completely dynamically defined. It is not necessary to define any templates.

2.3.2 Maui Packages

There are several Maui packages that must be included in a Maui application. The detailed documentation for each package is available at http://maui.bitmovers.com/documents/apidocs.

Package	**Description**
com.bitmovers.maui	This package contains the MauiApplication class. Since all Maui applications extend MauiApplication, this package must be included. Alternatively, an explicit package reference can be made when the class is extended.
com.bitmovers.maui.components.foundation	This package contains all of the UI components.
com.bitmovers.maui.layouts	This package contains all of the layout managers.
com.bitmovers.maui.events	This package contains all of the event interfaces and event object classes.

2.3.3 Maui Components

MComponent	**Description**
MButton	The MButton is one of the most commonly used Maui components. It represents a simple, clickable button.
MCheckBox	The MCheckBox class represents a checkbox, that is, an element which

	allows one to toggle between two states.
MDivider	MDivider is a horizontal line component. The divider is useful for organizing the user interface into logically grouped areas.
MExpandPane	This class is similar to the triangular widgets found in the list view of the Macintosh finder. Clicking on this component causes it to expand and display previously hidden components.
MImage	This class is a renderable image component. It can display a stored image from a given resource location.
MLabel	This class is a basic text component, useful for creating user interface labels or general text messages. This component can optionally be given a URL to turn it into a link.
MMenu	This class is the menu component, which represents a list of clickable menu items.
MMenuBar	This class is the menu component, which represents a list of clickable menu items.
MMenuItem	This class is represents a menu item, which is the basic clickable element in the menu system. To handle menu item clicks, you should catch events published by MMenuItem objects.
MPanel	MPanel is the simplest of all containers. It has no visual representation, but is useful for grouping and aligning components.
MRadioButton	This is a radio button. It belongs to an MRadioButtonGroup.
MRadioButtonGroup	This defines a group of radio buttons.
MSelectList	This class represents a pull-down select list type component, or the closest approximation on any supported device.
MTabbedPane	This class is a very useful container that can have several overlapping panels. It is similar to the tabbed panes used by most desktop operating systems.
MTable	MTable is a special container for tabular data.
MTextArea	This class is a multiple line text field component.
MTextField	This class is a basic, one line editable text field.

2.3.4 Layout Managers

Maui contains a few layout managers. These work similarly to layout managers in AWT and Swing. As much as the client browser is capable, Maui will attempt to layout the components according to the declared layout managers.

Detailed descriptions for the layout managers are available in the online API documentation at http://maui.bitmovers.com/documents/apidocs. Currently, the layout managers are:

Layout Manager	**Description**
MBorderLayout	This works like the BorderLayout in AWT. It has 5 layout areas: NORTH, SOUTH, EAST, WEST and CENTER.

MBoxLayout	The MBoxLayout class is based on AWT's BoxLayout layout manager. It arranges components in an unbroken line, either vertically or horizontally, depending on the axis constant used during construction (X_AXIS or Y_AXIS)
MFlowLayout	The MFlowLayout class is based on AWT's FlowLayout layout manager. It arranges components like text in a paragraph (i.e. sequentially, left to right, top to bottom).
MGridLayout	This layout lays things out in a grid, with a set number of columns and rows. Components are added left to right, top to bottom, much like the java.awt.GridLayout layout manager.

2.4 Code Example

```
package com.myExamples;

import com.bitmovers.maui.*;
import com.bitmovers.maui.components.foundation.*;
import com.bitmovers.maui.layouts.*;

public class HelloWorld extends MauiApplication
{
      public HelloWorld (Object aInitializer)
      {
            //
            //    All Maui applications must call the super constructor.
            //    The "aInitializer" object is used as a call back to the
            //    Maui Runtime to complete initialization of the MauiApplication
            //
            super (aInitializer, "Hello World");

            //
            //    Create an MFrame.
            //    This will contain all of the components
            //
            MFrame theFrame = new MFrame ("Hello World");

            //
            //    Set the layout manager.
            //
            theFrame.setLayout (new MFlowLayout (theFrame));

            //
            //    For fun, let's put the Hello World text inside an expand pane
            //
            MExpandPane theExpandPane = new MExpandPane ("Expand me");
            theExpandPane.add (new MLabel ("Hello World"));
            theFrame.add (theExpandPane);

            //
            //    Finally, add the frame to the Maui application
            //
            add (theFrame);

            //
            //    Nothing else is required.  The rendering engine will
            //    handle everything from here.
      }
}
```

3.0 Maui Events

As indicated in the overview, Maui converts HTTP messages into component level events. Maui's event sub-system is based on the standard event model defined in Java. All Maui events extend the java.util.EventObject class and all Maui listener interfaces extend the java.util.EventListener interface. This means that the Maui event sub-system should be familiar to any Java developer who has used AWT, Swing, JavaBeans, or many other Java technologies.

All MComponent events are described through a single EventObject class – MActionEvent. And there is only a single EventListener interface that is used by all MComponents – MActionListener. Since all events can be described through this mechanism, it greatly simplifies the entire event mechanism, but without any loss of information.

It is because of Maui's event delivery sub-system that there is no requirement to define forms templates in Maui applications, as is the case with XSL based transcoder systems.

This section provides details of Maui's event delivery sub-system, plus a working code example.

3.1 Event sub-system

The following sequence diagram shows the object activation sequence. It delineates the steps involved in creating MComponent level events, dispatching them to their respective MComponents, and publishing the MActionEvents to the registered MActionListeners.

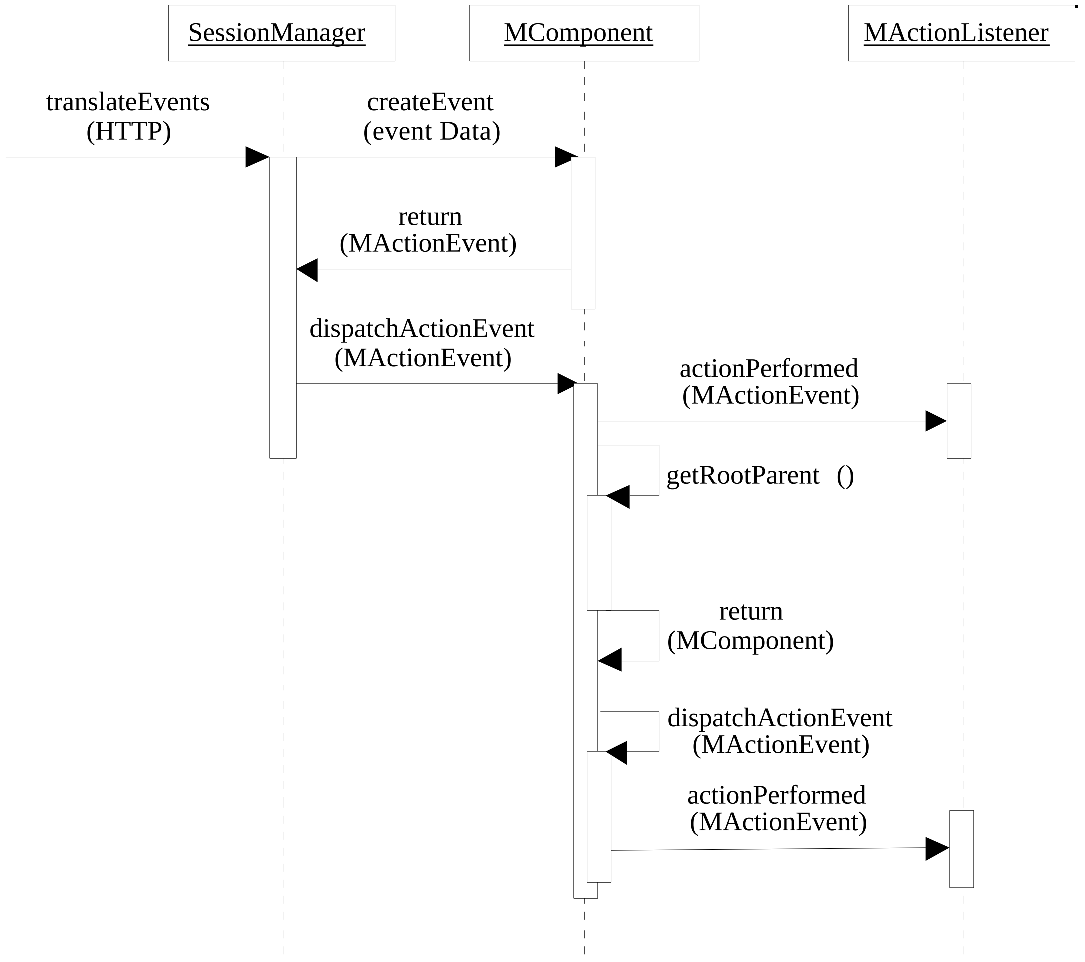

Figure 4 – Event Sequence Diagram

The steps in this figure are:

- An HTTP request is forwarded to the Session Manager.
- The Session Manager decomposes the request into event data for each MComponent to create an MActionEvent.
- The associated MComponent analyzes the event data and creates an MActionEvent. At any point after the MActionEvent is created, it can be consumed, which will prevent any further propagation of the event.
- If the event is not consumed, the MActionEvent is dispatched to the associated MComponent.

- While the MActionEvent isn't consumed, the MComponent publishes the event to each registered MActionListener.
- Once publication is completed and if the event is not consumed, the root parent MComponent (the MauiApplication) is retrieved.
- The event is dispatched to the MauiApplication. Since there is only one MauiApplication object for each Maui application, it means that any MActionListener that registers with MauiApplication will be notified of all MActionEvents for the application (unless the event was consumed earlier).
- While the MActionEvent isn't consumed, the MauiApplication publishes the event to each registered MActionListener.

3.1.5 Adding and Removing MActionListeners

All MActionListener objects register as event listeners with the target components using the "addActionListener" method. Conversely to remove an MActionListener the "removeActionListener" is used. See the example below.

3.2 Event Classes and Interfaces

Class/Interface	Description
MauiEvent	This is an abstract class that serves mostly as a placeholder. It extends EventObject.
MActionEvent	This descendant of MauiEvent is the single EventObject that describes all possible MComponent events. No other EventObjects are emitted by the MComponents.
MauiEventListener	This is an abstract interface that serves mostly as a placeholder. It extends EventListener.
MActionListener	This descendant of MauitEventListener is the single EventListener that is used by MComponents. No other EventListener interfaces are used by the MComponents.

3.3 Code Example

```
package com.myExamples;

import com.bitmovers.maui.*;
import com.bitmovers.maui.components.foundation.*;
import com.bitmovers.maui.layouts.*;

public class HelloWorld extends MauiApplication
{
      protected int counter = 0;

      public HelloWorld (Object aInitializer)
      {
            //
            //    All Maui applications must call the super constructor.
            //    The "aInitializer" object is used as a call back to the
            //    Maui Runtime to complete initialization of the MauiApplication
            //
            super (aInitializer, "Hello World");

            //
            //    Create an MFrame.
            //    This will contain all of the components
            //
            MFrame theFrame = new MFrame ("Hello World");

            //
            //    Set the layout manager.
            //
            theFrame.setLayout (new MFlowLayout (theFrame));

            //
            //    For fun, let's put the Hello World text inside an expand pane
            //
            MExpandPane theExpandPane = new MExpandPane ("Expand me");
            theExpandPane.add (new MLabel ("Hello World"));
            theFrame.add (theExpandPane);

            //
            //    Now lets add a button that will do some counting.
            //
            final MLabel theLabel = new MLabel ("No button presses");
            final MButton theButton = new MButton ("Button");

            //
            //    Register an MActionListener to listen for button presses.
            //    Increment a counter and update the MLabel with the count.
            //
            final MActionListener theActionListener = new MActionListener ()
                  {
                        public void actionPerformed (MActionEvent aEvent)
                        {
                              counter++;
                              theLabel.setText ("Button pressed " + counter + " times.");
                        }
                  };
            theButton.addActionListener (theActionListener);

            theFrame.add (theButton);
            theFrame.add (theLabel);

            //
            //    Now add a simple check box to handle add and removing the
            //    MButton's MActionListener.
            //
            final MCheckBox theCheckBox = new MCheckBox ("Listen on event", true);
            theCheckBox.addActionListener (new MActionListener ()
                  {
                        public void actionPerformed (MActionEvent aEvent)
                        {
                              if (theCheckBox.isChecked ())
                              {
                                    theButton.addActionListener (theActionListener);
                              }
                              else
                              {
                                    theButton.removeActionListener (theActionListener);
                              }
                        }
                  });
            theFrame.add (theCheckBox);

            //
            //    Finally, add the frame to the Maui application
            //
            add (theFrame);

            //
            //    Nothing else is required.  The rendering engine will
            //    handle everything from here.
      }
}
```

4.0 Properties

There are many runtime properties that can be used to configure and tweak Maui, and Maui applications. These properties range from basic settings like port numbers, to strategies for optimizing Maui's performance.

This section describes the different types of properties, and how they are used. However, it is not a comprehensive discussion of these properties. For details, see Appendix ii.

4.1 Maui Properties

These properties define characteristics and capabilities of the Maui runtime environment. They are read once at startup time and used to configure various Maui sub-systems and managers. They are not used after Maui initialization has completed. They are defined in the file "maui.properties", which is located in the same folder as the MauiRuntime.jar file.

It is also possible to override the settings from maui.properties by specifying runtime parameters. For example, if the property "maui.port", which specifies the server port number, is declared in maui.properties to be 9090, then this can be overridden to another port number using a command like:

```
java –cp MauiRuntime.jar com.bitmovers.maui.MauiRuntimeEngine maui.port=9091
```

This can be used for any maui property.

NB: There are many properties that are used for tuning Maui's performance. A discussion of these properties is outside the scope of this document.

4.2 Application Properties

These are properties that are specific to an application. With the exception of a few properties that are set by Maui, they are defined in properties files or resources, and loaded by Maui when the application is loaded. The properties are then accessible through the "setProperty" and "getProperty" methods in MauiApplication.

Since an instance of a MauiApplication is capable of changing its property settings, the Maui runtime only provides any instance of a MauiApplication with a clone of the application level properties. This is to ensure that property changes made by one instance of a MauiApplication do not affect the properties table of another instance of the same MauiApplication.

4.2.6 Predefined application properties

There are two application level property that have a specific usage within Maui. They are:

- maui.user.dir – This property is set by Maui, and refers to a working folder for the current application. All working folders reside under the ApplicationsWorkspace folder. Any files created by the application that are specific to the application should use this property as a prefix for locating the file. Note, that all instances of the same Maui application will share this folder space.

- maui.application.invisible – This property is set by the application, and is used by Maui's Application Manager. With this property set to true, the ApplicationManager will not keep the name of this application private. However, the application can still be accessed by explicitly typing in the URL to access. This shouldn't be considered to be a security measure.

4.3 Special Property Tables

There are two special property tables that are available to all Maui applications. These are:

- Session properties – A session object is maintained for any connection with a single client. A single session can contain any number of Maui applications; whenever a client requests requires the creation of a new application, it will be registered with the session associated with the client. If there is a requirement for different applications within a session to share information then use the session level properties table. The properties can be retrieved and set through the MauiApplication methods "getSessionProperty" and "putSessionProperty". For API details see http://maui.bitmovers.com/documents/apidocs/.
- Global properties – These are properties to be shared throughout the Maui runtime environment (across sessions and Maui applications). This is comparable to static variables, but with one significant difference; Class objects can be candidates for garbage collection. And when a Class is garbage collected, its static variables are lost. This means that the static variable values may not be permanent. However, using Maui's global properties, it can be guaranteed that the values stored in the global table will never be garbage collected. The properties can be retrieved and set through the MauiApplication methods "getGlobalProperty" and "putGlobalProperty". For API details see http://maui.bitmovers.com/documents/apidocs/.

4.4 Property Loading

To make Application property loading as flexible as possible, the process involves many cumulative steps. The same property table is used in each loading step. This means that the properties in the table can be accumulated from step to step. It also means that previously defined properties can be overridden at subsequent steps. The steps are:

- First check for the file "<jar file>.properties". For example, for a Maui application jar file named "foo.jar", look for the file "foo.properties".
- Look in the jar file for the resource "<jar file>.properties". If it is found, then load it into the existing property table. Continuing from the previous example, if "foo.jar" contains a resource called "foo.properties", it will be loaded into the property table.
- Look for the file "<jar file>.<class name>.properties". If it is found, then load it into the existing property table. For example, if the Maui application in "foo.jar" is named "MyHelloWorld", and the file "foo.MyHelloWorld.properties" exists, then load it into the property table.
- Look for the file "<package name>.<application name>.properties". If it is found, then load it into the existing property table. For example, if the full Maui application name is "com.myexamples.MyHelloWorld", and the file "com.myexamples.MyHelloWorld.properties" exists, then load it into the property table.

- Look for the file "<application name>.properties". If it is found, then load it into the existing property table. For example, if the Maui application name is "MyHelloWorld", and the file "MyHelloWorld.properties" exists, then load it into the property table.
- Look in the jar file for the resource "<jar file>.<application name>.properties". If it is found, then load it into the existing property table. For example, if "foo.jar" contains the resource "foo.MyHelloWorld.properties", then load it into the property table.
- If the previous resource wasn't found then look in the jar file for the resource "<package name>.<application name>.properties". If it is found, then load it into the existing property table. For example, if "foo.jar" contains the resource "com.myexamples.MyHelloWorld.properties", then load it into the property table.

These steps allow properties to be defined as both files within the Applications folder hierarchy as well as resources within jar files. It also allows jar file level properties to be defined, and then to be augmented or overridden by application level properties.

5.0 Resources

All Maui applications are bundled as jar files. The jar files may contain not only class files, but also resources, such as images, text, etc. All of these resources are accessible through a couple of mechanisms. Firstly, they can be retrieved through Maui's ResourceManager. Alternatively, since Maui has internal ClassLoaders, they can be retrieved through the ClassLoader's "getResourceAsStream" method.

5.1 ResourceManager (TBC)

The ResourceManager provides general access to resources within jar and zip files. It also can dynamically add resources to or remove resources from the ResourceManager's file list. These files then become generally available to the Maui runtime environment.

5.2 Accessing Resources (TBC)

Examples of how to access resources from jar file through ResourceManager.

5.3 Properties As Resources (TBC)

Examples of how to access properties from jar file through ResourceManager.

6.0 Deployment (TBC)

6.1 Bundling Maui Applications

CodeWarrior example

JDK jar utility example.

6.2 Loading, Unloading and Reloading Maui Applications

How to load, unload reload an application

What Maui does – folder scanning, class loader

6.3 Activating Maui Applications

URL's translated into references to Maui Applications. How Maui loads applications.

7.0 Other Features (TBC)

7.1 Site Specific Initialization (I_SiteInitializer)

Description

Code example

7.2 Extensions

Sharing common libraries through Extensions folder.

Appendix i – Class Loading (TBC)

How MauiClassLoaders are structured.

Appendix ii – Property Descriptions (TBC)

From maui.properties

Appendix iii – Troubleshooting (TBC)

Static variables losing their values (because of garbage collection of unreferenced Class). Addressed using global properties

Invalid package name reference – folder hierarchy for class files