Flax Developers Guide

# Pre-requisites

To develop with the Flax it is necessary to know the following technologies:

* HTML5. (Basic knowledge).
* JavaScript.

# Introduction

Flax was developed by Marcus Mascord of CSC Portugal. It is currently in the Beta version, and with time improvements and more features will be added to it.

Flax is to be used with the HTML5 canvas tag. The Canvas Framework is a JavaScript file “Canvas.js”, a stylesheet file “Canvas.css” and various XML files that are all included in the HTML page.

The HTML5 canvas, is a rectangle canvas that is added to a HTML page. The HTML5 canvas is primarily used for very graphical interactive applications such as games, graphs and animations etc.

The problem with the HTML5 canvas, is that the development time can be quite high. This is because it is just a blank canvas that the developer has to draw on. This can lead to high development time.

Using Flax, the development time needed for the HTML5 canvas is greatly reduced.

Flax is designed for mobile devices with touch screens such as tablets and smart phones. Flax will work within standard desktop browser such as FireFox, Chrome and Safari. As it is specifically designed for touch screens some of the interfaces may seem strange to a user with a mouse.

Getting Started

To use Flax, the HTML5 page such as index.html must contain the following skeleton, as shown here:

<!DOCTYPE html>

<html>

<head>

<link rel=StyleSheet type="text/css" href="css/mm.css" />

<script type="text/javascript" src="jquery-1.8.2.min.js"></script>

<script type="text/javascript" src="mm.js"></script>

<script type="text/javascript" src="mm\_ios.js"></script>

</head>

<body>

<canvas id="canvasName" width="320px" height="480px"></canvas>

<div id="main"/>

<div id="dpi"></div>

</body>

In the **<head>** tag it is necessary to include the “mm.css” stylesheet and “mm.js” and “mm\_ios.js” script. The “mm.js” script contains the bulk of the Flax API. The “mm\_ios.js” is specific to the iOS operating system, please select the correct file as shown in the following table for the OS the development is for:

|  |  |
| --- | --- |
| Operating System | JavaScript File |
|  |  |
| iOS | mm\_ios.js |
| Android | mm\_android.js |
| Blackberry Playbook and BB10 | mm\_blackberry.js |
| Firefox OS | mm\_firefox.js |
| Windows Phone 8 | mm\_wp8.js |
| Web Browser | mm\_web\_browser.js |

The specific operating system JavaScript file is small, it is possible to recreate or reuse one for an operating system that has not been listed and supports the HTML canvas tag.

It is necessary to include jQuery version 1.8.2, this can be download from:

www.jquery.com

The **<body>** tag must contain the HTML5 canvas, as shown:

<canvas id="canvasName" width="320px" height="480px"></canvas>

Give the canvas tag an id, this can be anything you like and in this example it is “canvasName”. You give the width and height of the canvas, again this can be size that you require. Normally this is the size of the mobile device you are developing for.

Also a <div> tag is required as shown here with the id “main”, the id can be called anything you like; here it is “main”:

<div id="main"/>

This must be included because it is necessary for the **Input** widget, this will be explained later.

The **<div>** tag with the id equal to “dpi”, must be included. The id must be “dpi”, this div is used by the Canvas Framework to calculate the size of the font on different devices. This must be included as shown:

<div id="dpi"></div>

# Widgets

Flax comes with a set of widgets. A widget is a reusable piece of UI.

There are two types of widgets a **normal widget** and a **container widget**.

A **container widget** is exactly the same as a **normal component** the only difference is that it can contain other widgets within it. In the documentation from now on when I refer to a **widget**, I may be referring to a **normal widget** or **container widget**.

This version of Flax contains the following widgets:

**Normal Widget:**

* Text
* Input
* Image

**Container Widgets:**

* Screen
* List
* Fragment
* Circle
* Polygon
* Animation
* Page
* PageFlow

## Screen Widget

The **Screen** **widget** needs specific explanation. The HTML5 tag has been specified as:

<canvas id="canvasName" width="320px" height="480px"></canvas>

The page will have a rectangular canvas with the width of 320, and the height of 480. When initialising Flax it is necessary to pass in the id of the canvas to be used, in this case “canvasName”.

Flax will automatically create the **Screen** widget on initialisation. This is the representation of the canvas, within Flax.

Every **widget** we add will be added to the **Screen** widget. For that reason the **Screen** widget is a **container widget** because it contains other widgets.

The **Screen widget** can therefore be considered as the **root widget** to which all other widgets belong. It has the exact same width and height that has been specified in the HTML5 canvas tag.

## Widgets

Each **widget** has the **m** attribute, this can be considered “main widget” attributes. As inheritance is not natural to JavaScript unlike Java, association has been used instead of inheritance. So instead of each component inheriting a class called **Widget,** each widget contains a link to the JavaScript “class” **Widget**.

This link is held in the attribute **m**, so association is used instead of inheritance.

So to sum it up, each **widget** contains an association to the class **Widget** via the attribute called **m**.

The class **Widget** contains common attributes that are used in the widgets. For example it contains the attributes **parent**, **l**, **x**, **y, w and h** amongst others.

## Parent Attribute

Each **widget** belongs to another **widget** except for the **Screen widget**, which is the **root widget**.

The attribute **parent** is an association to the **parent widget**. So if we add a **List widget** to the canvas, its parent would be the **Screen widget**. However if we add a **Fragment widget** to the **List widget** then its **parent** would be the **List widget**. This is because the **List widget** is a **container widget.**

Layer Attribute

The **l** attribute specifies which layer the widget is drawn on. For example this is to handle the overlapping of different widgets.

The **l** attribute is a simple integer number, and it is possible to have as many layers as there are integer numbers. This gives you more than enough layers for any application. The higher the layer number, the higher the layer it is drawn in. So a higher number layer is drawn on top of the lower layer number.

Note: A currently moving widget has the highest priority and is always drawn on top of all other layers.

So let us imagine that we want to draw three rectangles on top of each other, as shown in figure 1.



Figure 1: Layer Example

The pink rectangle was assigned the **l =** 1, the yellow **l =** 2 and the blue **l =** 3. The higher layers are drawn on top of the lower layers.

It is not necessary to have incremental layers by one position, for example it may have been an advantage to set this as **l** 10, 20 and 30 instead of 1, 2 and 3. This is an advantage because lets imagine later on you want to add a **widget** between the pink and the yellow rectangle. Using the system 1, 2 and 3 you would have to change some layer numbers.

However in the system 10, 20 and 30, it is possible just to add the **widget** to a new **layer** 11 for example. None of the other code will have to be modified.

Also let us imagine we have used the system 1, 2 and 3, and we realise we need a **widget** under the **layer** 1. This is easy; it is possible to use negative numbers, so for example we could add this **widget** to the **layer** -1.

This demonstrates how flexible the layer system is.

The layers work according to the **parent widget**. For example if a **Fragment widget** contains a **Button** and **Text widget**, the **layer** of the **Button** and **Text** widget is relative to the **Fragment** **widget** and not the **Screen** **widget**.

So let us imagine that we have 3 **Fragment** widgets with layers 1, 2 and 3. Fragment 2 of **layer** 2 has the two widgets **Button** and **Text**. If the **Button** has been set to **layer** 10 and the **Text** to **layer** 20 then the widgets will be drawn in this order:

* Draw fragment 1 which is layer 1.
* Draw fragment 2 which is layer 2.
  + Draw Button which is layer 10 with the parent fragment 2.
  + Draw Text which is layer 20 with the parent fragment 2.
* Draw box 3 which is layer 3.

As is clearly shown, the layers are relevant to the **parent widget**. And the fragment 3 would be drawn above the **Button** and **Text widget** that belong to fragment 2.

Note: Many widgets can be added to the same layer. However if they overlap there is no way to control how they overlap if they belong to the same layer. So make sure these do not overlap otherwise undesirable affects could be introduced.

## X and Y Attributes

The **x** and **y** attributes specify the position of the **widget**, compared to its **parent widget** as specified in the **parent** attribute. The **x** and **y** refer to the top left of the **parent** component as 0 and 0. So **x** refers to the horizontal axis or going from left to right and **y** refers to the vertical axis or going from top to bottom as shown in figure 2 for the **Screen widget** of 1024 by 600.

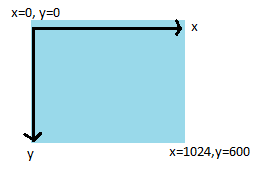


Figure 2

When adding a **widget** to the **Screen** or **composite widget** it is possible to specify the **x** and **y** position in pixels or percentages. Each **widget** has a specific add function that is used to create the **widget**. As part of the add function we pass in the **x** and **y** position. It is only possible to specify the percentage in the add function. Once the **widget** has been created we can only change the **x** and **y** attribute by its pixel position.

Let us imagine in the add function we set that **x** = “10%” and **y** = “10%”. This is relative to its parent. In this example if this widget is added to the Screen component, then the x position is relative to the screen width 1024 pixels. So in this example the “10%” will be converted to “10%” of 1024 which gives an x pixel position rounded to 102. Equally “10%” of the height of the screen is 60 as the screen height is 600.

This is relative to width and height of the **parent widget**.

On the add function it is possible to specify the x and y as pixels, this is done as so x = “50”, y = “100”.

The **x** and **y** attributes, once the widget has been added to the parent widget will be integer values. The **x** and **y** position can be manipulated after this, but only by specifying the **x** and **y** pixel position as an integer value.

Some **container** widgets will ignore the **x** and **y** that has been specified to the widgets belonging to them when they were created.

## Width and Height Attributes

The **w** and **h** attributes contain the width and the height of the widget.

Like the **x** and **y** attributes in the add function these can be specified as pixels or percentages. The **w** and **h** attributes are calculated according to the **w** and **h** attributes of their parent.

However once the **w** and **h** attributes have been added they become integers that represent the pixel value.

So let us imagine for a screen with a **width** of 1024, if we set the **width** of the widget to “10%” this will be converted to the rounded value of 102 pixels. Also if the **height** of the screen is 600 and the **height** attribute is set to “10%” then this will be converted to the rounded value of 60 pixels.

It is possible to set the **height** and **width** values in the add function of the widget in pixels as for example **h** = “10”, **w** = “10”.

## Alignment

Instead of using x and y attributes to position the widgets, it is possible to position the widgets using the align attributes. The two align attributes are alignHoz and alignVert.

AlignHoz, aligns the widget horizontally within its parent it can take three values these are LEFT, RIGHT and CENTER. The LEFT value will align the left edge of this widget to the left edge of its parent. The RIGHT value will align the right edge of this widget to the right edge of its parent. The CENTER value aligns the widget to the center of its parent. The examples below show how a blue fragment is positioned within its parent, which is the black fragment:

AlignHoz=LEFT



AlignHoz=CENTER



AlignHoz=RIGHT



The blue fragment has been aligned to the very edge of the black fragment in the left and right example. It may be necessary to align the blue fragment to the left or right but with some spacing. This is possible with the AlignSpacingHoz attribute, this attribute is set with pixel value of the spacing. Two examples are shown here:

AlignHoz=LEFT, AlignSpacingHoz=20



AlignHoz=RIGHT, AlignSpacingHoz=20



AlignVert, aligns the widget vertically within its parent it can take three values these are TOP, BOTTOM and CENTER. The TOP value will align the top edge of this widget to the top edge of its parent. The BOTTOM value will align the bottom edge of this widget to the bottom edge of its parent. The CENTER value aligns the widget to the center of its parent. The examples below show how a blue fragment is positioned within its parent, which is the black fragment:

AlignVert=TOP



AlignVert=BOTTOM



AlignVert=CENTER



The blue fragment has been aligned to the very edge of the black fragment in the

top and bottom example. It may be necessary to align the blue fragment to the top or bottom but with some spacing. This is possible with the AlignSpacingVert attribute, this attribute is set with pixel value of the spacing. Two examples are shown here:

AlignVert=TOP, AlignSpacingVert =10



AlignVert=BOTTOM, AlignSpacingVert=10



Alignment Notes:

* If the widget has been set with an x value and an alignHoz value, the alignHoz value will always override the x value. The x value will be ignored.
* If the widget has been set with an y value and an alignVert value, the alignVert value will always override the y value. The y value will be ignored.
* It is possible to mix x and y attributes with align attributes. For example we could set x and alignVert, or we could set y and alignHoz.
* Some parent widgets will ignore the alignment values that have been set in the child widget. Please see the detail for each parent widget.
* Alignment should be used where responsive design is required, it makes it easier to fit to different screen sizes. Using fragments within fragments and alignments, the correct layout can be achieved.

## MVC – Layout Overview

It is possible to code all of the application using just JavaScript code with the Flax API, however this is not recommended.

Flax allows for all UI widgets to be configured using XML. This leads to good separation of the view from the controller code. It is easy to change the appearance of the UI in XML without changing the underlying JavaScript code.

An example of this would be a button widget. It is possible to define the colour, text, position and size of the button widget in XML. The action when the user presses the button is a function within JavaScript code. Therefore it is possible for the developer to change the colour, text, position and size of the button without changing the underlying code.

A Flax application is build using a number of different XML files, including:

|  |  |
| --- | --- |
| **XML File** | **Description** |
|  |  |
| **common.xml** | Contains information about common widgets that will be reused throughout the application. |
| **pageflow.xml** | This indicates which pages are contained within this page flow. More than one page flow can exist in this file. |
| **page.xml** | A page.xml file is necessary for each page within the application; it contains each widget within that page. A different page file with a different name will be necessary for each page in the application. |
| **messages.xml** | This contains the text messages for the application so that it can easily be internationalized. One of these files should be available for each language the application uses, for example English -> messages\_en.xml and French -> messages\_fr.xml. |

## Common.xml (Overview)

The common.xml file contains information about common widgets that will be reused throughout the application.

A good example might be a blue button. To keep the look and feel of the application consistent from page to page, the blue button will be reused across different pages. Instead of constantly redefining the blue button on each page where it will be used, it can simple be defined once in the common.xml file.

A common.xml file example defining the blue button is:

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

<CommonWidgets>

<WidgetSection>

<Id>Main</Id>

<PreLoad>True</PreLoad>

<SectionWidgets>

<Widget>

<Id>BlueButton</Id>

<Class>Fragment</Class>

<X>10%</X>

<AlignVert>TOP</AlignVert>

<H>50</H>

<W>80%</W>

<L>1</L>

<Style>

<Colour>#0066FF</Colour>

</Style>

</Widget>

</SectionWidgets>

</WidgetSection>

</CommonWidgets>

In the XML example above, we will ignore some of the tags that will be explained later in this documentation. Take a look at the elements between the <Widget> and </Widget> tag, that defines the blue button.

The first tag that is important is the <Id> tag, this must be different for each widget defined within the common.xml file and it must be defined. This is being set to “BlueButton” for this example, in fact what we are doing is defining a new class called “BlueButton” that will be used throughout our application.

The <Class> defines which class this widget belongs to, and in this example it is the “Fragment” class. We could say “BlueButton” is a new class widget that inherits from the “Fragment” widget.

For this example the other tags are less important but they define the position, size and colour of the blue button.

The common.xml defines the attributes of this widget but it does not actually place it onto the screen within a page.

In any of our page xml files, if we want to reuse this new widget “BlueButton” we simple need to add a reference to it in the page xml file, as:

<Widget>

<Class>BlueButton</Class>

</Widget>

In this example, we only have to indicate what the class is for the widget.

The blue button will then be placed inside our page and shown to the user. The position and size and colour of this button are defined in our common.xml file.

It might be necessary to keep most of the characteristics of the blue button the same, but change its position just for this page. This is possible, because any attribute can be overridden at page level.

For example to change its vertical alignment from top to bottom we can define our button as this in our page.xml file:

<Widget>

<Class>BlueButton</Class>

<AlignVert>BOTTOM</AlignVert>

</Widget>

The advantage of using the common.xml file and not defining the “BlueButton” widget separately in our page file is that:

* It cuts down on XML code, reuse.
* If we want to change an attribute, such as the colour of the blue button is that it will be changed across the whole application.

It is also possible to reuse the blue button in the JavaScript code. To add the blue button to a page in JavaScript code, a code snippet is shown:

// Obtain the Widget Class “BlueButton”

var blueButtonClass = mm.App.getWidgetClass(“BlueButton”);

// Instantiate a new “BlueButton” instance

var newBlueButton = mm.App.addWidgetX(null, blueButtonClass);

// Add the new blue button to the page

mm.App.add(page, newBlueButton);

As a rule of thumb, the UI elements should be defined in the XML files and as the examples here show it is easier. However there may be times when it is necessary to use JavaScript code in which case it is possible.

In the blue button example we just created one simple widget within the common.xml file. It is also possible to create very complex widgets within the common.xml that contain widgets within widgets. An example of this is that our application might require a header that will be reused across all the pages. This header contains a back button and header text.

In this example we can create our new “Header” class widget, this widget will be of the class Fragment. Within this widget class we can define a back button as an image widget and a text widget, these will be placed inside the <Widgets> tag so they will be child widgets of our “Header”.

Ignoring the standard tags within the common.xml file, our header can be defined as:

<Widget>

<Id>Header</Id>

<Class>Fragment</Class>

<X>0</X>

<AlignVert>TOP</AlignVert>

<H>50</H>

<W>100%</W>

<L>1</L>

<Style>

<Colour>#FFFFFF</Colour>

</Style>

<Widgets>

<Widget>

<Id>BackButton</Id>

<Class>Image</Class>

<H>32</H>

<W>32</W>

<L>1</L>

<AlignHoz>LEFT</AlignHoz>

<AlignVert>CENTER</AlignVert>

<Src>images/back\_button\_img.png</Src>

<Click>

<ClosePage>True</ClosePage>

</Click>

</Widget>

<Widget>

<Id>HeaderText</Id>

<Class>Text</Class>

<AlignHoz>CENTER</AlignHoz>

<AlignVert>CENTER</AlignVert>

<Style>

<Colour>#000000</Colour>

</Style>

<Font>Arial</Font>

<FontSize>18</FontSize>

<Text>Header</Text>

</Widget>

</Widgets>

</Widget>

From this example a new widget class has been created which is called “Header”, inside the “Header” we have a back button called “BackButton” that when pressed will return the user to the previous page. The “Header” contains header text called “HeaderText”, that will contain the name of the page. I will not go into detail here about each tag, however if you are interested please check out the details for the “Fragment”, “Image” and “Text” widgets.

It is now possible to use this new widget class “Header” within our page xml file. In this page.xml file it is necessary to set the text value of the “HeaderText” to the name of the page, this will change for each page. This example shows how that is possible, this page will have the header “Start Page”.

<Widget>

<Class>Header</Class>

<Widgets>

<Widget>

<Id>HeaderText</Id>

<Text>Start Page</Text>

</Widget>

</Widgets>

</Widget>

We want to override the “text” attribute of the “HeaderText” widget, however we do not want to create a new widget within the “Header” widget. To override the “HeaderText” we do not specify the “Class” as “HeaderText”, we must specify the “Id” so that the canvas framework knows we are referring to this widget.

For example for another page, let us call it “Contact Page” it may be necessary to add another widget to the header. This widget is an information button, which the user presses to get information. The header widget with this new information button can be added to the page as:

<Widget>

<Class>Header</Class>

<Widgets>

<Widget>

<Id>HeaderText</Id>

<Text>Contact Page</Text>

</Widget>

<Widget>

<Class>Image</Class>

<H>32</H>

<W>32</W>

<L>1</L>

<AlignHoz>Right</AlignHoz>

<AlignVert>CENTER</AlignVert>

<Src>images/info\_button\_img.png</Src>

<Click>

<Navigation>InfoPage</Navigation>

</Click>

</Widget>

</Widgets>

</Widget>

As is shown in this example, we have used the id for “HeaderText” so we are referring to the one that has already been defined in the “Header” widget in the common.xml. We are overriding it and setting its text to “Contact Page”. The logic works like this:

* If the id specified can be found above in the same parent that is specified in the common.xml file than it is overridden.
* If the id specified couldn’t be found above in the same parent or if an id has not been set a new widget is created and added to its parent widget.

The widget that has been added is the new information button that is of the class “Image”.

## Page and PageFlow (Overview)

A typical application is build up from a number of pages. It is normal that one page will call another page etc and that is handled by the page flow.

The PageFlow widget is a container widget that contains all of the pages that are necessary for the page flow. It handles the page transitions from one page to the other etc.

An application typically only has one page flow, but it is possible to have more than one page flow where necessary. For example on a tablet application it might be decided to split the screen into two separate sections: one section being one page flow, and the other section being another page flow.

It is not necessary for a PageFlow to completely fill the screen. So the page flow can be considered a section of a screen like a fragment. The difference being that the pageflow contains pages.

Page flows are important to Flax they get they have their own XML file.

Let us have a look at a pageflow.xml file example for an application with 2 pages. Page1 and Page2, the pageflow.xml file is:

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

<PageFlows>

<PageFlow>

<PreLoad>True</PreLoad>

<Id>MainPageFlow</Id>

<Class>PageFlow</Class>

<X>0</X>

<Y>0</Y>

<W>100%</W>

<H>100%</H>

<StartPage>Page1</StartPage>

<Widgets>

<Widget>

<Include>page1.xml</Include>

</Widget>

<Widget>

<Include>page2.xml</Include>

</Widget>

</Widgets>

<TransformationName>PAGE\_FLOW\_RIGHT</TransformationName>

</PageFlow>

</PageFlows>

As can be seen the pageflow is defined in the <PageFlow> tag, and it is possible to define more than one pageflow in this file.

The <PreLoad> tag specifies if the pageflow is loaded when the application is first opened, or if it is only loaded when required.

The <Id> tag specifies the id of this pageflow, and the class tag specifies the “PageFlow” class. This may be a different pageflow class that was defined in the common.xml file.

The <X>, <Y>, <W> and <H> tags specify its position and size relative to its parent the screen widget.

The <StartPage> tag contains the <Id> of the page that will be the first page of this pageflow. It will automatically open to this page. This page must be specified within the <Widgets> tag.

The <Widgets> tag specifies all of the pages that are part of this pageflow. Each page widget is defined in the <Widget> tag. It is possible to define all of the page tags within the pageflow.xml, however this would lead to a large and difficult to read XML file. Page reuse across different pageflows would not be possible. To solve this problem, the <Include> tag allows for each page.xml file to be inserted.

The <TransformationName> tag specifies which transformation will be used when one page navigates to another page.

In the above example we have a page1.xml and a page2.xml file. The page1.xml contains a button, that when pressed will open page 2.

The page1.xml file example is shown:

<Widget>

<Id>Page1</Id>

<Class>Page</Class>

<PreLoad>True</PreLoad>

<X>0</X>

<Y>0</Y>

<W>100%</W>

<H>100%</H>

<Widgets>

<Widget>

<Class>BlueButton</Class>

<Click>

<Navigation>Page2</Navigation>

</Click>

</Widget>

</Widgets>

</Widget>

The page1.xml uses the BlueButton that was defined in 1.5.6 Common.xml file section.

In the example above the <Id> tag has been set to Page1 the <Class> tag to Page and the position and size of the page has been set.

The BlueButton widget has been added to the page, with the click action. This click action then navigates to Page2, when a user presses it.

Page2 will contain another BlueButton that closes the page, and returns the view to the previous page. The page2.xml XML code is:

<Widget>

<Id>Page2</Id>

<Class>Page</Class>

<PreLoad>True</PreLoad>

<X>0</X>

<Y>0</Y>

<W>100%</W>

<H>100%</H>

<Widgets>

<Widget>

<Class>BlueButton</Class>

<Click>

<ClosePage>True</ClosePage>

</Click>

</Widget>

</Widgets>

</Widget>