

Flex Whiteboard Component

<http://flashcomguru.com>



Table of Contents

Introduction	3
Setting up the whiteboard	3
Server-side (Flash Media Server)	3
Client-side (Flex Builder)	3
Canvas drawing Area	5
Menu View Stack	5
Using The Public APIs.....	5
disconnect().....	6
drawMode	6
enableCreation	6
enableSelection.....	6
API Wrappers / Controllers	6
▪ PageController	6
▪ ToolController.....	6
▪ ShapeController	7
▪ KeyController	7
▪ ContextMenuController	7
How To's.....	7
Customize the Whiteboards UI	7
Removing a field from a menu bar	7
Fading in the drawing canvas when the page loads.....	8
Make the canvas scroll, scale or be fixed size.....	9
Adding an image to the whiteboard	9
Giving Pages a background color / any property	9
Changing the stamps shapes.....	10
Multiple unique whiteboards.....	11
Adding custom keyboard shortcuts.....	11
Notes.....	12
ShapeRef object	12

Introduction

The Whiteboard Component is written for use with Flex applications and uses Flash Media Server as a backend for data and synchronization. The whiteboard comes in the form of a container `Whiteboard.WhiteboardManager`. This class is a canvas and all the whiteboards UI elements can be put into it or around it, this gives you control over all non shape UI such as the menu bars and toolbars etc for a fully customized experience.

As well as UI customizability there are also many public APIs which give full control over the whiteboard functionality, as low level as creating shapes with just method calls. The public APIs are split into several 'controllers' which host relevant methods, properties and events.

Setting up the whiteboard

Server-side (Flash Media Server)

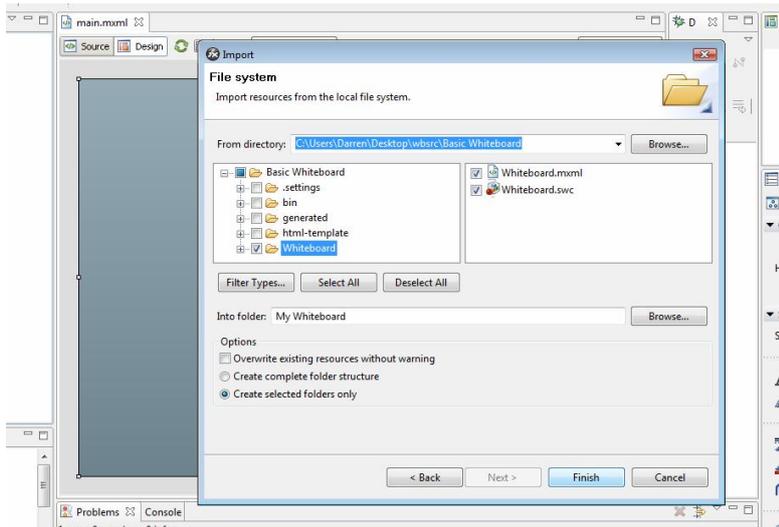
(If you are new to flash media server please read http://flashcomguru.com/articles/fms2_basics.cfm and the various other getting started articles on flashcomguru)

To setup your serverside application simply take the `Whiteboard.asc` and copy it into your application directory and then add the line `load("Whiteboard.asc");` to your own serverside code.

To get started quickly with the samples copy the `flex_whiteboard` folder from the `/serverside/` directory into your localhost applications folder (typically `c:\program files\Macromedia\Flex Media Server 2\applications\`).

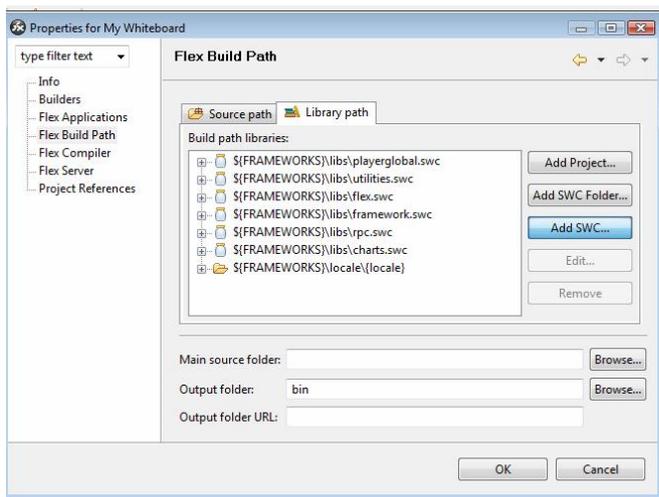
Client-side (Flex Builder)

In order to use the whiteboard in your project the file `Whiteboard.SWC` must be included in your project as well as the resource files (menus and icons). To do this find a version of the whiteboard from the samples you wish to use such as the Basic Whiteboard and browse into it. In each sample there is a folder called 'Whiteboard', this entire folder must be brought into your project. You can do this by either importing the folder in eclipse by right clicking on your application -> import -> file system and then browsing to the folder.



Or you can copy the Whiteboard folder from the samples directory to your projects directory usually found at 'my documents\Flex Builder 2\{project name}\' and then refresh your project (right click -> refresh).

Once you have the Whiteboard folder you need to add the Whiteboard.SWC to your build path. To do this right click on your project and go to properties. You will see a window similar to the one below.



Navigate to the Flex Build Path item in the tree and you will see a button 'Add SWC', click this button and another window will appear to browse for the component. Type in 'Whiteboard/Whiteboard.swc' and press ok and ok again on the properties dialog.

Now the component is included you can add the Whiteboard component to your application by dragging and dropping it from the Components panel under Custom onto your application.



You finally need to connect the whiteboard to your serverside. This should be done after the whiteboards creationComplete event has been called, so in the parent creationComplete event or

the applicationComplete event etc. To connect it simply create your net connection and call either whiteboard.connect or whiteboard.connectAs.

```
nc = new NetConnection();
nc.connect("rtmp://flex_whiteboard");
whiteboard.connectAs(nc, "demo1");
```

The connect(nc) method is the same as calling .connectAs(nc , this.name); The second argument of connectAs is the instance name. If you wanted to have several whiteboards in the same application you could use 1 instance of the whiteboard UI and just call connectAs with different IDs. The shared objects on the serverside which store the data will be named with:

wb.{instance ID}.{soName e.g. pages|p1|cache}.

You can now run the application.

Notes: if you are not going to use a sample Whiteboard.mxml you must make sure to give the whiteboard manager 2 things. A canvas to draw on and a viewstack with the sub menus.

Canvas drawing Area

The canvas can be any canvas anywhere with any settings and you register it by calling whiteboardManager.registerDrawingArea (my_canvas); The whiteboard will not work without a canvas.

Menu View Stack

You should also give the whiteboard a menu stack if you intend for users to be able to create shapes or edit them. Basically you must have a ViewStack which contains a number of HBox's, each HBox should have its name property set to the tools name, such as 'Select' or 'Rectangle'. Inside the HBoxes of tools you want to have menus there should be a UIComponent which extends Whiteboard.tools.menuBars.WhiteboardMenuBar. This should be called 'menu'. You can simply copy and paste the existing view stack from one of the samples they are found in Whiteboard/Whiteboard.mxml in most samples along with the default canvas.

Once you have your viewstack you should register it via whiteboardManager.registerMenuStack(my_view_stack);

Using The Public APIs

The whiteboard.mxml component extends from the class WhiteboardManager which can be found in the asdocs. The whiteboard contains 3 properties and 3 methods, the rest of the methods and properties are the controllers which own other areas of functionality.

disconnect()

The connect and connectAs methods were explained above, the disconnect methods should be used when the whiteboard is no longer in use, or before connecting it to another net connection or instance, when your net connection closes the whiteboard will call this method itself automatically.

drawMode

The property draw mode controls how the whiteboard should handle the boundaries of the canvas, it can either scroll (`WhiteboardManager.DRAW_MODE_SCROLLING`), scale to fit the content (`WhiteboardManager.DRAW_MODE_SCALE`) or be fixed size (`WhiteboardManager.DRAW_MODE_FIXED`) not allowing shapes to exceed the boundaries.

enableCreation

This Boolean allows or disallows the user to create new shapes

enableSelection

This Boolean allows or disallows the user to select and modify shapes

API Wrappers / Controllers

The rest of the public APIs are split into controllers. The controllers are available from the `WhiteboardManager` class which is what the `Whiteboard.mxml` extends from. For each controller there are two ways to acquire it, as a property such as `'my_wb.pageController'` or as a method `'my_wb.getPageController()'`. The reason for both is because flex cant bind to a property of a property but most of the properties in the controllers are bindable, such as `'getPageController().selectedPage'` so for binding the method must be used.

An example usage might be to paste all the items in the clipboard where they were + 15 pixels, this would be done as such: `'my_wb.shapeController.pasteAtOffset(15, 15);'` Simply acquire the controller and call the appropriate method.

- **PageController** (`whiteboardManager.pageController || whiteboardManager.getPageController()`)
 - The page controller has properties for getting and setting the current page and events relevant to the page changing. It also has functionality for creating and clearing pages, along with APIs for storing settings against a page so you can create scenarios that are page specific, such as a page having a background image.
- **ToolController** (`whiteboardManager.toolController || whiteboardManager.getToolController()`)
 - The tool controller allows you to set the active tool, such as 'Rectangle' or 'Circle' etc and a `toolChange` event. There is also a helper method `registerButton` which will allow you to associate any `UIElement` with a selected property with a tool so when a tool is selected the appropriate button is. However if you did not want to

use buttons you can just handle that manually listening to the toolChange event.

- **ShapeController** (`whiteboardManager.shapeController || whiteboardManager.getShapeController()`)
 - The shape controller exposes a large number of methods for manually modifying shapes on the whiteboard. It has methods for getting/setting selected shapes, adjusting shapes, deleting shapes etc. It also has clipboard methods to support cut/copy/paste functionality. For both of these there are relevant events such as selectionChange, newShape, clipboardChange etc.
- **KeyController** (`whiteboardManager.keyController || whiteboardManager.getKeyController()`)
 - This class allows you to easily create keyboard shortcuts for the whiteboard by registering a key on its own, with shift or control to trigger a callback.
- **ContextMenuController** (`whiteboardManager.contextMenuController || whiteboardManager.getContextMenuController()`)
 - This class offers a ready built context menu for the whiteboard which you can add remove items from or manually add items to the menu object. It supports the clipboard functions, bring to front, and shape transforming.

How To's

Customize the Whiteboards UI

The entire user interface for the whiteboard can be found in `Whiteboard/Whiteboard.mxml`, double click on this file and you will be presented with a regular mxml component which contains several buttons etc. You can remove these buttons or style them however you like, the whiteboard is simply a manager and a container.

There is a canvas in `Whiteboard.mxml`, this is where shapes are drawn, you can edit this in any way you like, such as adding a background image or background color.

To edit the menu bar UI you need to edit the menu bar components found in `Whiteboard/tools/menuBars/`. Each `_menu{...}.mxml` is fully editable if you wanted to skin, theme or completely rip out and change the menu bar.

Removing a field from a menu bar

If you wanted to have a field for a shape static, such as only have 1 fixed font for the text tool this can be done fairly easily.

First find the appropriate editor for the tool, in the `Whiteboard.mxml` you will see a `ViewStack` with `HBox`'s named as the tool and inside them will be an instance of the correct menu bar.

For the most part you can't remove a property all together because the tool has registered it as 'required for draw' but you can remove it from the user. For a list of required fields for each tool see the tool info at the bottom of this document.

For the example of only having 1 font in the text tool, open the `_menuText.mxml` from `Whiteboard/tools/menuBars` and delete the font combo box.

Whiteboard menu bars implement 2 methods `getValue` and `setValue`. `getValue` is called when the whiteboard needs the current value from the menu, such as for a new shape been created, it will call `getValue` with the property name, you must then return what the value is. In our case we can replace:

```
else if(valueID == "font")
{
return fontCombo.value;
```

with return "Verdana" or whichever fixed font we wish to use.

The `setValue` method is called when a shape is selected so the menu bar can update itself to have all the values of the selected shape. We can remove entirely the statement for setting the font.

Menu bars can also inform the whiteboard about a change by calling `updateValue("valueID", value)`; you can remove all instances of `updateValue("font"` also.

Fading in the drawing canvas when the page loads

This is a simple effect to make the page loading less harsh and instant. Add a fade tag to your `Whiteboard.mxml` and configure it however you want:

```
<mx:Fade id="fadeEffect" duration="500" alphaFrom="0.0" alphaTo="1.0"></mx:Fade>
```

Then in the `drawingArea` canvas set the `showEffect` to your effect:

```
showEffect="fadeEffect"
```

Then in the `init` function register 2 events which you will use to make the canvas visible and invisible:

```
pageController.addEventListener(PageEvent.PAGE_CHANGE, function() { drawingArea.visible = false; });
```

```
pageController.addEventListener(PageEvent.PAGE_CHANGED, function() { drawingArea.visible = true; });
```

`PAGE_CHANGE` is called as a page is about to connect to gets its data and `PAGE_CHANGED` is called when the first data is in which is after the shapes on the page have been drawn.

Make the canvas scroll, scale or be fixed size

To best fit your application 3 draw modes are included:

- Scrolling – shapes are allowed to go off the canvas and this will cause the canvas to show its scroll bars and let users scroll to see all of the shapes.
- Scale to fit – shapes are allowed to go off the canvas again but when they do the whole page is scaled so it fits into the size of the drawingArea canvas.
- Fixed size – shapes will never be allowed outside of the edges of the canvas.

To set the drawMode simply assign the value on your whiteboard or whiteboardManager to one of the constants on whiteboardManager, e.g.

```
my_whiteboard.drawMode = WhiteboardManager.DRAW_MODE_SCALE;
```

```
my_whiteboard.drawMode = WhiteboardManager.DRAW_MODE_SCROLLING;
```

```
my_whiteboard.drawMode = WhiteboardManager.DRAW_MODE_FIXED;
```

Adding an image to the whiteboard

See samples/Images in Whiteboard

Images have no built in UI because they are the kind of thing which requires customization, a custom upload system etc. Once you have that system in place it is simple to add an image. Simply take the shapeController and call createShape as such:

```
my_whiteboard.shapeController.createShape( "Image", {x:50, y:50, width:100, height:100, src:"foo.jpg", lineSize:1, lineColor:-1} );
```

And an image will be added with all those settings. The lineColor, like fillColor and lineColor on other shapes can be -1, -1 means no line or no fill.

The image you use must be on the same site as the whiteboard or you must have loaded a policy file because flash's security won't allow cross domain image loading.

Giving Pages a background color / any property

See samples/Page Settings Whiteboard

The flex whiteboard has APIs that allow you to store data against a particular page, this could be used for any page relevant settings such as background image, page name etc. These methods can be found in the page controller, a simple example would be:

```
whiteboard.pageController.setCurrentPageSetting("bgcolor", 0xCCCCCC);
```

You can set any property you like (it must be in the standard format of a property, aka just alpha numeric). The above code sets a value called bgcolor for the current page as 0xCCCCCC. When that method is called the event below is called:

```
whiteboard.pageController.addEventListener(PageEvent.PAGE_SETTINGS
```

This event is called when settings for the current page are changed and on first connecting to a page with settings. If there are no settings there will be no event so it's important to clean up what you've done in PageEvent.PAGE_CHANGE which is called just before the page changes.

For a background color setting you could drag in a color picker and when it changes call:

```
whiteboard.drawingArea.setStyle("backgroundColor", 0xFFFFFFFF);
```

You would subscribe to the PageEvent.PAGE_SETTINGS event and handle it like this:

```
private function onGetPageSettings(event:PageEvent):void
{
    var bgcolor:Number = whiteboard.pageController.getCurrentPageSetting("bgcolor") as
Number;
    if(bgcolor != NaN)
    {
        whiteboard.drawingArea.setStyle("backgroundColor", bgcolor);
    }
}
```

if the color is there set the drawingArea's background color. In the PAGE_CHANGE event you would clear the background color incase the next page has no settings:

```
whiteboard.drawingArea.setStyle("backgroundColor", 0xFFFFFFFF);
```

It is also possible to send settings when creating a new page, e.g.

```
whiteboard.pageController.createPage( {bgcolor:0xCCCCCC, pageName:"my page"} );
```

You could show a dialog when a user presses the > arrow to create a new page and ask for a page name and display that name instead of "Page x of y".

Changing the stamps shapes

To change/add or remove a shape from the stamp tool you need to modify Whiteboard/tools/menuBars/_menuStamp.mxml.

You will see in the header import statements, such as:

```
[Embed(source='../icons/stamps.swf', symbol='stamp_diamond')]
[Bindable]
public var stamp3:Class;
```

You need to add/change or remove a statement like this, you can change the symbols in stamps.swf by editing stamps fla in Whiteboard/icons.

You also need to update the stamp array to include or remove the item you modified.

```
private var _stamps:Array = [stamp1, stamp2, stamp3, stamp4, stamp5, stamp6];
```

Finally you should add or remove your stamp item from the ViewStack at the bottom of the class `id="vs"`.

```
<mx:VBox tooltip="Copyright" icon="@Embed(source='../icons/stamps.swf',  
symbol='stamp_c')"/>
```

Its index in the ViewStack must correspond to the index of the stamp in the `_stamps` array.

Multiple unique whiteboards

see `samples/WhiteboardMainDemo` and its "Room" setting

FMS components tend to take an instance/prefix approach to instancing whereby it ID's your components instance with `[componentID].[instanceName].[resourceName]`, the whiteboard does this.

Traditionally the `instanceName` is the name property of the components UI and that's true again for the whiteboard if you use the `.connect(nc)` method, however there is also a `whiteboard.connectAs(nc, instanceName);` method.

By setting the instance name you can have multiple whiteboards with their own pages and history.

Adding custom keyboard shortcuts

Keyboard shortcuts are controlled by the `keyController`, a keyboard shortcut you might like to add could be when the ESC key is pressed any selected shape deselected.

This would be done as such:

```
whiteboard.keyController.registerShortcut(Keyboard.ESCAPE,  
whiteboard.shapeController.deselectShapes, true);
```

We call `registerShortcut` because we just want the ESC key to trigger it, not ESC + shift or ESC + ctrl. The first argument is the keycode, this is not a charcode, if you want a character or number to be the keycode use the `KeyController.charToKeycode` method. The second argument is the callback method, this can be your own method or just one already there such as above. The last argument states whether the whiteboard must be focused to trigger the event, for example, if we setup C to trigger copy and we set the last argument to false when a user was typing in a text field and hit the C it would copy shapes, whereas if the argument was true it would only copy when the whiteboard has focus.

You can force focus on the whiteboard by calling `whiteboard.forceFocus();`

Notes

ShapeRef object

The ShapeRef object is used to reference a shape on the stage, it should never be instantiated, you can get the correct one from the shapeController such as from allShapes, selectedShapes or in a shape event.

It contains a property called 'sprite', this is the actual shape MovieClip, there is also a property called 'propertyBag', this is a helper class which holds all the data for a shape. You can update properties on a shape by editing it in the propertyBag and then calling save, e.g.

```
(shapeController.selectedShapes[o] as ShapeRef).propertyBag.setProperty("width", 200);  
(shapeController.selectedShapes[o] as ShapeRef).save();
```

and the change will be sent to everybody.

Shape properties & if required

If you are manipulating a shape manually such as with shapeController.createShape(or shapeController.setShapeProperty you need to know what properties a shape supports, if a property is required you must include it when you create a shape via the createShape method. All colors are numbers and can be -1 for no fill/no line.

Circle		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
lineSize	True	
lineColor	True	Can be -1 for no line
fillColor	True	Can be -1 for no fill

Freehand		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
lineSize	True	

color	True	Can be -1 for no line
points	True	Array of objects with {x: , y: } for each point, first point is the .moveTo

Image		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
lineSize	True	
lineColor	True	Can be -1 for no line
src	True	String URL for the images location

Line		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
lineSize	True	
color	True	Can be -1 for no line
type	True	0 = straight line 1 = 1 arrow 2 = 2 arrows

Polygon		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
lineSize	True	
lineColor	True	Can be -1 for no line

fillColor	True	Can be -1 for no fill
points	True	Number of corners, should be more than or equal to 5

Rectangle		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
lineSize	True	
lineColor	True	Can be -1 for no line
fillColor	True	Can be -1 for no fill

Stamp		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
fillColor	True	Can be -1 for no fill
type	True	Which image index, this corresponds to the <code>_stamps</code> array in <code>_menuStamp.mxml</code>

Text		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
color	True	(font color)
text	True	The text message
font	True	String of the font face name

fontSize	True	Normal font size numbers, 8+
bold	False	Boolean
italic	False	Boolean
underline	False	Boolean

Triangle		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
lineSize	True	
lineColor	True	Can be -1 for no line
fillColor	True	Can be -1 for no fill
type	True	0 = Equilateral triangle 1 = Right angle triangle

Wedge		
Property	Required	Notes if applicable
x	True	
y	True	
width	True	
height	True	
lineSize	True	
lineColor	True	Can be -1 for no line
fillColor	True	Can be -1 for no fill
angle	True	How much of the shape is complete, 0-360