

MATE vs. SWIZ

2nd Generation Flex Frameworks



Darron Schall

Principal Architect, Universal Mind

twitter: @darronschall

Outline

- Mate Resources
- Swiz Resources
- Same Architecture
- Key Differences
- Which is better?

Learn Mate

- Watch Laura: <http://tv.adobe.com/watch/360flex-conference/mate-flex-framework-by-laura-arguello/>
- Mate and Modules: <http://www.slideshare.net/ghedwards/flex-modular-development-and-mate>
- Official Documentation: <http://mate.asfusion.com/page/documentation>

Example Mate Apps

- The Official Mate Examples
- More Examples from ASFusion
- Brian Rinaldi's MateExample
- Mate and the Presentation Model

Learn Swiz

- Watch Chris: <http://tv.adobe.com/watch/360flex-conference/introduction-to-the-swiz-framework-for-flex-by-chris-scott/>
- Official Documentation: <http://swizframework.org/docs/>
- Older Documentation: <http://code.google.com/p/swizframework/wiki/GettingStarted>

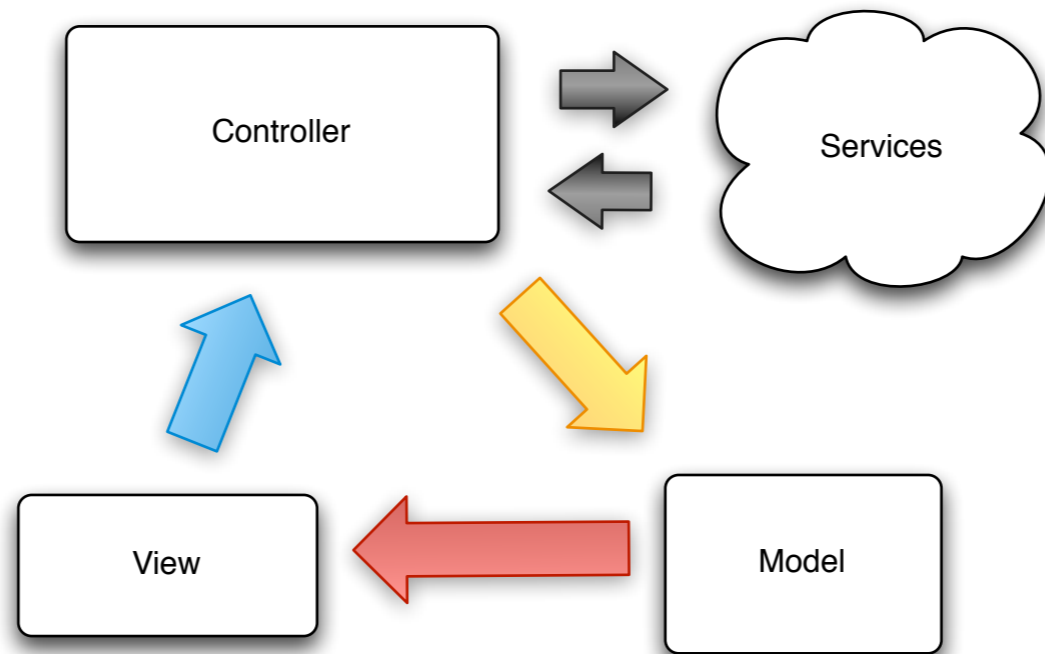
Swiz Example Apps

- Ben Clinkinbeard's [Example with Presentation Model](#)
- Christophe Coenraets: <http://coenraets.org/blog/2009/02/sample-application-using-the-swiz-framework-and-blazeds/>
- Sonke's Swiz Category: <http://soenkerohde.com/category/swiz/>

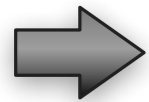
Same MVC

- Views dispatch events and bind to model properties
- Controllers communicate with server and call model methods
- Model contains logic
- Dependencies are injected by framework
- Works well with Presentation Model
- Reusable components with high testability
- Clean code

Same MVC



View dispatches events that get handled by the Controller



Controller invokes remote service (if necessary)



Controller tells the model to do something (usually with the data it received from the cloud)



After the model is done with it's logic, binding events get dispatched which cause View updates

Alternatively, the framework can inject new data into the view from the model.

Injectables

singletons (lowercase “s”)

Mate

- Defined in EventMap
- Uses ObjectBuilder
- Can cache a single instance or always create a new one

Swiz

- Defined in Beans.xml
- Declared as vars
- Use Prototype to create new instances

Injectables

Mate

inside EventMap.mxml

```
<!--  
    The FlexEvent.PREINITIALIZE event is a good place for creating and initializing managers.  
-->  
<EventHandlers type="{FlexEvent.PREINITIALIZE}">  
    <ObjectBuilder generator="{ DocumentFactory }"/>  
  
    <ObjectBuilder generator="{ ApplicationManager }">  
        <!--  
            "lastReturn" refers to the return value of the last action, in this case the object  
            created by the ObjectBuilder tag above  
        -->  
        <Properties documentFactory="{ lastReturn }"/>  
    </ObjectBuilder>  
  
    <ObjectBuilder generator="ClassNameToInstantiate" constructorArguments="{ ['argument1', 'argument2'] }" />  
</EventHandlers>
```

Injectables

Swiz

inside Beans.mxml

```
<swiz:BeanLoader xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:swiz="http://swiz.swizframework.org"
  xmlns:presentation="models.presentation.*"
  xmlns:models="models.*"
  xmlns:controllers="controllers.*">

  <mx:Script>
    <![CDATA[
      import models.presentation.EmployeeViewPresentationModel;
    ]]>
  </mx:Script>

  <models:ApplicationModel id="appModel" />

  <controllers:ApplicationController id="appController" />

  <presentation:MainPresentationModel id="mainPresentationModel" />

  <!--
    We use the Prototype tag so that we can pass a constructor argument. In this case the argument
    is the dispatcher property provided by BeanLoader, which will allow EmployeeViewPresentationModel
    to dispatch events that the event bus can hear.
  -->
  <swiz:Prototype id="employeePresoModel"
    classReference="{ EmployeeViewPresentationModel }"
    constructorArguments="{ dispatcher }"
    singleton="true" />

</swiz:BeanLoader>
```

Injection

Mate

- Defined in EventMap
- Uses Injectors
- Can inject properties
- Can inject listener functions

Swiz

- Uses [Autowire] metadata above a variable
- Autowire by interface
- Autowire by bean/property

Injection

Mate

Injecting a model into a view

```
<Injectors target="{ Login }" >  
  <PropertyInjector targetKey="model" source="{ LoginPresentationModel }" />  
</Injectors>
```

Injecting a model property into a view

```
<Injectors target="{ Login }" >  
  <PropertyInjector targetKey="userName" source="{ LoginPresentationModel }" sourceKey="userName" />  
</Injectors>
```

Injecting a property between classes

```
<Injectors target="{ LoginPresentationModel }" >  
  <PropertyInjector targetKey="loginStatus" source="{ AuthorizationManager }" sourceKey="status" />  
</Injectors>
```

Injecting a listener method

```
<Injectors target="{ Login }" >  
  <ListenerInjector eventType="{ SomeEvent.EVENT_TYPE }" method="eventTypeHandler" />  
</Injectors>
```



The downside of this approach is that we don't get compiler help for the property names because they're defined as strings. If you refactor and change the name of the property, you have to be careful to make sure the EventMaps are all updated as well.

The eventTypeHandler function must be a public method in Login - This method will be called by Mate any time the EventMap handles SomeEvent.EVENT_TYPE.

Injection

Swiz

Injection by type / interface

```
/**
 * Public member variable that will hold our presentation model.
 * Since [Autowire] does not include a bean attribute, Swiz will autowire by type.
 * Again, notice that even though the type of this property is IMainPresentationModel,
 * Swiz is smart enough to find a bean that implements this interface and inject it here.
 * Coding to interfaces is good!
 */
[Autowire]
[Bindable]
public var model:IMainPresentationModel;
```

Injection by property lookup

```
/**
 * Inject the employees property of ApplicationModel into this field.
 */
[Autowire( bean="appModel", property="employees" )]
[Bindable]
public var employees:ArrayCollection;
```



[Autowire] metadata can be used in any class, just like the injector target in Mate.

The preferred [Autowire] technique is to let the type/interface determine what gets injected.

Event Handling

all your dispatch are belong to us

Mate

- Defined in EventMap
- Uses EventHandlers
- bubbles="true"
- scope.dispatcher for non-display list classes

Swiz

- [Mediate] metadata above an AS function
- SwizConfig support
 - mediateBubbledEvents="true"
 - strict="true"
- Optional global dispatcher
- Prototype constructorArguments="dispatcher"

Event Handling

Mate

inside EventMap.mxml

```
<!-- Logging in, normally this would send a server request, but we are making it simple here -->
<EventHandlers type="{ LoginEvent.LOGIN }">
  <MethodInvoker generator="{ AuthorizationManager }" method="login"
    arguments="{ [ event.username, event.password ] }" />

  <MethodInvoker generator="{ NavigationManager }" method="updateAfterLogin"
    arguments="{ lastReturn }" />
</EventHandlers>
```

Swiz

inside Controller.as

```
[Mediate( event="UserEvent.SAVE_USER", properties="user" )]
public function saveUser( user:Object ):void
{
  for each ( var employee:Object in employees )
  {
    if ( employee.id == user.id )
    {
      employee.name = user.name;
      employee.location = user.location;
      break;
    }
  }
}
```



Again, more “magic strings” for method names in Mate.

Calling Services

Mate

- Use service invoker
- result/fault handler logic nested in inner tags
- UnhandledFaultEvent

Swiz

- Use executeServiceCall
- Define result/fault handler methods
- serviceCallFaultHandler

Calling Services

Mate

```
<EventHandlers type="{ FlexEvent.APPLICATION_COMPLETE }">
  <HTTPServiceInvoker instance="{ employeesService }">
    <resultHandlers>
      <MethodInvoker generator="{ EmployeeParser }" method="loadEmployeesFromXML" arguments="{ responseObject }" />
      <PropertySetter generator="{ EmployeeManager }" targetKey="employeeList" arguments="{ lastReturn }" />
    </resultHandlers>
  </HTTPServiceInvoker>
</EventHandlers>
```

Swiz

```
[Mediate( event="ContactEvent.SAVE", properties="contact" )]
public function save( contact:Contact ):void
{
    // "contact" at the end is pass-through data that will be available in the result handler
    executeServiceCall( contactService.save( contact ), save_resultHandler, null, [ contact ] );
}

private function save_resultHandler( event:ResultEvent, contact:Contact ):void
{
    contact.id = event.result.id;
}
```

Chain Async Calls

Mate

- Nest another service invoker in resultHandler

Swiz

- Use CommandChain

Mock Services

Mate

- Use MockRemoteObject with a mockGenerator class
- Services.xml swaps RemoteObject with MockRemoteObject

Swiz

- Service and Mock share interface
- Change implementation in Beans.xml
- Use TestUtil mockResult and mockFault

Mock Services

Mate

inside Services.mxml

```
<!--  
    <mx:RemoteObject id="myService" destination="MySampleService" />  
-->  
  
<MockRemoteObject id="myService" mockGenerator="{ MockSampleService }" delay="1" showBusyCursor="true" />
```

inside MockSampleService.as

```
public function getUserList():ArrayCollection  
{  
    var users:Array = new Array();  
    var user:User;  
  
    // Generate 10 fake users  
    for ( var i:int = 1; i <= 10; i++ )  
    {  
        user = new User();  
        user.id = i;  
        user.firstName = "First " + i;  
        // More here... Math.random(), lorem ipsum generation, etc.  
  
        users.push( user );  
    }  
  
    return new ArrayCollection( users );  
}
```



The key here is that both the RemoteObject and the MockRemoteObject have the same "id" value. We just pick which object we want to have that id - the rest of the application is unaware what the underlying implementation does.

Mock Services

Swiz

MockSampleDelegate.as

```
public class MockTwitterService implements ITwitterService
{
    public function updateStatus( status:String ):AsyncToken
    {
        if ( status == "success" )
        {
            var result:* = new Object();

            return TestUtil.mockResult( result, 2000, true );
        }
        else
        {
            return TestUtil.mockFault( new Fault( "1", "error" ), 1000, true );
        }
    }
}
```

inside Controller .as

```
[Autowired]
public var twitterService:ITwitterService;
```

inside Beans.xml

```
<services:TwitterService id="twitterService" />
<!--<services:MockTwitterService id="twitterService" />-->
```



Injection by interface here. We swap the “injectable” in Beans between the real and mock, but since they both implement the same interface, whichever service we define gets injected into the controller.

Who Wins?

Mate

- Injection is cleaner
- MXML logic is **easy**
- MXML logic is **weird**
- **Great** documentation
- **More** novice-friendly
- **More** “magic strings”

Swiz

- Logic all in
ActionScript
- More Testable
- **OK** documentation
(but it's getting
better)
- **Less** Framework
magic



There is no clear winner. It comes down to a matter of preference and style. The key question centers around if you can live with expressing actions in MXML. In general, Mate seems a bit easier learn because of the documentation and examples coupled with the simplicity of dealing with asynchronicity in resultHandler/faultHandlers.

Swiz Autowire on the views sometimes “know” too much about where the property is coming from when autowiring by bean/property name. Mate keeps that information out of the view, which makes the view a little more reusable.

Remember though, no matter which framework you choose, the overall application architecture will end up being similar. It's the same MVC!