Documentation

Getting Started

This tutorial explains how to get started with Mate. As an example, we'll create a stock quote retrieval screen which sends the quote symbol to the server, receives the current price and stores it in the model for the view to show.

All Mate projects must have:

1. One or more events (custom or built-in)
2. One or more Event Maps

Typically, the basic steps to create a Mate project are:

1. Add the compiled framework code to your project (Mate.swc).
2. Create a file that will be the EventMap.
3. Include the event map in your main Application file.
4. Create a custom event.
5. Somewhere, dispatch that event.
6. Add EventHandlers in your event map that listen for the event type you dispatched.
7. Execute some actions inside the EventHandlers block (ie: call the server, store data, etc).
8. Repeat 4-7 for every event you need.

Get the source for this tutorial

Creating a new project

In Flex Builder, create a new Flex project called "StockQuotesExample". Let the main source folder be "src" (default folder).

In the libs folder it creates, place the compiled framework SWC (Mate.swc). This will let you use all Mate classes and tags.

The Quote custom event

Every Mate project is driven by events. In the stock quote example, when the user enters the stock symbol and clicks on the "Get Quote" button, we'll create a new event containing that information that will be sent to the server. Therefore, we need to create a custom event to indicate that the user wants to submit the symbol and retrieve the current price.

Our event will be very simple and it will contain one property: the symbol.

```
package com.asfusion.mate.stockQuoteExample.events
{
   import flash.events.Event;
```

```
   public class QuoteEvent extends Event
   {
      public static const GET: String = "getQuoteEvent";

      public var symbol : String;


      public function QuoteEvent(type:String, bubbles:Boolean=true,
cancelable:Boolean=false)
      {
         super(type, bubbles, cancelable);
      }

   }
}
```

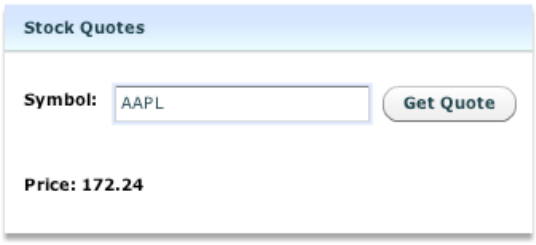
The code above assumes this event is contained within the package: com.asfusion.mate.stockQuoteExample.events

The event also contains a constant that we will use to specify the event type. One event can specify more than one event type.

We also make this event bubble up by default (the second argument of the constructor). Otherwise, we will need to remember to specify it when we instantiate it.

Creating the UI

The user interface will only need a text input and a button:



Dispatching the QuoteEvent

When the user clicks the Get Quote button, we'll create the QuoteEvent and dispatch it:

```
import com.asfusion.mate.stockQuoteExample.events.QuoteEvent;

private function getQuote() : void {

   var quoteEvent:QuoteEvent = new QuoteEvent(QuoteEvent.GET);
   quoteEvent.symbol = symbolInput.text;
```

```
    dispatchEvent(quoteEvent );
}
```

The Event Map

The EventMap is where we place the handlers for all the events the application creates (there could be more than one event map, though).

To add the event map, we need to create a new MXML file, with name "MainEventMap". This component must extend from EventMap. At this point, the event map would be empty and it should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<EventMap
    xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns="http://mate.asfusion.com/">

</EventMap>
```

Note: we use no namespace for http://mate.asfusion.com so that we don't have to add it to every tag in the event map. You can copy the code above to your file as a starting point.

We'll add the event map to our main Application file:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
xmlns:maps="com.asfusion.mate.stockQuoteExample.maps.*">

    <maps:MainEventMap />

</mx:Application>
```

Setting up debugging

In order to know whether our event map is receiving the events that get dispatched, we add the debugger tag to the event map:

```
<Debugger level="{Debugger.ALL}" />
```

The EventMap so far:

```
<?xml version="1.0" encoding="utf-8"?>
<EventMap xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="http://mate.asfusion.com/">

    <Debugger level="{Debugger.ALL}" />
```

```
</EventMap>
```

Listening for QuoteEvent.GET

In our event map, we will listen for the quote event so that we can send the request to the server.

We'll add an EventHandlers tag that will specify the event type we are listening to. We'll also set the debug attribute to true so that we can see when the handlers run in the debugging output window.

```
<EventHandlers type="{QuoteEvent.GET}" debug="true">

</EventHandlers>
```

At the top of the event map we'll need to import the event class.

```
<mx:Script>
   <![CDATA[
      import com.asfusion.mate.stockQuoteExample.events.QuoteEvent;
   ]]>
</mx:Script>
```

Inside this EventHandlers block, we'll place the actions we want to perform when the event is dispatched. In this case, we would like to make a server call, for which we'll use the RemoteObjectInvoker tag. Assuming the service in a folder called stockQuoteExample and it is called QuoteService, you will specify the call as follows:

```
<RemoteObjectInvoker destination="ColdFusion" source="stockQuoteExample.QuoteService"
   method="getQuote"
   arguments="{event.symbol}"
   debug="true">

</RemoteObjectInvoker>
```

We are calling the method getQuote on that service and sending the symbol coming from the event as an argument of the remote method call.

Handling the server result

The server returns a numerical value with the stock's current price. We will handle that result inside the RemoteObjectInvoker 's resultHandlers and call the function "storeQuote" on the QuoteManager class.

```
<EventHandlers type="{QuoteEvent.GET}" debug="true">
   <RemoteObjectInvoker destination="ColdFusion"
      source="stockQuoteExample.QuoteService"
      method="getQuote"
      arguments="{event.symbol}"
      debug="true">
```

```
      <resultHandlers>

         <MethodInvoker generator="{QuoteManager}"
            method="storeQuote" arguments="{resultObject}"/>

      </resultHandlers>

   </RemoteObjectInvoker>

</EventHandlers>
```

If you have shared data that many views will access, you may want to create a "model". In this simple example, you don't really need a model, but because it is something you will usually need, we'll add it anyway.

Inside the resultHandlers, we are using a MethodInvoker to create an instance of QuoteManager (if it doesn't already exist) and then call the method storeQuote. Inside the resultHandlers, we can access the result coming from the server, which we can pass as the argument of the method call.

Creating our Model, the QuoteManager

The QuoteManager will handle the business logic that has to do with quotes. It will also store the current symbol's quote so that it can be used by views. In the previous section, we were calling the method storeQuote(price) that stores the value of the current symbol's price. The class definition for this manager is:

```
package com.asfusion.mate.stockQuoteExample.business
{
   public class QuoteManager
   {

      [Bindable]
      public var currentPrice:Number;

      public function storeQuote(price:Number):void {
         currentPrice = price;
      }

   }
}
```

Ideally, the currentPrice property would be read-only instead of public. But in order to do that and still making it bindable, we will need to do some additional work.

Also, when the method storeQuote is called, we can execute any necessary business logic.

Showing the current price value in the view

So far, when the event is dispatched, we make a service call, the server returns the current price and that price is stored in the QuoteManager. But now we need to be able to show that value in the view.

In the view where we want to show the price, we'll add a property:

```
[Bindable]
public var price:Number;
```

and then show that number anywhere in the view we want, for example, in a Label:

```
<mx:Label text="Price: {price}" />
```

That's all we need in the view.

Getting the current price from the model Manager to the view

But how does the view get this variable populated from the price stored in the QuoteManager?

In our event map, we'll add another set of tags. These tags will assign a property on the model to a property on the view, and because the property on the model is bindable, the view will always get the most current value.

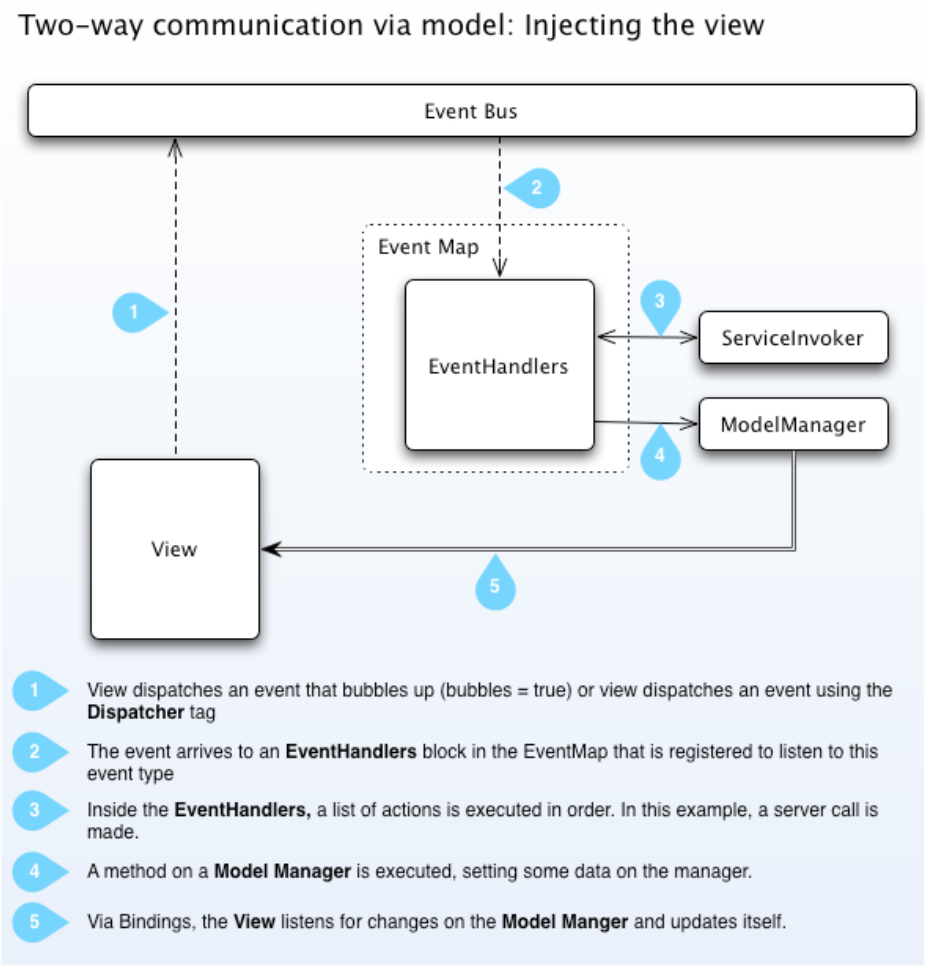
```
<Injectors target="{QuotePanel}">

   <PropertyInjector targetKey="price" source="{QuoteManager}" sourceKey="currentPrice"
/>

</Injectors>
```

When you add this, make sure you have all the necessary import statements at the top of your event map.

Event-Service call-Model Manager-View



View the example running, view the source and download the example in the example page.

Overview

Event map

Central to Mate is the **Event Map** . In the Event Map (or multiple event maps), you define what needs to happen when certain events are dispatched. Each event type you want to listen to will have its own **Event Handlers** block in the event map.

```
<EventMap xmlns="http://mate.asfusion.com/">
   ...event handlers blocks here...
</EventMap>
```

An EventMap is typically an stand-alone mxml file, and you must place it in your Application file. This requirement is due to the component creation cycle that Flex follows. If you place your event map deep inside your application, it may not be instantiated early enough to listen to all the events. Placing it in the Application file also allows you to be able to listen to early Flex events such as FlexEvent.PREINITIALIZE.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
```

```
xmlns:maps="com.yourdomain.maps.*">

   <maps:MyEventMap />

   ...other application components here...

</mx:Application>
```

Event Handlers

An **Event Handlers** block defined in the **Event Map** will run whenever an event of the type specified in the EventHandlers ' "type" argument is dispatched. Note that in order for the handlers to be able to listen for the given event, this event must have its bubbling setting as true and be dispatched from an object that has Application as its root ancestor, or the event must be dispatched by a Mate Dispatcher (such is the case when dispatching events from a PopUp window).

To define a an Event Handlers block, use the **EventHandlers** tag.

```
<EventHandlers type="myEventType">
... here what you want to happen when this event is dispatched...
</EventHandlers>
```

As mentioned earlier, you place these Event Handler lists inside an Event Map:

```
<EventMap xmlns="http://mate.asfusion.com/">

   <EventHandlers type="myEventType">
      ... here what you want to happen when this event is dispatched...
   </EventHandlers>

   <EventHandlers type="myOtherEventType">
      ... here what you want to happen when this other event is dispatched...
   </EventHandlers>

</EventMap>
```

Inside the EventHandlers tag, you will place all the handlers that need to be called when the event is dispatched.

For example, if you wanted the handler list to run, you must dispatch an event of type "myEventType". Say you have a button that when clicked, the event is dispatched (note that this event has its bubble property set to true):

```
private function buttonClicked():void {
 var event:MyCustomEvent = new MyCustomEvent("myEventType", true);
 dispatchEvent(event);
}
```

This example assumes you have a custom event class called MyCustomEvent. When the button is clicked and

the function buttonClicked() is executed, the first event handler list will receive the event and run each of the handlers it contains in order.

There are several handlers you can specify and each of them serves a different purpose:

- MethodInvoker
- CommandInvoker
- EventAnnouncer
- WebServiceInvoker
- HTTPServiceInvoker
- RemoteObjectInvoker
- ObjectBuilder
- DataCopier
- StopHandlers
- InlineInvoker
- ResponseAnnouncer

Method Invoker

MethodInvoker is one of the most used tags. When placed inside an **EventHandlers** tag and the handlers are executed, it will create an object of the class specified in the "generator" attribute. It will then call the function specified in the "method" attribute on the newly created object. You can pass arguments to this function that come from a variety of sources, such as the event itself, a server result object, or any other value.

```
<MethodInvoker
   generator="ClassNameToInstantiate"
   method="methodToExecute"
   arguments="{['argument1', 'argument2']}"/>
```

The above example would be the same as doing the following in ActionScript code:

```
var myWorker:ClassNameToInstantiate = new ClassNameToInstantiate();
myWorker.methodToExecute('argument1', 'argument2');
```

There are other ways of specifying attributes and there are additional attributes you can use. Refer to the documentation for detailed use information.

Command Invoker

The **CommandInvoker** tag is very similar to the **MethodInvoker** tag, but limited. It only allows to specify the generator class to instantiate. It will always call the method "execute" and pass the current event as its only argument. This tag is very useful when reusing Cairngorm commands.

```
<CommandInvoker generator="CommandClassNameToInstantiate" />
```

Event Announcer

Whenever you want to trigger another event inside an event handlers block, you use the **EventAnnouncer** tag.

```
<EventAnnouncer type="myEventType" generator="EventClassNameToInstantiate" />
```

You can also provide properties for the newly created event:

```
<EventAnnouncer type="myEventType" generator="MyEventClass">
   <Properties myProperty="myValue" myProperty2="other value"/>
</EventAnnouncer>
```

As in the case of the MethodInvoker arguments, properties for the new event can be brought from the original event, a server result, and other sources.

The above example would be the same as doing the following in ActionScript code:

```
var myEvent:MyEventClass = new MyEventClass("myEventType", true);
myEvent.myProperty = "myValue";
myEvent.myProperty2 = "other value";
dispatchEvent(myEvent);
```

The main difference is that in the case of the **EventAnnouncer**, the event is dispatched whenever the container event handler list is run and in the order specified in the list.

Service Calls

You can call different service types by using the HTTPServiceInvoker tag, the WebServiceInvoker tag or the RemoteObjectInvoker tag. If you have an already created service (HTTP service, Web Service, and Remote Object), you can use the "instance" attribute so that you don't have to specify the service properties in the Event Map. In that way, you can use a separate file to create all the services you use. You can use the Service Invoker tags (WebServiceInvoker, RemoteObjectInvoker, HTTPServiceInvoker) to create your services inline in the event map.

```
<WebServiceInvoker instance="{myServiceInstance}" method="serverMethodToCall"
arguments="{['argument1', 'argument2']}" />
```

When a service is called, there are two type of responses you can get: either a result with the contents of what the service returned, or a fault when there was an error when contacting the server or an error was thrown by the server. It's important to note that these possible responses are received asynchronously, so that after the request to the server was made, it is not possible to know when the response will be received.

When you use any of the Service Invoker tags inside the EventHandlers block, the server request is made and the list execution continues. Any other handlers placed after the service will execute right after the server request is made. But what if you want to wait until the server responds to the request? Inside the Service Invoker tags, you can place another list of handlers that will execute when the server replies, or throws an error.

```
<WebServiceInvoker instance="{myServiceInstance}" ...>

   <resultHandlers>
       ... this lsit of handlers executes when server returns results ...
   </resultHandlers>
```

```
</WebServiceInvoker>
```

In the same way, you can place another sequence that will execute when the request ends up in an error.

```
<WebServiceInvoker instance="{myServiceInstance}" ...>

   <faultHandlers>
      ... this sequence executes when server returns an error ...
   </faultHandlers>

</WebServiceInvoker>
```

Of course you can have both sequences inside one service tag:

```
<WebServiceInvoker instance="{myServiceInstance}" ...>

   <resultHandlers>
      ... this sequence executes when server returns results ...
   </resultHandlers>

   <faultHandlers>
      ... this sequence executes when server returns an error ...
   </faultHandlers>

</WebServiceInvoker>
```

WebService Invoker

The WebServiceInvoker tag is used to create a web service instance and call a method on the web service created. To use this tag, you need to specify its wsdl attribute that will determine the address of the webservice, or you use the *instance* attribute as specified in the previous section. You also need to specify the method to call.

```
<WebServiceInvoker wsdl="wsdAddress" method="methodToCall" arguments="{['argument1',
'argument2']}">

</WebServiceInvoker>
```

This tag will also accept all mx.rpc.soap.WebService tag attributes.

As specified in the previous section, you can have resultHandlers and faultHandlers inside the tag to handle service results and faults.

```
<WebServiceInvoker wsdl="wsdAddress" method="methodToCall" arguments="{['argument1',
'argument2']}">
   <resultHandlers>
```

```
      ... these handlers execute when server returns results ...
   </resultHandlers>

   <faultHandlers>
      ... these handlers execute when server returns an error ...
   </faultHandlers>
</WebServiceInvoker>
```

HTTPService Invoker

The HTTPServiceInvoker tag is used to create an HTTP Service instance and make a GET or POST request to that service. To use this tag, you need to specify the same attributes you would when creating an HTTP service with the <mx:HTTPService> tag, or you use the *instance* attribute as specified in the "Service Calls" section.

```
<HTTPServiceInvoker url="URLToCall" />
```

This tag will also accept all mx.rpc.http.HTTPService tag attributes.

You can have resultHandlers and faultHandlers inside the tag to handle service results and faults.

```
<HTTPServiceInvoker url="URLToCall">
   <resultHandlers>
      ... these handlers execute when server returns results ...
   </resultHandlers>
   <faultHandlers>
      ... these handlers execute when server returns an error ...
   </faultHandlers>
</HTTPServiceInvoker>
```

RemoteObject Invoker

The RemoteObjectInvoker tag is used to create a Remote Object instance and call a method on the object created. To use this tag, you need to specify the same attributes you would when creating a remote object with the <mx:RemoteObject> tag, or you use the *instance* attribute as specified in the "Service Calls" section. In addition, you need to specify what method to call.

```
<RemoteObjectInvoker
   destination="YourDestination"
   source="path.to.your.service"
   method="methodToCall"
   arguments="{['argument1', 'argument2']}" />
```

This tag will also accept all mx.rpc.remoting.RemoteObject tag attributes.

You can have resultHandlers and faultHandlers inside the tag to handle service results and faults.

```
<RemoteObjectInvoker destination="YourDestination" source="path.to.your.service"
```

```
method="methodToCall" arguments="{['argument1', 'argument2']}">
   <resultHandlers>
      ... this sequence executes when server returns results ...
</resultHandlers>
   <faultHandlers>
      ... this sequence executes when server returns an error ...
   </faultHandlers>
</RemoteObjectInvoker>
```

Data Copier

The DataCopier tag is a handy tag to quickly copy values into some storage. You can use the event handler list's "data" as a temporary storage from where handlers that follow in the list can read values. You can also use some other external variable as the storage.

```
<DataCopier destination="data" destinationKey="someProperty" source="result"
sourceKey="someProperty" />
```

Stop Handlers

Event Handlers run all handlers in order. If you need to stop the handlers execution before it reaches the end of the list, you can use the StopHandlers tag.

If there exists a MethodInvoker tag right before the StopHandlers tag, and the execution of the function returned a value, you can compare it to some other value and stop the handlers if it is equal.

```
<StopHandlers lastReturnEquals="someValue" />
```

A more flexible approach is to use the stopFunction attribute and handle the logic externally. The function that you implement will return true if the execution must stop or false if not.

```
<StopHandlers stopFunction="myStopSequenceFuntion" />
```

Then you implement your evaluation function:

```
private function myStopFunction(scope:Scope):Boolean {
 ... here you do some evaluation to determine
 whether to stop the execution of the list or not...
 return false;
    //or return true;
```

Message Handlers

The list of EventHandlers runs when an event of the type specified is dispatched. The list of MessageHandlers, on the other hand, runs whenever a Message sent by a Flex Messaging Service is received. This message must match certain criteria similar to the criteria specified when using the <mx:Consumer> tag.

To define the handlers, use the **MessageHandlers** tag.

```
<MessageHandlers destination="YourGateway">
   ... here what you want to happen when this message is received ...
</MessageHandlers>
```

Just like when using the EventHandlers tag, you place this tag inside an EventMap. Between the tag block, you place the actions to perform when the message is received. You can use the same tags as you would inside the EventHandlers tag: MethodInvoker, EventAnnouncer, etc.

Tags

Learn how to use each Mate tag

EventMap

A fundamental part of Mate is the **EventMap** tag. It allows you define mappings for the events that your application creates. It is basically a collection of EventHandlers blocks, where each block matches an event type.

```
<EventMap>
   ...event handler lists here...
</EventMap>
```

The EventMap tag doesn't have any attribute, but its importance lies in the tags in can contain.

Inner tags

EventHandlers

The event map contains an EventHandlers block for every event that you want to listen to. Each block is executed every time an event of the type specified is dispatched. Blocks are defined by using the EventHandlers tag.

```
<EventMap xmlns="http://mate.asfusion.com/">

   <EventHandlers type="myEventType">
      ... here what you want to happen when this event is dispatched...
   </EventHandlers>

   <EventHandlers type="myOtherEventType">
      ... here what you want to happen when this other event is dispatched...
   </EventHandlers>

</EventMap>
```

See **EventHandlers** for more information.

MessageHandlers

In addition to EventHandlers blocks, an event map can also contain MessageHandlers blocks. A block of **MessageHandlers** will be executed when a given Flex Messaging Service message is received. For the sequence to run, you must specify the same attributes you would in an <mx:Consumer> tag, so that when a message is received by the application, it is routed to the correct MessageHandler.

See MessageHandlers for more information.

An event map may contain any combination of EventHandlers and MessageHandlers tags:

```
<EventMap xmlns="http://mate.asfusion.com/">

   <EventHandlers type="myEventType">
      ... here what you want to happen when this event is dispatched...
   </EventHandlers>

   <MessageHandlers destination="ColdFusionGateway">
      ... here what you want to happen when this message is received ...
   </MessageHandlers>

</EventMap>
```

Using Smart Objects

The EventMap tag provides a set of Smart Objects that can be used within the EventHandlers and MessageHandlers tags. These objects expose data such as the current event, the value returned by a MethodInvoker or the result of a server call.

Examples of usage can be found in the documentation for MethodInvoker, EventAnnouncer and the other tags that can be used inside the EventHandlers and MessageHandlers.

These smart objects are specified by using bindings, but it is important to note that this binding is executed only once, at the beginning of the application.

event

Available only inside the EventHandlers tag. It refers to the event that made the list of handlers execute. The event itself or properties of the event can be used as arguments of MethodInvoker methods, service methods, properties for any of the handlers, etc.

For example, a method called by a MethodInvoker can receive event properties as arguments, or the event itself:

```
<EventHandlers type="myEventType">
```

```
   <MethodInvoker
      generator="{MyWorker}"
      method="doWork"
      arguments="{[event.userName, event.age]}"/>

   <MethodInvoker
      generator="{MyWorker}"
      method="doWork"
      arguments="{event}"/>
</EventHandlers>
```

message

Only available inside a MessageHandlers tag. It refers to the message received that made the list of handlers execute. The message itself or properties of the message can be used as arguments of MethodInvoker methods, service methods, properties for any of the handlers, etc.

For example, a method called by a MethodInvoker can receive message properties as arguments, or the message itself:

```
<MessageHandlers ...>
   <MethodInvoker
      generator="{MyWorker}"
      method="doWork"
      arguments="{[message.userName, message.age]}"/>

   <MethodInvoker
      generator="{MyWorker}"
      method="doWork"
      arguments="{message}"/>
</MessageHandlers>
```

resultObject

Only available inside a resultHandlers inner tag. It refers to the result returned by a service that made the result handlers execute. The result itself or properties of the result can be used as arguments of MethodInvoker methods, service methods, properties for any of the handlers, etc.

For example, a method called by a MethodInvoker can receive result properties as arguments, or the result itself:

```
<resultHandlers>
   <MethodInvoker
      generator="{MyWorker}"
      method="doWork"
      arguments="{[resultObject.userName, resultObject.age]}"/>

   <MethodInvoker
      generator="{MyWorker}"
```

```
      method="doWork"
      arguments="{resultObject}"/>
</resultHandlers>
```

See more information about resultHandlers inner tag at: " Handling a service result or fault ".

fault

Only available inside a faultSequence inner tag. It refers to the fault returned by a service that made the sub sequence execute. The fault itself or properties of the fault can be used as arguments of MethodInvoker methods, service methods, properties for any of the handlers, etc.

For example, an object instantiated by a MethodInvoker can receive fault properties as arguments, or the fault itself:

```
<faultHandlers>
   <MethodInvoker
      generator="{MyWorker}"
      method="doWork"
      arguments="{[fault.faultDetail, fault.errorID]}"/>

   <MethodInvoker
      generator="{MyWorker}"
      method="doWork"
      arguments="{fault}"/>
</faultHandlers>
```

See more information about faultHandlers inner tag at: "Handling a service result and fault".

lastReturn

lastReturn is always available, although its value might be *null* . It typically represents the value returned by a method call made by a MethodInvoker , but other handlers might also return a value, such as:

- **token**: returned by RemoteObjectInvoker , WebServiceInvoker and HTTPServiceInvoker (value is returned before call result is received)
- **boolean value** : returned by EventAnnouncer after dispatching the event. True for successful dispatch, false for unsuccessful (either a failure or when preventDefault() was called on the event).

All other handlers will nullify any previous value.

If that returned value is an object, you can access specific properties with the same syntax used by other smart objects: lastReturn.myProperty

See " Using lastReturn " in the MethodInvoker tag documentation.

data

Every EventHandlers and MessageHandlers block contain a placeholder object called "data". This object can

be used to store temporary data that many tags in the list can share.

For example, a function called by a MethodInvoker can receive data properties as arguments, or the data itself:

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{data}"/>

<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{[data.userName, event.age]}"/>
```

scope

Every handlers list contains an object (IScope) that represents the running list's scope. As an IScope, you can use it to stop the currently running list. If you cast it to Scope, for example, you can also access an event dispatcher, the current event and other properties. This scope can be used by functions called by MethodInvoker s.

Smart Objects limitations

SmartObject cannot be casted. For example, if you are expecting a custom event MyEvent, you cannot cast the event SmartObject to your custom class. Therefore, this is not allowed:
arguments="{[MyEvent(event).username]}"

Where should the Event Map go?

Typically, the Event Map is a standalone mxml file, although it can be placed inside any other mxml file. When the Event Map is a standalone file, any namespace used inside must be added. Inside the event map, we have the default namespace to be the mate the namespace so that we don't have to prefix the tags with the namespace.

```
<EventMap>
...event handler lists here...
</EventMap>
```

The best place to put the event map is in the Application file. This is due to the component creation cycle that Flex follows. If you place your event map deep inside your application, it may not be instantiated early enough to listen to all the events. Placing it in the Application file also allows you to be able to listen to early Flex events such as FlexEvent.PREINITIALIZE.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
      xmlns:maps="com.yourdomain.maps.*">
```

```
   <maps:MyEventMap />

   ...other application components here...

</mx:Application>
```

EventHandlers

(This tag must be placed inside an < EventMap > tag)

The list of event handlers defined by the **EventHandlers** tag that is defined in the EventMap will run whenever an event of the type specified in the "type" argument is dispatched. Note that in order for the list to be able to listen for the given event, this event must have its bubbling setting as true and be dispatched from an object that has Application as its root ancestor, or the event must be dispatched by a Mate Dispatcher (such is the case when dispatching events from a PopUp window).

```
<EventHandlers type="myEventType">
   ... here what you want to happen when this event is dispatched...
</EventHandlers>
```

Attributes

type

String

required

The event type that, when dispatched, should trigger the list of handlers to run. It should match the type specified in the event when created. All events extending from flash.events.Event have a "type" property.

While this attribute is a string, a constant is most commonly used:

```
<EventHandlers type="{MyEvent.MY_EVENT_TYPE}">
   ... here what you want to happen when this event is dispatched...
</EventHandlers>
```

Any bubbling up event or an event dispatched by a Mate Dispatcher can be used, including Flex built-in events such as FlexEvent.APPLICATION_COMPLETE.

priority

Number

If you expect to have several listeners for the same events (the Event Map, several views, etc), you can assign

each listener a different priority to manage the order at which those listeners will be notified. By default, the listeners are notified by the order at which they were registered (when they have the same priority). The listeners will be called from highest priority (highest number) to lowest. Default value is 0.

Note that as Flex documentation states, although the listeners will be called in that order, there is no guarantee that one listener will finish execution before the others are called.

debug

Boolean

Whether to show debugging information for this event handlers block. If true, Console output will show debugging information as this list runs.

start

Event handler

In the start attribute you can supply an event handler for the event of type ActionListEvent.START. This event is dispatched right before the handlers are called, when the list starts execution.

Example:

```
<EventHandlers type="{MyEvent.MY_EVENT_TYPE}" start="trace('Execution of handlers list started!')">
   ...
</EventHandlers>
```

end

Event handler

In the end attribute you can supply an event handler for the event of type ActionListEvent.END. This event is dispatched right after all the handlers have been called, when the list ends execution (although this event might be dispatched before asynchronous calls have returned).

Example:

```
<EventHandlers type="{MyEvent.MY_EVENT_TYPE}" end="trace('Execution of handlers list ended!')">
   ...
</EventHandlers>
```

Inner tags

Allowed inner tags:

- MethodInvoker
- EventAnnouncer

- RemoteObjectInvoker
- WebServiceInvoker
- HTTPServiceInvoker
- ObjectBuilder
- ResponseAnnouncer
- ServiceResponseAnnouncer
- CommandInvoker
- DataCopier
- StopHandlers

Order of the inner tags is important

The order in which the inner tags ("handlers") are placed is important because it determines the order in which each action will be taken.

For example, say you have defined the following sequence:

```
<EventHandlers type="myEventType">

   <!-- Step 1 -->
   <MethodInvoker generator="WorkerOne" method="doSomeWork" />

   <!-- Step 2 -->
   <EventAnnouncer type="myEventTypeTwo" generator="EventClassNameToInstantiate" />

   <!-- Step 3 -->
   <MethodInvoker generator="WorkerTwo" method="doSomethingElse" />

</EventHandlers>
```

In this scenario, when an event of type "myEventType" is dispatched, the following will happen:

1. An instance of WorkerOne will be created and the method doSomeWork() will be called
2. A new event of type "myEventTypeTwo" will be dispatched.
3. An instance of WorkerTwo will be created and the method doSomethingElse() will be called

For more information about the list order when using asynchronous services, refer to " Handling a service result or fault ".

MessageHandlers

(This tag must be placed inside an < EventMap > tag)

The MessageHandlers tag allows you to register a list of handlers as a consumer of a Flex Messaging Service. All the tags inside the MessageHandlers tag will be executed in order when a Message matching the criteria is received. This tag accepts the same attributes as the <mx:Consumer> tag.

```
<MessageHandlers destination="YourGateway">
   ... here what you want to happen when this message is received ...
```

```
</MessageHandlers>
```

The above example would be the same as doing the following:

```
<mx:Consumer destination="YourGateway" />
```

You would also need to create event handlers for the message and fault. That is not necessary when using the MessageHandlers tag. Inner tags such as MethodInvoker are executed when the message is received. If you need to handle a fault, you can use the faultHandlers inner tag. See "Handling a subscription fault".

Attributes

debug

Boolean

Whether to show debugging information for this handlers block. If true, Console output will show debugging information as the list runs.

Other attributes

See mx.messaging.Consumer for a list of attributes.

Inner tags

faultHandlers

A list of handlers to run if the subscription attempt throws an error. See "Handling a subscription fault" below.

Other allowed inner tags:

- MethodInvoker
- EventAnnouncer
- RemoteObjectInvoker
- WebServiceInvoker
- HTTPServiceInvoker
- ResponseAnnouncer
- DataCopier
- StopHandlers
- ServiceResponseAnnouncer
- ObjectBuilder

Handling a subscription fault

When the MessageHandlers tag is placed in the EventMap, it automatically subscribes to the destination provided. The subscription attempt may throw an error (ie: attempting to subscribe to a subtopic in a channel that doesn't accept subtopics). If you want to perform some action when the subscription cannot be made, you can use a <faultHandlers> inner tag. Inside that handlers list, you can use the same tags you would in the main body of an < EventHandlers > or <MessageHandlers> tag.

MethodInvoker

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

MethodInvoker is one of the most used tags. When placed inside a EventHandlers tag and the list of handlers is executed, it will create an object of the class specified in the "generator" attribute. It will then call the function specified in the "method" attribute on the newly created object. You can pass arguments to this function that come from a variety of sources, such as the event itself, a server result object, or any other value. Unless you specify cache="false", this object instance will be "cached" and not instantiated again.

Example:

```
<MethodInvoker
   generator="ClassNameToInstantiate"
   method="methodToExecute"
   arguments="{['argument1', 'argument2']}" />
```

The above example would be the same as doing the following in ActionScript code:

```
var myWorker:ClassNameToInstantiate = new ClassNameToInstantiate();
myWorker.methodToExecute('argument1', 'argument2');
```

The main difference is that in the case of the MethodInvoker, the object is created and the function executed whenever the list of handlers is run and in the order specified in the list.

Note: To call static methods, methods on a singleton object, or on an object that has been created elsewhere, use the InlineInvoker tag instead.

Attributes

generator

required

The generator attribute specifies what class should be instantiated and run.

Suppose you have a class called "MyWorker" in the package com.yourdomain.business. You can specify a complete path to com.yourdomain.business.MyWorker:

```
<MethodInvoker
   generator="com.yourdomain.business.MyWorker"
   method="methodToExecute"
   arguments="{['argument1', 'argument2']}"/>
```

Generally you may want to use a binding to specify the class name. Assuming you have an import statement like this in your Event Map:

```
import com.yourdomain.business.MyWorker;
```

or simply:

```
import com.yourdomain.business.*;
```

You can then instantiate your object using bindings:

```
<MethodInvoker
   generator="{MyWorker}"
   method="methodToExecute"
   arguments="{['argument1', 'argument2']}"/>
```

The advantage of using this syntax is that if you are using Flex Builder, you can press the command key (Mac) or the Ctrl key (Windows) and click on the generator class (MyWorker in the example) and it will take you to the class definition.

method

The method attribute specifies what function to call on the created object. If your class MyWorker contains a function called doWork(), you would then write:

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
/>
```

arguments

If the function in your class has arguments, you can pass them via the "arguments" attribute. Suppose your doWork function has the following signature:

```
public function doWork(name:String, value:Number)
```

then you can pass those arguments as follows:

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{['Tom', 36]}"/>
```

Note that the arguments attribute expects an array. Besides passing literal values, you can pass values coming from the event that triggered the execution of the list of handlers:

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{[event.userName, event.age]}"/>
```

This assumes that the event contained a userName property and an age property.

If this method invoker tag is inside an <resultHandlers> tag that originated from a < WebServiceInvoker >, <HTTPServiceInvoker > or <RemoteObjectInvoker > tag, you can also pass values coming from the server result:

```
<RemoteObjectInvoker .....service attributes....>

   <resultHandlers>

      <MethodInvoker
         generator="{MyWorker}"
         method="doWork"
         arguments="{[resultObject.userName, resultObject.age]}"/>

   </resultHandlers>

</RemoteObjectInvoker>
```

Again, this assumes the server returned an object that contained a userName property and an age property.

If this MethodInvoker tag is inside an <faultHandlers> tag that originated from one of the service invoker tags, you can also pass values coming from the server fault. This comes in handy when you want to handle possible server errors.

```
<RemoteObjectInvoker .....service attributes....>

   <faultHandlers>

      <MethodInvoker
         generator="{MyWorker}"
         method="doWork"
         arguments="{[fault.faultDetail, fault.errorID]}"/>

   </faultHandlers>

</RemoteObjectInvoker>
```

You can also pass the complete event, resultObject, fault, data or lastReturn as an argument depending on what your function expects.

For example, if the function has the following signature:

public function doWork(event:MyCustomEvent):void

you would be able to send the event as an argument of the method:

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{[event]}"/>
```

If you wanted to send a service result, you would pass the resultObject as an argument.

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{[resultObject]}"/>
```

Your worker function signature would depend on what you are expecting to receive from the server. If you are using Flash Remoting, you might be getting a mapped class, such as a Customer.

public function doWork(customer:Customer)

If you are using HTTPService, you might be getting an XML object:

public function doWork(xmlDoc:XML)

You can also use the service fault, the sequence data or a lastReturn.

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{[fault]}"/>
```

The data is always an object, so your function must accept an object:

public function doWork(data:Object)

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{[data]}"/>
```

Receiving the value returned by a previous MethodInvoker execution:

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{[lastReturn]}"/>
```

Of course you can use any combination of arguments:

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{[event.age, resultObject, 'Tom']}"/>
```

Specifying only one argument

Although the arguments attribute expects an array, it is also possible to supply an object when there is only one argument that the worker function expects. See these examples:

```
<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="A string"/>

<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{event}"/>

<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{event.userName}"/>

<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{resulObject}"/>

<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{resulObject.userId}"/>

<MethodInvoker
   generator="{MyWorker}"
   method="doWork"
   arguments="{lastReturn}"/>
```

Specifying the arguments in verbose mode

Finally, arguments can also be passed by using SmartObjects.

For example, if you want to pass event.userName and event.age, you can pass them as follows:

```
<MethodInvoker generator="{MyWorker}" method="doWork">
   <arguments>
      <SmartObject source="event" key="userName" />
      <SmartObject source="event" key="age" />
   </arguments>
</MethodInvoker>
```

cache

Boolean

The cache attribute lets you specify whether this newly created object should be kept live so that the next time an instance of this class is requested, this already created object is returned instead.

For example, say you have two sequences that want to use the same instance of an object. Since the default value for this attribute is "true", it will do that by default. On the other hand, if you wanted to have two different instances, then you must set this attribute to "false".

```
<EventHandlers type="myEventType">

   <MethodInvoker
      generator="{MyWorker}" method="doWork"
      cache="false" />

</EventHandlers>

<EventHandlers type="myOtherEventType">

   <MethodInvoker
   generator="{MyWorker}" method="doDifferentWork"
   cache="false" />

</EventHandlers>
```

Inner tags

Properties

You can add properties to your object by using the Properties tag inside the MethodInvoker tag. These properties will be set before calling the function specified in the "method" attribute, so you can be sure that those properties will be available when the function is executed. These properties must be public.

Suppose you are creating an instance of a ShippingCalculator class. This class has a property called weightFactor and flatFee. In order to set those two properties, you can use the <Properties> inner tag. As attributes of the Properties tag, you can specify the names of your properties and set the values of those properties by setting the value of those attributes as follows:

```
<MethodInvoker
```

```
   generator="{ShippingCalculator}"
   method="calculateShipping" .... >

      <Properties weightFactor="0.5" flatFee="3" />

</MethodInvoker>
```

Besides specifying literal values, you can assign values coming from the event that triggered the handlers:

```
<MethodInvoker
   generator="{ShippingCalculator}"
   method="calculateShipping" .... >

      <Properties weightFactor="{event.factor}" flatFee="{event.fee}" />

</MethodInvoker>
```

This assumes that the original event contained a factor property and a fee property.

If this MethodInvoker tag is inside a <resultHandlers> tag that originated from one of the service invoker tags, you can also pass values coming from the server result:

```
<RemoteObjectInvoker .....service attributes....>

   <resultHandlers>

      <MethodInvoker
         generator="{ShippingCalculator}"
         method="calculateShipping" .... >

            <Properties weightFactor="{resultObject.factor}"
flatFee="{resultObject.fee}" />

      </MethodInvoker>

   </resultHandlers>

</RemoteObjectInvoker>
```

This assumes the server returned an object that contained a factor property and a fee property.

If this MethodInvoker tag is inside an <faultHandlers> tag that originated from one of the service builder tags, you can also pass values coming from the server fault. This comes in handy when you want to handle possible server errors.

```
<RemoteObjectInvoker .....service attributes....>

   <resultHandlers>

      <MethodInvoker
         generator="{MyErrorHandler}"
         method="handleShippingError" .... >

            <Properties myErrorProperty="{fault.faultDetail}" />

      </MethodInvoker>

   </resultHandlers>

</RemoteObjectInvoker>
```

You can also use the complete resultObject, fault or data as a property depending on what your event properties are.

For example, if the server returned a number with the value of the flat fee, you would be able to set the property "flatFee" with the server result:

```
<MethodInvoker
   generator="{ShippingCalculator}"
   method="calculateShipping" .... >

      <Properties flatFee="{resultObject}" />

</MethodInvoker>
```

You can use the service fault:

```
<MethodInvoker
   generator="{MyErrorHandler}"
   method="handleShippingError" .... >

      <Properties error="{fault}" />

</MethodInvoker>
```

You can also use the sequence data. The data property must be of type Object.

```
<MethodInvoker
   generator="{ShippingCalculator}"
   method="calculateShipping" .... >

      <Properties myProperty="{data}" />
```

```
</MethodInvoker>
```

Finally, you can use the lastReturn

```
<MethodInvoker
   generator="{ShippingCalculator}"
   method="calculateShipping" .... >

      <Properties myProperty="{lastReturn}" />

</MethodInvoker>
```

Using lastReturn

Every tag inside a handlers block generates a "lastReturn" that can be used by the immediately next tag. While most tags do not return any value, you can make your method invokers take advantage of it.

If your function call returns something other than void, that value will be stored in the scope's "lastReturn" value. This value can be used by other tags as they would use the event, the result object or server faults. This value will be null if your function returns void.

For example, say you have a class that calculates shipping costs and you want to send that value along with other things to the server. So your function would look like:

```
public function calculateShipping(weight:Number):Number {
   //do calculation
   return someNumber;
}
```

In your handlers block, you will call your class and then send the information to the server:

```
<MethodInvoker generator="{MyClass}" method="calculateShipping"
arguments="{[event.itemWeight]}" />

<WebServiceInvoker arguments="{[lastReturn, event.itemId]}" .....service attributes....>
   ...
</WebServiceInvoker>
```

Because your method call returned a value, the next tag (the WebServiceInvoker) can use it.

More information at Using Smart Objects (lastReturn) .

CommandInvoker

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

The CommandInvoker tag is very similar to the MethodInvoker tag, but limited. It only allows specifying the generator class to instantiate. It will always call the method "execute" and pass the current event as its only argument. This tag is very useful when reusing Cairngorm commands.

```
<CommandInvoker generator="CommandClassName" />
```

The above example would be the same as doing the following in ActionScript code:

```
var myCommand:CommandClassName = new CommandClassName();
myCommand.execute(event);
```

CommandInvoker is only a short-cut tag, as the same can be accomplished with a MethodInvoker tag:

```
<MethodInvoker
   generator="CommandClassNameToInstantiate"
   method="execute"
   arguments="{event}"/>
```

Note: when placed inside a MessageHandlers list, the command will receive a MessagingEvent.

Services

RemoteObjectInvoker

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

The RemoteObjectInvoker tag is used to create a Remote Object instance and call a method on the object created. To use this tag, you need to specify the same attributes you would when creating a remote object with the <mx:RemoteObject> tag. In addition, you need to specify what method to call. This tag will also accept all mx.rpc.remoting.RemoteObject tag attributes (with the exception of the "operations" property).

<RemoteObjectInvoker destination=" YourDestination " source=" path.to.your.service " method=" methodToCall " arguments=" {['argument1' , 'argument2']}" />

The above example would be the same as doing the following:

<mx:RemoteObject id=" myService " destination=" ColdFusion " source=" path.to.this.service " />
myService.methodToCall('argument1' , 'argument2');

You would also need to either create event handlers for the server result and fault or create a Responder to

handle them. That is not necessary when using the RemoteObjectInvoker tag because results and faults are handled by Mate. See section " Handling a service result or fault ".

With this tag, you can also utilize an already created RemoteObject instance. This is useful when the same services are used by several EventHandlers blocks or several EventMaps.

Suppose you have a Remote Object tag (either in the event map itself or in a different file):

```
<mx:RemoteObject id="myService " destination="ColdFusion " source="path.to.this.service " />
```

You can call a method on that already created service as follows:

```
<RemoteObjectInvoker instance="{myService}"
 method="methodToCall"
 arguments="{['argument1', 'argument2']}" />
```

Attributes

method

required

The method attribute specifies what function to call on the remote object instance.

arguments

If the remote method has arguments, you can pass them via the "arguments" attribute.

Suppose you have a RemoteObject method called getPhotos and it expects a user name and an album name as arguments. You can specify them with the arguments attribute:

```
<RemoteObjectInvoker
 destination="YourDestination" source="path.to.your.service"
 method="getPhotos"
 arguments="{['Tom','My Album']}" />
```

Note that the arguments attribute expects an array. Besides passing literal values, you can pass values coming from the event that triggered the list execution:

```
<RemoteObjectInvoker
 destination="YourDestination" source="path.to.your.service"
 method="getPhotos"
 arguments="{[event.userName, event.album]}"/>
```

This assumes that the event contained a userName property and an album property.

You can also pass the complete data or lastReturn as an argument depending on what your service expects.

```
<RemoteObjectInvoker
 destination="YourDestination" source="path.to.your.service"
 method="getPhotos"
 arguments="{data}"/>
```

```
<RemoteObjectInvoker
 destination="YourDestination" source="path.to.your.service"
 method="getPhotos"
 arguments="{lastReturn}"/>
```

Of course you can use any combination of arguments:

```
<RemoteObjectInvoker
 destination="YourDestination" source="path.to.your.service"
 method="getPhotos"
 arguments="{[event.age, lastReturn, 'Tom']}"/>
```

instance

A RemoteObject instance

The instance attribute specifies the already created service instance to use to make the call. This attribute must be supplied using bindings because it needs to point to an already created object.

Suppose you have a RemoteObject already created:

<mx:RemoteObject id=" myService " destination=" ColdFusion " source=" path.to.this.service " />

You can make a call to this service by using the RemoteObjectInvoker tag, with instance {myService}. Note that any property you need for your service must be defined in the RemoteObject tag.

<RemoteObjectInvoker **instance=" {myService }"** />

debug

Boolean

Whether to show debugging information for this RemoteObject resultHandlers and faultHandlers. If true, Console output will show debugging information as those handlers run.

Inner tags

resultHandlers

A set of handlers to run when the server call returns a result. Inside this inner tag, you can use the same tags you would in the main body of an < EventHandlers > block, including other service calls.

faultHandlers

A set of handlers to run when the server call returns a fault. Inside this inner tag, you can use the same tags you would in the main body of an < EventHandlers >, including other service calls.

Handling a service result or fault

You can have a resultHandlers block and a faultHandlers block inside the tag to handle service results and faults.

```
<RemoteObjectInvoker destination="YourDestination"
   source="path.to.your.service"
 method="methodToCall"
 arguments="{['argument1', 'argument2']}">

 <resultHandlers>
  ... this list executes when server returns results ...
 </resultHandlers>

 <faultHandlers>
  ... this list executes when server returns an error ...
 </faultHandlers>

</RemoteObjectInvoker>
```

See "Handling a service result or fault " for more information.

HTTPServiceInvoker

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

The HTTPServiceInvoker tag is used to create an HTTP Service instance and make a GET or POST request to that service. To use this tag, you need to specify the same attributes you would when creating an HTTP service with the <mx:HTTPService> tag (with the exception of xmlDecode, xmlEncode, and lastResult). In addition, you need to specify what method to call. This tag will also accept all mx.rpc.http.HTTPService tag attributes.

<HTTPServiceInvoker url="URLToCall " />

The above example would be the same as doing the following:

<mx:HTTPService id="myServiceInstance " url="http://www.example.com/services ">

myServiceInstance.send();

You would also need to either create event handlers for the server result and fault or create a Responder to handle them. That is not necessary when using the HTTPServiceInvoker tag. See section "Handling a service result or fault".

With this tag, you can also utilize an already created HTTPService instance. This is useful when the same services are used by several EventHandlers blocks or several EventMaps.

Suppose you have a HTTP Service tag (either in the event map itself or in a different file):

<mx:HTTPService id="myService " url="http://www.example.com/services " />

You can call a method on that already created service as follows:

<HTTPServiceInvoker instance="{myService }" />

Attributes

url

required if no instance is specified

This attribute specify the address to send the request. Refer to mx.rpc.http.HTTPService documentation for more information.

instance

An HTTPService instance

The instance attribute specifies the already created service instance to use to make the call. This attribute must be supplied using bindings because it needs to point to an already created object.

Suppose you have a HTTPService already created:

<mx:HTTPService id=" myService " url=" http://www.example.com/services " />

You can make a call to this service by using the HTTPServiceInvoker tag, with instance {myService}. Note that any property you need for your service must be defined in the HTTPService tag.

```
<HTTPServiceInvoker instance="{myService}" />
```

debug

Boolean

Whether to show debugging information for this HTTPService resultHandlers and faultHandlers. If true, Console output will show debugging information as those handlers run.

Inner tags

Request

You can add variables to your HTTP request by using the Request inner tag. If using GET method, these variables will be sent as query string parameters. Otherwise, they will be added to the POST request.

As attributes of the Request tag, you specify the names of the variables you want to send and set the values of those variables by settings the value of the attributes.

Suppose you need to send a "photo_id" variable and a username variable with your request. Then you will specify:

```
<HTTPServiceInvoker instance="{myService}">
 <Request
  photo_id="17"
```

```
  username="{event.username}" />
</HTTPServiceInvoker>
```

resultHandlers

A set of handlers to run when the server call returns a result. Inside this inner tag, you can use the same tags you would in the main body of an < EventHandlers > block, including other service calls.

faultHandlers

A set of handlers to run when the server call returns a fault. Inside this inner tag, you can use the same tags you would in the main body of an < EventHandlers >, including other service calls.

Handling a service result or fault

You can have a resultHandlers block and a faultHandlers block inside the tag to handle service results and faults.

```
<HTTPServiceInvoker url="URLToCall">

 <resultHandlers>
  ... this list executes when server returns results ...
 </resultHandlers>

 <faultHandlers>
  ... this list executes when server returns an error ...
 </faultHandlers>

</HTTPServiceInvoker>
```

See " Handling a service result or fault " for more information.

WebServiceInvoker

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

The WebServiceInvoker tag allows you to create a Web Service (mx.soap.WebService) in your handlers list and call a method on that web service, in one step. To use this tag, you need to specify its wsdl attribute that will determine the address of the webservice. You also need to specify the method to call. In addition to those two, this tag will accept all mx.rpc.soap.WebService tag attributes (with the exception of xmlSpecialCharsFilter and operations).

```
<WebServiceInvoker
 wsdl="/myservices/myService.cfc?wsdl"
 method="serverMethodToCall"
 arguments="{['argument1', 'argument2']}" />
```

The above example would be the same as doing the following:

<mx:WebService id="myServiceInstance" wsdl="/myservices/myService.cfc?wsdl">

myServiceInstance.serverMethodToCall('argument1' , 'argument2');

You would also need to either create event handlers for the server result and fault or create a Responder to handle them. That is not necessary when using the WebServiceInvoker tag.

See section " Handling a service result or fault ".

With this tag, you can also utilize an already created WebService instance. This is useful when the same services are used by several EventHandlers blocks or when using multiple event maps.

Suppose you have a WebService tag (either in the event map itself or in a different file):

<mx:WebService id="myServiceInstance " wsdl="/myservices/myService.cfc?wsdl ">

You can call a method on that already created service as follows:

<WebServiceInvoker instance="{myService }" method="serverMethodToCall " />

Attributes

wsdl

required if no instance is specified

This attribute specifies the address for the WebService definition file. Refer to mx.rpc.soap.WebService documentation for more information.

method

required

The method attribute specifies what function to call on the web service instance.

arguments

If the remote method has arguments, you can pass them via the "arguments" attribute.

Suppose you have a WebService method called getPhotos and it expects a user name and an album name as arguments. You can specify them with the arguments attribute:

```
<WebServiceInvoker
 wsdl="/myservices/myService.cfc?wsdl"
 method="getPhotos"
 arguments="{['Tom','My Album']}" />
```

Note that the arguments attribute expects an array. Besides passing literal values, you can pass values coming from the event that triggered the list of handlers to execute:

```
<WebServiceInvoker
```

```
 wsdl="/myservices/myService.cfc?wsdl"
 method="getPhotos"
 arguments="{[event.userName, event.album]}"/>
```

This assumes that the event contained a userName property and an album property.

You can also pass the complete data or lastReturn as an argument depending on what your service expects.

```
<WebServiceInvoker
 wsdl="/myservices/myService.cfc?wsdl"
 method="getPhotos"
 arguments="{data}"/>

<WebServiceInvoker
 wsdl="/myservices/myService.cfc?wsdl"
 method="getPhotos"
 arguments="{lastReturn}"/>
```

Of course you can use any combination of arguments:

```
<WebServiceInvoker
 wsdl="/myservices/myService.cfc?wsdl"
 method="getPhotos"
 arguments="{[event.age, lastReturn, 'Tom']}"/>
```

instance

A WebService instance

The instance attribute specifies the already created service instance to use to make the call. This attribute must be supplied using bindings because it needs to point to an already created object.

Suppose you have a WebService already created:

```
<mx:WebService  id=" myServiceInstance  " wsdl="/myservices/myService.cfc?wsdl  ">
```

You can make a call to this service by using the WebServiceInvoker tag, with instance {myService}. Note that any property you need for your service must be defined in the WebService tag.

```
<WebServiceInvoker  instance=" {myService }" method="  getPhotos " />
```

debug

Boolean

Whether to show debugging information for this WebServiceInvoker resultHandlers and faultHandlers. If true, Console output will show debugging information as those handlers run.

Inner tags

resultHandlers

A set of handlers to run when the server call returns a result. Inside this inner tag, you can use the same tags you would in the main body of an < EventHandlers > block, including other service calls.

faultHandlers

A set of handlers to run when the server call returns a fault. Inside this inner tag, you can use the same tags you would in the main body of an < EventHandlers >, including other service calls.

Handling a service result or fault

Just like all the "service invoker" tags, you can have a resultHandlers block and a faultHandlers block inside the tag to handle service results and faults.

```
<WebServiceInvoker
 wsdl="/myservices/myService.cfc?wsdl"
 method="serverMethodToCall"
 arguments="{['argument1', 'argument2']}" >

 <resultHandlers>
  ... this list executes when server returns results ...
 </resultHandlers>

 <faultHandlers>
  ... this list executes when server returns an error ...
 </faultHandlers>

</WebServiceInvoker>
```

See " Handling a service result or fault " for more information.

Handling a service result or fault

When a service is called, there are two type of responses you can get: either a result with the contents of what the service returned, or a fault when there was an error when contacting the server or an error was thrown by the server. It's important to note that these possible responses are received asynchronously, so that after the request to the server was made, it is not possible to know when the response will be received.

When you use the service tags (RemoteObjectInvoker , WebServiceInvoker , HTTPServiceInvoker) inside an EventHandlers block or MessageHandlers block and the event is dispatched or message is received, the server request is made and the execution of the handlers continues. Any other handler placed after the service will execute right after the server request is made. But what if you want to wait until the server responds to the request? Inside all of the service invoker tags, you can place another list of handlers that will execute when thes erver replies, or throws an error.

```
<WebServiceInvoker instance="{myServiceInstance}" ...>

 <resultHandlers>
  ... this list executes when server returns results ...
 </resultHandlers>

</WebServiceInvoker>
```

In the same way, you can place another list of handlers that will execute when the request ends up in an error.

```
<WebServiceInvoker instance="{myServiceInstance}" ...>

 <faultHandlers>
  ... this list executes when server returns a fault ...
 </faultHandlers>

</WebServiceInvoker>
```

Of course you can have both sequences inside one service tag:

```
<WebServiceInvoker instance="{myServiceInstance}" ...>

 <resultHandlers>
  ... this list executes when server returns results ...
 </resultHandlers>

 <faultHandlers>
  ... this list executes when server returns a fault ...
 </faultHandlers>

</WebServiceInvoker>
```

Inside the resultHandlers or faultHandlers tags, you can place any other tag you would normally place in your <EventHandlers > block. They are handler lists that run asynchronously when the server response is received. Inside the resultHandlers, you have a special value your handlers that are inside the result or fault handlers block can use: the"resultObject" that contains what was sent by the server as a response to the call made. Inside the faultHandlers, they can access the"fault" that contains the server error. Those two values can be used as arguments for MethodInvoker s, EventAnnouncer s, other services, etc.

Those two sub-handlers are only run after the server response is received, so you can be sure that you will have the values needed to continue execution. In this way, you can even create a chain of service calls, such that one is called only when the response from the previous service call has been received.

For example, suppose you have an event of type"myEventType". When that event is dispatched, you would like to make a service call and also call a method on an object to execute some business logic. When the response of that server call, you want to create an event, perhaps to notify views that the data has been received and also call a method passing the result so that is can save it in the model.

To do that, you would write your sequence as follows:

```
<EventHandlers type="myEventType">
 <WebServiceInvoker instance="{myServiceInstance}" ...>

  <resultHandlers>

   <EventAnnouncer type="myEventTypeTwo"
    generator="EventClassNameToInstantiate" />

   <MethodInvoker generator="WorkerTwo"
    method="receiveResults"
    arguments="{resultObject}" />

  </resultHandlers>
```

```
 </WebServiceInvoker>

 <MethodInvoker generator="WorkerOne"
  method="doSomething />

</EventHandlers>
```

In this scenario, the following will happen:

1. The service call will be made
2. The WorkerOne function doSomething will be called
3. After some time, say 5 seconds, the server returns the results
4. A new event of type "myEventTypeTwo" will be dispatched by the EventBuilder
5. The WorkerTwo function receiveResults will be called with one argument containing the results of the server call.

See Using Smart Objects for more information on the resultObject and fault.

EventAnnouncer

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

EventAnnouncer allows you to dispatch events from an EventHandlers block. When the list of handlers is executed, it will create an event of the class specified in the "generator" attribute. It will then add any properties to the newly created event and dispatch it. You can pass properties to the event that come from a variety of sources, such as the original event that triggered the list execution, a server result object, or any other value.

Example:

```
<EventAnnouncer generator="MyEventClass" type="myEventType">

   <Properties myProperty="myValue" myProperty2="100"/>

</EventAnnouncer>
```

The above example would be the same as doing the following in ActionScript code:

```
var myEvent:MyEventClass = new MyEventClass("myEventType", true);
myEvent.myProperty = "myValue";
myEvent.myProperty2 = 100;
dispatchEvent(myEvent);
```

The main difference is that in the case of the EventAnnouncer, the event is dispatched whenever the event handlers list is run and in the order specified in the list.

Attributes

generator

required

The generator attributes specifies what class of event should be instantiated and dispatched. If this attribute is not specified, then a DynamicEvent will be generated.

Suppose you have a custom event called "MyEvent" in the package com.yourdomain.events. You can specify a complete path to com.yourdomain.events.MyEvent:

```
<EventAnnouncer
   generator="com.yourdomain.events.MyEvent"
   type="myEventType" />
```

Generally, you may want to use a binding to specify the class name. Assuming you have an import statement like this in your Event Map:

```
import com.yourdomain.events.MyEvent;
```

or simply:

```
import com.yourdomain.events.*;
```

You can then declare your event class using bindings:

```
<EventAnnouncer
 generator="{MyEvent}"
 type="myEventType" />
```

The advantage of using this syntax is that if you use Flex Builder, it will allow you to navigate to your MyEvent.as file by pressing Command and clicking on the class name (MyEvent) inside the generator attribute.

type

required

The type attribute specifies the event type you want to dispatch. Suppose you have an event class definition as follows:

```
public class MyEvent extends Event {
 public static const MY_EVENT_TYPE:String = "myEventType";
 public var myProperty:String;
 public var myProperty2:uint;
 }
```

You can then specify the type attribute with the event type literal string:

```
<EventAnnouncer
 generator="{MyEvent}"
 type="myEventType" />
```

Or you can use the binding syntax again:

```
<EventAnnouncer
```

```
 generator="{MyEvent}"
 type="{MyEvent.MY_EVENT_TYPE}" />
```

This will allow the compiler to check that the type you specified exists.

constructorArgs

Object or Array

If your event has a constructor signature that differs from the default Flash Event constructor, you need to specify the parameters with this attribute.

bubbles

Although you can specify the event's bubbles property, whether you set it to true or false will have little effect, as the event will be dispatched from the Mate Dispatcher itself (the Application by default).

cancelable

Boolean

Default value: true

Indicates whether the behavior associated with the event can be prevented.

Inner tags

Properties

You can add properties to your event by using the SmartProperty tag inside the EventAnnouncer tag. The properties to set in your event must be public.

As attributes of the Properties tag, you can specify the names of your properties and set the values of those properties by setting the value of those attributes.

Suppose you are creating an event of the class shown previously (MyEvent) and assigning values for its two properties (myProperty and myProperty2):

```
<EventAnnouncer generator="{MyEvent}" type="{MyEvent.MY_EVENT_TYPE}">
 <Properties
  myProperty="myValue"
  myProperty2="100" />
</EventAnnouncer>
```

Besides specifying literal values, you can assign values coming from the event that triggered the event handlers list execution:

```
<EventAnnouncer generator="{MyEvent}" type="{MyEvent.MY_EVENT_TYPE}">
 <Properties
  myProperty="{event.userName}"
  myProperty2="{event.age}" />
```

```
</EventAnnouncer>
```

This assumes that the original event contained a userName property and an age property.

If this event announcer tag is inside a <resultHandlers> block that originated from any service tag (RemoteObjectInvoker , WebServiceInvoker , HTTPServiceInvoker), you can also pass values coming from the server result:

```
<RemoteObjectInvoker .....service attributes....>
 <resultHandlers>
  <EventAnnouncer generator="{MyEvent}"
   type="{MyEvent.MY_EVENT_TYPE}">

   <Properties
    myProperty="{resultObject.userName}"
    myProperty2="{resultObject.age}" />
  </EventAnnouncer>

 </resultHandlers>
</RemoteObjectInvoker>
```

This assumes the server returned an object that contained a userName property and an age property.

If this event builder tag is inside an <faultSequence> tag that originated from a Service builder tag, you can also pass values coming from the server fault. This comes in handy when you want to handle possible server errors.

```
<RemoteObjectInvoker .....service attributes....>
 <faultHandlers>
  <EventAnnouncer generator="{MyEvent}"
   type="{MyEvent.MY_EVENT_TYPE}">

   <Properties
    myProperty="{fault.faultDetail}"
    myProperty2="{fault.errorID}" />
  </EventAnnouncer>

 </faultHandlers>
</RemoteObjectInvoker>
```

You can also use the complete resultObject, fault or data as a property depending on what your event properties are.

For example, if the server returned a string, you would be able to set the event property "myProperty" with the server result:

```
<RemoteObjectInvoker .....service attributes....>
 <resultHandlers>
  <EventAnnouncer generator="{MyEvent}"
   type="{MyEvent.MY_EVENT_TYPE}">

   <Properties
    myProperty="{resultObject}"
  </EventAnnouncer>
```

```
 </resultHandlers>
</RemoteObjectInvoker>
```

You can also use the service fault or the sequence data.

```
<RemoteObjectInvoker .....service attributes....>
 <faultHandlers>
  <EventAnnouncer generator="{MyEvent}"
   type="{MyEvent.MY_EVENT_TYPE}">

   <Properties
    myProperty="{fault}"
  </EventAnnouncer>

 </faultHandlers>
</RemoteObjectInvoker>


<EventAnnouncer generator="{MyEvent}" type="{MyEvent.MY_EVENT_TYPE}">

 <Properties myProperty="{data}"
</EventAnnouncer>
```

Dispatcher

The **Dispatcher** can be used to dispatch an event from anywhere in your application. It can be used as a tag within your views and can be instantiated in ActionScript classes. Although views can dispatch events using their "dispatchEvent()" method, if these views as used within Popup windows, the event will not be received by other views or the Event Map. By using the Dispatcher, we can guarantee that the event will be received by all registered listeners.

Example:

```
<mate:Dispatcher generator="MyEventClass" type="myEventType">

   <mate:eventProperties>
      <mate:EventProperties myProperty="myValue" />
   </mate:eventProperties>

</mate:Dispatcher>
```

Using dispatcher as a tag

Using the dispatcher tag allows you to dispatch an event from any MXML component. You can use the dispatcher attributes to make the dispatcher create the event for you or you can create an event and use the dispatcher only to dispatch the already created event.

It also allows you to receive direct responses such that only the object that dispatched the event receives this response.

Attributes

generator

The generator attributes specifies what class of event should be instantiated and dispatched.

Suppose you have a custom event called "MyEvent" in the package com.yourdomain.events. You can specify a complete path to com.yourdomain.events.MyEvent:

```
<mate:Dispatcher

        generator="com.yourdomain.events.MyEvent"

        type="myEventType" />
```

Generally, you may want to use a binding to specify the class name. Assuming you have an import statement like this in your Event Map:

```
import com.yourdomain.events.MyEvent;
```

or simply:

```
import com.yourdomain.events.*;
```

You can then declare your event class using bindings:

```
<mate:Dispatcher

        generator="{MyEvent}"

        type="myEventType" />
```

type

String

The type attribute specifies the event type you want to dispatch. Suppose you have an event class definition as follows:

```
public class MyEvent extends Event {

        public static const MY_EVENT_TYPE:String =  "myEventType" ;

        public var myProperty:String;

        public var myProperty2:uint;
```

```
}
```

You can then specify the type attribute with the event type literal string:

```
<mate:Dispatcher

        generator="{MyEvent}"

        type="myEventType" />
```

Or you can use the binding syntax again:

```
<mate:Dispatcher

        generator="{MyEvent}"

        type="{MyEvent.MY_EVENT_TYPE}" />
```

This will allow the compiler to check that the type you specified exists.

Inner tags

eventProperties

You can add properties to your event by using the EventProperties tag inside the eventProperties inner tag. The properties to set in your event must be public.

As attributes of the EventProperties tag, you can specify the names of your properties and set the values of those properties by setting the value of those attributes.

Suppose you are creating an event of the class shown previously (MyEvent) and assigning values for its two properties (myProperty and myProperty2):

```
<mate:Dispatcher

        generator="{MyEvent}"

        type="{MyEvent.MY_EVENT_TYPE}">

        <mate:eventProperties>

                <mate:EventProperties

                        myProperty="myValue"
```

```
                    myProperty2="100" />

        </mate:eventProperties>

</mate:Dispatcher>
```

ResponseHandler

The ResponseHandler tag can be used to receive a response from an event that was dispatched from this dispatcher instance.

See Receiving a response to a dispatched event.

ServiceResponseHandler

The ServiceResponseHandler tag can be used to receive a service response from an event that was dispatched from this dispatcher instance.

See Receiving a response to a dispatched event.

Receiving a response to a dispatched event

After dispatching an event using the Dispatcher tag, the view that dispatched this event can receive a response. Those responses are sent from the EventMap within the EventHandlers block that was listening for the event dispatched by the Dispatcher.

It's important to note that this response will be received only by the view instance that dispatched the event, even if there are other instances of the same view or other views dispatch the same event.

Using the ResponseHandler tag

```
<mate:ResponseHandler type="searchCustomerResult"
response="onSearchResultReceived(event.result)" />
```

Using the ServiceResponseHandler tag

A simple way to receive a response generated by a service call. It contains result and fault handlers you can use as you would when receiving a normal service (ie: RemoteObject) result or fault. It also contains a response handler that can be used for any situation.

```
<mate:ServiceResponseHandler
   result="trace(event.result)"
   fault="trace(event.fault.faultString)"
   response="trace(event.data)" />
```

This tag listens for responses sent by ServiceReponse tags created in the EventHandlers block that was listening for the event dispatched by the Dispatcher. The use of this tag has no effect if its matching counterpart is not

included in the EventMap . ServiceResponseAnnouncer tags must be included in a resultHandlers and in a faultHandlers.

Different ways of dispatching an event

Suppose you want to create and dispatch a MyEvent event of type MyEvent.MY_EVENT_TYPE with properties myProperty="myValue" and myProperty2=100

Creating the event and using the Dispatcher tag to dispatch it

In this scenario, you have a simple Dispatcher tag:

```
<mate:Dispatcher id="myDispatcher" />
```

And you create the event as usual:

```
var myEvent:MyEvent = new MyEvent(MyEvent.MY_EVENT_TYPE);
myEvent.myProperty = "myValue";
myEvent.myProperty2 = 100;
```

Then you use the dispatcher to dispatch it:

myDispatcher.dispatchEvent(myEvent);

Using the default event type and class and setting properties at dispatch time

You can set the event type that you will be dispatching and the class to instantiate as your event in the Dispatcher tag:

```
<mate:Dispatcher

        generator="{MyEvent}"

        type="{MyEvent.MY_EVENT_TYPE}">
```

The Dispatcher will create an event of the given class and type, so it is not necessary to manually create it. In order to dispatch it, you use the dispatcher's generateEvent() function:

```
myDispatcher.generateEvent();
```

If you need to add properties to the event, as in the original example, you can add those properties at the moment of dispatching the event in the form of an object:

```
myDispatcher.generateEvent({myProperty:   "myValue", myProperty2: 100});
```

Creating an event with properties set in the Dispatcher tag

The dispatcher tag allows you to define all properties of an event within the Dispatcher tag itself. You can assign the event type, the event class to instantiate and the properties that the event will contain:

```
<mate:Dispatcher

        id="myDispatcher"

        generator="{MyEvent}"

        type="{MyEvent.MY_EVENT_TYPE}">

        <mate:eventProperties>

                <mate:EventProperties

                        myProperty="myValue"

                        myProperty2="100" />

        </mate:eventProperties>

</mate:Dispatcher>
```

You then dispatch the event at any time, for example, when the user clicks on a button:

```
<mx:Button click="myDispatcher.generateEvent()" />
```

On the button click, the event with the properties set in the Dispatcher tag will be dispatched.

Using dispatcher in ActionScript

When you have a class that is not a visual component, it is not possible to make it dispatch an event unless you extend it from EventDispatcher or make it implement the IEventDispatcher interface.

The dispatcher class allows you to dispatch events in non-visual components or any other ActionScript class.

You only need to instantiate an instance of Dispatcher, create the event and use the instance to dispatch it:

```
var myDispatcher:Dispatcher =new Dispatcher();

var myEvent:MyEvent =new MyEvent(MyEvent.MY_EVENT_TYPE);

myEvent.myProperty ="myValue";

myEvent.myProperty2 = 100;
```

Then you use the dispatcher to dispatch it:

```
myDispatcher.dispatchEvent(myEvent);
```

Injectors

(This tag must be placed inside an < EventMap > tag)

An Injectors tag defined in the Event Map is a container for InjectorPropertys that will inject properties coming from a source to a target, but they can also be used for other purposes.

```
<Injectors target="{MyTargetClass}">
    ...
</Injectors>
```

The Injectors tag are similar to the EventHandlers , and when executed, they will run all inner tags in order. EventHandlers are executed when an event of the matching type is dispatched. Injectors, on the other hand, are executed when an object of the class defined in the target attribute is created. In order for the Injectors to run, the object of that class needs to be created within the display list, or be instantiated by the ObjectBuilder tag with attribute registerTarget="true".

Attributes

target

Class
required

The class that, when instantiated, should trigger the inner tags to run. This target must be supplied using bindings, but this binding is only executed once, when the event map is created.

debug

Boolean

Whether to show debugging information for this Injectors block. If true, Console output will show debugging information as the injectors are applied.

start

Event handler

In the start attribute you can supply an event handler for the event of type ActionListEvent.START. This event is dispatched right before the actions (normally PropertyInjector items) are called, when the list starts execution.

Example:

```
<Injectors target="{MyTargetClass}" start="trace('Execution of injectors list
started!')">
...
</Injectors>
```

end

Event handler

In the end attribute you can supply an event handler for the event of type ActionListEvent.END. This event is dispatched right after all the items in the list have been executed.

Example:

```
<Injectors target="{MyTargetClass}" end="trace('Execution of injectors list ended!')">
...
</Injectors>
```

Inner tags

PropertyInjector

PropertyInjector s defined in the Injectors tag will apply or bind properties specified in the source to the target specified in the parent Injectors target. See more information in the PropertyInjector documentation

Other tags

All other tags allowed inside EventHandlers and MessageHandlers are allowed inside the Injectors. This allows you to inject other properties to the target, such as data coming from a server call (one-time only).

PropertyInjector

(This tag must be placed inside an <Injectors> tag)

An Injectors tag defined in the Event Map is a container for InjectorPropertys that will inject properties coming from a source to a target.

```
<Injectors target="{MyTargetClass}">
   <PropertyInjector targetKey="propertyToPopulate" source="{MyModel}"
sourceKey="propertyFromMyModel" />
</Injectors>
```

Attributes

targetKey

String

required

The property of the target (defined in the parent Injectors tag) that needs to be injected.

source

Class or Object

The object that will be the source that will populate the target property. If this is an object, it will used as is. If this is a class, then an already instantiated object will be retrieved from the cache (instantiated by MethodInvoker or ObjectBuilder with cache=true). If cache does not contain an object of this class, a new object will be created and added to the cache.

sourceKey

String

Property of the source that will be used to populate the target property. Mate will attempt to bind this property to the target property so that whenever the property on the source changes its value, the target will be updated with the new value.

If this attribute is not supplied, then the source object will be injected into the target (instead of a property of the source)

Injecting views

You can inject values to properties declared in views. If you use the default settings in the tag InjectionSettings, then you only need to create a property in your view. Then you need a source that will store the value the view will get. For example, if you have a UserManager class that stores the current user, and you want to show information about that user in a view, you can do the following:

1. Have a view called UserInformation
2. In that view, have a property called "user".
3. In your UserManager class, have a public bindable property called currentUser (this property can be read-only by the use of getters).
4. Inject this property into the view.

```
<Injectors target="{UserInformation}">
   <PropertyInjector targetKey="user" source="{UserManager}" sourceKey="currentUser" />
</Injectors>
```

When the view is instantiated, the property user will be injected into it with the value coming from the UserManager. If the property on the UserManager is bindable, the view will get the latest value any time it changes.

The source can be any "model" that stores data. We use "Managers", but the data could also be stored in a

ModelLocator. For example:

```
<Injectors target="{UserInformation}">
   <PropertyInjector targetKey="user" source="{ModelLocator.getInstance()}"
sourceKey="currentUser" />
</Injectors>
```

Our source will be an already instantiated object (the instance of the ModelLocator), and then we'll bind the property currentUser to the target property.

Injecting view adapters

You can also inject other objects into your views that are not data coming from a model.

As an example, you can inject a "view adapter" into your view. You would then send data to the adapter, which the adapter would format, massage and filter specifically for the view. Data would be send to the adapter also via injection.

```
<Injectors target="{UserInformation}">
   <!-- create the adapter -->
   <ObjectBuilder generator="{UserInformationAdapter}" registerTarget="true" />
   <!-- inject the adapter into the view -->
   <PropertyInjector targetKey="myAdapter" source="{lastReturn}" />
</Injectors>
```

In the code above, the view will contain a public property called "myAdapter". Then the view can use it to access the adapter properties to populate its controls.

In order to populate the adapter with the data, we can use injectors just like we did in the model-to-view example, but instead of injecting the view, we'll inject the adapter:

```
<Injectors target="{UserInformationAdapter}">
   <PropertyInjector targetKey="user" source="{UserManager}" sourceKey="currentUser" />
</Injectors>
```

DataCopier

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

The DataCopier tag allows you to save values into some object. A possible storage is the "data" object available while the sequence is running.

```
<DataCopier destination="data" destinationKey="someProperty" source="result"
sourceKey="someProperty" />
```

The DataCopier tags is a handy tag to quickly copy values from a source into some storage. You can use the event handlers scope "data" as a temporary storage from where handlers that follow in the list can read values. You can also use some other external variable as the storage.

Attributes

source

The source attribute specifies where to get the data to copy from. It can be one of this options:

- event
- data
- result
- fault
- lastReturn
- message
- scope
- currentEvent (maybe the same or different from the event if the tag is placed inside a resultHandlers block, faultHandlers block or any handlers block other than EventHandlers and MessageHandlers.

or another object. If you wish to specify another object, you need to use bindings:

```
<DataCopier source="{myModel}" ... />
```

sourceKey

If you need a property from the source instead of the source itself, you need to specify this attribute.

destination

The destination attribute specifies where to place the data. It can be one of this options:

- event
- data
- result

or another object.

```
<DataCopier destination="{myModel}" ... />
```

destinationKey

If you want to set the value of a property of the destination object, instead of the destination itself, you need to specify this attribute.

Listener

Listener allows you to register a view as a listener for an event type. As long as the event bubbles up or is dispatched via the mate: **Dispatcher** tag or class, the registered listeners will be notified.

Example:

```
<mate:Listener type="myEventType" method="handleThisEvent" />
```

or

```
<mate:Listener type="myEventType" receive="handleThisEvent(event)" />
```

To handle the event received, you can use the *method* attribute or the *receive* attribute. The above examples accomplish exactly the same goal.

If the dispatcher of the event is a child component, we could do the same in ActionScript:

```
addEventListener("myEventType", handleThisEvent);
```

Note, however, that this will only work for events that are bubbling up from children components. The Listener tag, on the other hand, allows you to listen to events dispatched anywhere in your application, even from views contained in PopUp windows. Note: for PopUps, views must use the Dispatcher tag.

Attributes

type

required

The type attributes specifies the type of event for which we would like to register.

Suppose you have an event class definition as follows:

```
public class MyEvent extends Event {

        public static const MY_EVENT_TYPE:String =  "myEventType";

}
```

You can then specify the type attribute with the event type literal string:

```
<mate:Listener type="myEventType" method="handleThisEvent" />
```

Or you can use the binding syntax:

```
<mate:Listener type="{MyEvent.MY_EVENT_TYPE}" method="handleThisEvent" />
```

This will allow the compiler to check that the type you specified exists.

method

either method or receive must be provided

The method attribute specifies the method to call when an event is received. Called method will automatically receive the event.

If you have a listener tag as follows:

```
<mate:Listener type="myEventType" method="handleThisEvent" />
```

You will need to create a method called "handleThisEvent" that will be called when an event of type "myEventType" is received.

```
private function handleThisEvent(event:MyEvent):void {

        // handle the event

}
```

receive

either method or receive must be provided

The receive event handler allows you to handle the event inline. In this attribute you can write simple ActionScript statements.

If, for example, you wanted to change the state of the view when an event is dispatcher, you could do that inline as in the following example:

```
<mate:Listener

        type="myEventType"

        receive="currentState='myOtherState'" />
```

In this inline handler you also have access to the event that was dispatched.

```
<mate:Listener

        type="myEventType"

        receive="trace(event.myProperty)" />
```

ResponseAnnouncer

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

The **ResponseAnnouncer** tag is placed inside an EventHandlers block, so that when an object dispatches an

event, and the list of handlers runs, this tag will allow you to send responses directly to the object that dispatched the event. These responses are actually custom events that you need to create. Using the **ResponseAnnouncer** tag is very similar in form and purpose of the EventAnnouncer tag, with the important difference that when using the EventAnnouncer tag, all listeners of that event will be notified, whereas when using the **ResponseAnnouncer** tag, only the object that dispatched the original event will be notified.

Example:

```
<EventHandlers type="myEventType">

   <ResponseAnnouncer generator="MyEventClass" type="myEventType">

      <Properties myProperty="myValue" myProperty2="100"/>

   </ResponseAnnouncer>

</EventHandlers>
```

The use of this tag will have no effect if the original event was not dispatched using the **Dispatcher** tag. Moreover, this tag will have no effect if no **ResponseHandler** tag was added as an inner tag to the Dispatcher tag.

For a more in-depth description of attributes and inner tags, see EventAnnouncer.

Attributes

type

required

The type of the event to generate as the response. This type is the one that the <ResponseHandler> tag will listen to. See <ResponseHandler> for more information.

generator

The event class to instantiate. If this attribute is not specified a DynamicEvent will be generated.

constructorArguments

Array

If your event has a constructor signature that differs from the default Flash Event constructor, you need to specify the parameters with this attribute.

Using a custom event

Using a custom event gives the most control over what you send as a response. You can set any properties on

that event with data coming from various places such as the result of a server call. When the ResponseHandler gets this response in the form of a custom event, its handler will know what properties to look for in this event.

To send a custom event as a response, you specify the generator attribute to point to your custom event class. Both the class and event type can be specified using bindings:

```
<ResponseAnnouncer generator="{MyEventClass}" type="{MyEventClass.MY_EVENT_TYPE}">

   <Properties myProperty="myValue" myProperty2="100"/>

</ResponseAnnouncer>
```

Using a DynamicEvent

If you don't want to create a custom event only to receive a response, you can use a DynamicEvent. You do so by simply not specifying any generator. You must still supply the event type, which can be a simple string. You just need to ensure that the event type matches exactly that one specified by the < ResponseHandler > in the view. You can set any property to this dynamic event. In the view, you will receive this dynamic event, and while you will be able to get to the properties you set, you won't be able to get compiler errors if you make mistakes in the names of the properties you are trying to access.

```
<ResponseAnnouncer type="myEventType">

   <Properties myProperty="myValue" myProperty2="100"/>

</ResponseAnnouncer>
```

InlineInvoker

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

InlineInvoker allows you to call a method without having to instantiate an object (compare with MethodInvoker). You only need to provide a method to call, which can be a method created in the event map (within a Script block), a static method on a class, or method provided by an already created object or singleton.

Any value returned by the function will be available in the lastReturn. See Using lastReturn and Using Smart Objects (lastReturn)

Example:

```
<InlineInvoker
   method="methodToExecute"
   arguments="{['argument1', 'argument2']}"/>
```

Attributes

method

required

The method attribute specifies what function to call. If you had a function called doWork(), you would then write:

```
<InlineInvoker method="doWork" />
```

arguments

If the function in your class has arguments, you can pass them via the "arguments" attribute. Suppose your doWork function has the following signature:

```
public function doWork(name:String, value:Number)
```

then you can pass those arguments as follows:

```
<InlineInvoker
   method="doWork"
   arguments="{['Tom', 36]}"/>
```

Note that the arguments attribute expects an array. See MethodInvoker (arguments section) for more information.

Calling a static method

With the InlineInvoker tag, you can call static methods. For example, you could call the function random() of the Math library:

```
<InlineInvoker method="Math.random" />
```

Remember that any value returned by the function will be available in the lastReturn (Using lastReturn).

Calling a local method

If you have created a method in your event map, you can call it with the InlineInvoker tag:

```
<mx:Script>
private function myMethod():void {
   trace('you called me!');
}
]]>
</mx:Script>

<EventHandlers type="myEventType">
   <InlineInvoker method="myMethod" />
</EventHandlers>
```

Calling a method on a singleton class

Suppose you have a singleton class that contains the method "myMethod" and that you access the singleton instance by the getInstance() method. Then you could call your the method by:

```
<InlineInvoker method="{MySingleton.getInstance().myMethod}" />
```

ObjectBuilder

(This tag must be placed inside an < EventHandlers > tag or < MessageHandlers > tag)

When placed inside an EventHandlers block, and the list of handlers is executed, it will create an object of the class specified in the "generator" attribute. You can pass arguments to the constructor of this class that come from a variety of sources, such as the event itself, a server result object, or any other value. Unless you specify cache="false", this object instance will be "cached" and not instantiated again when using the MethodInvoker or PropertyInjector s

Example:

```
<ObjectBuilder generator="ClassNameToInstantiate"
   constructorArguments="{['argument1','argument2']}" />
```

The above example would be the same as doing the following in ActionScript code:

```
var myObject:ClassNameToInstantiate = new ClassNameToInstantiate('argument1',
'argument2');
```

Attributes

generator

required

The generator attribute specifies what class should be instantiated.
Suppose you have a class called "MyClass" in the package com.yourdomain.business. You can specify a complete path to com.yourdomain.business.MyClass:

```
<ObjectBuilder generator="com.yourdomain.business.MyClass" />
```

Generally you may want to use a binding to specify the class name. Assuming you have an import statement like this in your Event Map:

```
import com.yourdomain.business.MyClass;
```

or simply:

```
import com.yourdomain.business.*;
```

You can then instantiate your worker using bindings:

```
<ObjectBuilder generator="{MyClass}" />
```

The advantage of using this syntax is that if you are using Flex Builder, you can press the command key (Mac) or the Ctrl key (Windows) and click on the generator class (MyWorker in the example) and it will take you to the class definition.

constructorArguments

Array

If the constructor of you class requires arguments, you can pass them via the "constructorArguments" attribute. Suppose your constructor has the following signature:

```
public function MyClass(name:String, value:Number)
```

then instantiating the class as follows will work:

```
<ObjectBuilder generator="{MyClass}"
   constructorArguments="{['Tom', 36]}"/>
```

Note that the constructorArguments attribute expects an array. Besides passing literal values, you can pass values coming from the event that triggered the handlers list:

```
<ObjectBuilder
   generator="{MyClass}"
   arguments="{[event.userName, event.age]}"/>
```

This assumes that the event contained a userName property and an age property.

You can also pass values coming from a service result, fault, or others. See MethodInvoker, arguments attribute for more information.

cache

Boolean

The cache attribute lets you specify whether this newly created object should be kept live so that the next time an instance of this class is requested, this already created object is returned instead. The instance can be requested by a MethodInvoker or a PropertyInjector.

For example, you may want to have a MethodInvoker use an already instantiated instance created by an

ObjectBuilder. Since the default value for this attribute is "true", it will do that by default. On the other hand, if you wanted to have two different instances, then you must set this attribute to "false".

```
<EventHandlers type="myEventType">
   <ObjectBuilder generator="{MyClass}" />
</EventHandlers>

<EventHandlers type="myOtherEventType">
   <MethodInvoker generator="{MyClass}" method="doWork" />
</EventHandlers>
```

Inner tags

Properties

You can add properties to your instantiated object by using the Properties tag inside the ObjectBuilder tag. These properties must be public.

Suppose you are creating an instance of a ShippingCalculator class. This class has a property called weightFactor and flatFee. In order to set those two properties, you can use the <Properties> inner tag. As attributes of the Properties tag, you can specify the names of your properties and set the values of those properties by setting the value of those attributes as follows:

```
<ObjectBuilder generator="{ShippingCalculator}">

   <Properties weightFactor="0.5" flatFee="3" />

</ObjectBuilder>
```

Besides specifying literal values, you can assign values coming from the event that triggered the sequence:

```
<ObjectBuilder generator="{ShippingCalculator}">

   <Properties weightFactor="{event.factor}" flatFee="{event.fee}" />

</ObjectBuilder>
```

This assumes that the original event contained a factor property and a fee property.

Other sources can include service results or faults, values returned by a MethodInvoker, etc.

StopHandlers

(This tag must be placed inside an < EventHandlers > tag or a < MessageHandlers > tag)

Handlers inside an EventHandlers block run all listeners in order. The StopHandlers tag lets you stop execution of the handlers before it reaches the end of the list. The list can be stopped based on whether the "lastReturn" is equal to some value, or based on an external function that tells whether or not the sequence must be stopped.

```
<StopHandlers lastReturnEquals="someValue" />
```

or

```
<StopHandlers stopFunction="myStopSequenceFuntion" />
```

Attributes

lastReturnEquals

either lastReturnEquals or stopFunction must be provided

If there exists a MethodInvoker right before the StopHandlers tag, and the execution of the function called by the MethodInvoker returned a value ("lastReturn"), you can compare it to some other value and stop the subsequent handlers execution if it is equal.

```
<StopHandlers lastReturnEquals="someValue" />
```

Though other tags return values, generally speaking, only MethodInvoker s return values you can use. Normally, that value is nullified after other tags are executed. See Using lastReturn .

stopFunction

either lastReturnEquals or stopFunction must be provided

A more flexible approach than using lastReturnEquals is to use the stopFunction attribute and handle the logic externally. The function that you implement needs to return true if the list execution must stop or false if not.

```
<StopHandlers stopFunction="myStopSequenceFuntion" />
```

Then you implement your evaluation function:

```
private function myStopHandlersFunction(scope:IScope):Boolean {

        ... here you do some evaluation to determine

                whether to stop the sequence or not...

        return false; //or return true;

}
```

eventPropagation

String

Possible values are: noStop, stopPropagation, stopImmediatePropagation (default: stopImmediatePropagation)

This attribute lets you stop the event that triggered the execution of the handlers list (EventHandlers). If there are any listeners for this event other than this list of handlers, they will not be notified if the propagation of the event is stopped. See Flex documentation regarding the difference between stop propagation and stop immediate propagation.

The default of this attribute is to immediate stop propagation of the event ("stopImmediatePropagation").

Using the scope parameter in the stopFunction

The stopFunction function you implement must receive one argument of type IScope. You can use the values in the EventHandlers ' scope to determine whether or not the execution must be stopped.

Implementors of the interface IScope contain these properties and functions:

event : the event that triggered the list execution. Although it is of type flash.events.Event, you can cast it to a custom event.

data: the data object. It is used to store custom information while the list is running.

lastReturn: the returned value from the last handler that run.

isRunning(): a function that returns whether the list execution is currently running.

dispatcher : an event dispatcher that can be use to dispatch events.

currentEvent : the event that triggered the handlers list or inner handlers list, such as resultHandlers or faultHandlers (ResultEvent and FaultEvent respectively) to run .

Depending on the type of handlers list the StopHandlers tag is in, this scope can be of one the Scope subtypes:

1. **MessageScope**
   This type exists when the StopHandlers tag is inside a < MessageHandlers > block as opposed to an <EventHandlers >. It contains properties specific to a "message received" event:

   message : the message received (type mx.messaging.messages.IMessage)

   messageEvent : the original MessageEvent (type mx.messaging.events.MessageEvent)

   fault: if the consumer subscription generated a fault, it will be available in this property
2. **ServiceScope**

This sequence type exists when the StopHandlers tag is inside a <resultHandlers> or <faultHandlers> block that generated from a Service call (either by using WebServiceInvoker, HTTPServiceInvoker or RemoteObjectInvoker tags). It contains data returned by the server.

result: the result object returned by the server when the server did not generated a fault.

fault: the fault (if any) generated by the service call.

resultEvent: the result event generated by the service call. This property is of type mx.rpc.events.ResultEvent

faultEvent: the fault event generated by the service call. This property is of type mx.rpc.events.FaultEvent

Handlers can also be stopped inside a function called by MethodInvoker that either implements the IScopeReceiver interface (see Implementing the IScopeReceiver interface) or by passing the scope as an argument in the function call.

ResponseHandler

(This tag must be placed inside a <Dispatcher> tag)

The **ResponseHandler** tag can be used to receive a response from an event that was dispatched from a dispatcher instance. After dispatching an event using the **Dispatcher** tag, the view that dispatched this event can receive a response. Those responses are sent from the EventMap within the **EventHandlers** block that was listening for the event dispatched by the **Dispatcher**.

It's important to note that this response will be received only by the view instance that dispatched the event, even if there are other instances of the same view or other views dispatch the same event.

```
<mate:Dispatcher>

   <mate:ResponseHandler type="searchCustomerResult"
response="onSearchResultReceived(event.mySearchResult)" />

</mate:Dispatcher>
```

Responses are handled by either the response event handler or by the method attribute. The response event handler allows you to write inline code. If you use the method attribute, you must implement a method that receives an event (either a custom event or a DynamicEvent).

Attributes

type

Type of the event that this ResponseHandler is listening to. This type needs to match the ResponseAnnouncer type defined in the ListenerSequence in the EventMap .

response

either response or method must be specified

Event handler that receives the "response". The event received is triggered by a ResponseAnnouncer created in the EventMap . This event will contain any properties set in the **ResponseAnnouncer** by using the **Properties** tag.

method

either response or method must be specified

Method that will handle the response event. It must accept a custom event (or Event that can be casted within the method) or a DynamicEvent.

ServiceResponseAnnouncer

(This tag must be placed inside an < EventHandlers > tag)

The **ServiceResponseAnnouncer** tag is placed inside an EventHandlers block, so that when an object dispatches an event, and the list of handlers runs, this tag will allow you to send responses directly to the object that dispatched the event.

These responses are 3 predefined events:

- response
- result
- fault

Because of those predefined events, this tag is used inside a <resultHandlers> block or <faultHandlers> block that are generated after server calls. This response will contain the server result when placed inside a resultHandlers, and the server fault when placed inside a faultHandlers. In addition, you can supply an object that will be found in the "data" property of the event or any other custom property by using the <Properties> tag.

Example:

```
<RemoteObjectInvoker ...>

   <!-- result handlers list gets executed when service returns with a result -->
   <resultHandlers>
      <ServiceResponseAnnouncer type="result" />
   </resultHandlers>

   <!-- server fault -->
```

```
   <faultHandlers>
      <ServiceResponseAnnouncer type="fault"/>
   </faultHandlers>

</RemoteObjectInvoker>
```

The use of this tag will have no effect if the original event was not dispatched using the **Dispatcher** tag. Moreover, this tag will have no effect if no **ServiceResponseHandler** tag was added as an inner tag to the Dispatcher tag.

See <**ServiceResponseHandler**> for more information.

Attributes

type

required

The type of the event to generate as the response. This type is the one that the <ServiceResponseHandler> tag will handle. response, result, or fault are valid values.

data

Any object that you wish to send in the data property of the event sent as the response that will be received by the **ServiceResponseHandler** tag.

Inner tags

Properties

You can add properties to the event sent as the response by using the Properties tag inside the ServiceResponseAnnouncer tag. The properties set will be dynamically added to the ResponseEvent that serves as the response.

As attributes of the Properties tag, you can specify the names of your properties and set the values of those properties by setting the value of those attributes.

```
<ServiceResponseAnnouncer type="result">

   <Properties myProperty="{resultObject.property}"/>

</ServiceResponseAnnouncer>
```

ServiceResponseHandler

(This tag must be placed inside a <Dispatcher> tag)

The **ServiceResponseHandler** tag can be used to receive a response from an event that was dispatched from a dispatcher instance. After dispatching an event using the Dispatcher tag, the view that dispatched this event can receive a response. Those responses are sent from the EventMap within the EventHandlers block that was listening for the event dispatched by the Dispatcher.

It's important to note that this response will be received only by the view instance that dispatched the event, even if there are other instances of the same view or other views dispatch the same event.

This tag is a simple way to receive a response generated by a service call. It contains result and fault handlers you can use as you would when receiving a normal service (ie: RemoteObject) result or fault. It also contains a response handler that can be used for any general situation.

```
<mate:Dispatcher>

   <mate:ServiceResponseHandler
      result="trace(event.result)"
      fault="trace(event.fault.faultString)"
      response="trace(event.data)" />

</mate:Dispatcher>
```

Attributes

result

Event handler that receives "result" type of service responses. The event received is triggered by a ServiceResponseAnnouncer with type="result" created in the EventMap. When the ServiceResponseAnnouncer tag is placed inside a resultHandlers block, this result event will contain the following properties:

- result
- fault (null)
- data (might be null if no data was supplied in the ServiceResponseAnnouncer tag)
- any other properties set in the ServiceResponseAnnouncer by using the **Properties** tag.

fault

Event handler that receives "fault" type of service responses. The event received is triggered by a ServiceResponseAnnouncer with type="fault" created in the EventMap. When the ServiceResponseAnnouncer tag is placed inside a faultHandlers block, this fault event will contain the following properties:

- result (null)
- fault
- data (might be null if no data was supplied in the ServiceResponseAnnouncer tag)

any other properties set in the **ServiceResponseAnnouncer** by using the **Properties** tag.

response

Event handler that receives "response" type of service responses. The event received is triggered by a ServiceResponseAnnouncer with type="response" created in the EventMap. This event will contain the

following properties:

- result (null if ServiceResponseAnnouncer was placed inside a faultHandlers block)
- fault (null if ServiceResponseAnnouncer was placed inside a resultHandlers block)
- data (might be null if no data was supplied in the ServiceResponseAnnouncer tag)

any other properties set in the **ServiceResponseAnnouncer** by using the **Properties** tag.

Debugger

The Debugger tag allows debugging your Mate code (EventHandlers and sub-handlers blocks, MessageHandlers, etc). When using this tag, you specify a level of debugging, which will filter the type of messages you see.

It is recommended that you remove this tag from your code when you are ready to deploy. Its use during production will impact performance if you enabled the debugging in many objects (ie: in many EventHandlers) or if you specified a low debugging level such as ALL or DEBUG.

```
<mate:Debugger level="{Debugger.ALL}" />
```

Attributes

level

int
Provides access to the level the debugger is currently set at. Value values are:

- Debugger.FATAL: designates events that are very harmful and will eventually lead to application failure
- Debugger.ERROR: designates error events that might still allow the application to continue running.
- Debugger.WARN: designates events that could be harmful to the application operation
- Debugger.INFO: designates informational messages that highlight the progress of the application at coarse-grained level.
- Debugger.DEBUG: designates informational level messages that are fine grained and most helpful when debugging an application.
- Debugger.ALL: intended to force a target to process all messages.

Best Practices

Events

A fundamental part of Mate are the events, since all communication between the different parts of the application is made via events.

The EventHandlers in the EventMap subscribe to listen to events of particular types. The type specified is very important because it will determine whether or not a sequence must be run. This type is a string and that string must be unique throughout your whole application.

We define this type as a constant in the event itself. Suppose you have an event called CustomerEvent:

```
import flash.events.Event;

public  class CustomerEvent  extends Event {

            public  function  CustomerEvent(type:String, bubbles:Boolean=  true, cancelable:Boolean= true) {

            super(type, bubbles, cancelable);

            }

}
```

We add a constant for the event type:

```
public  static  const ADD:String =  "addCustomerEvent" ;
```

In our event map, then, we can use this constant as the event type. This helps us not making mistakes when typing the event type.

```
<EventHandlers type="{CustomerEvent.ADD}">
    .....
</EventHandlers>
```

You can use the same in your view Listeners:

```
<mate:Listener type="{CustomerEvent.ADD}" ..../>
```

And in your dispatchers:

```
<mate:Dispatcher generator="{CustomerEvent}" type="{CustomerEvent.ADD}" ...>
```

Creating a unique type

Since the event type must be unique, you can append the name of your event class so that it will be less likely to conflict with other events:

"add" + "CustomerEvent"= "addCustomerEvent"

```
public  static const ADD:String = "addCustomerEvent";
```

If you want to ensure there are no conflicts, you can also use the full package name as part of the event type:

```
public  static const ADD:String = "com.mydomain.events.CustomerEvent.ADD";
```

The above approach is not recommended if this event is a view event declared by the Event metatag in the view, as it will make the type too large and unreadable in parent views that wish to add event handlers for that event.

Having several types in the same event

You can have more than one type in the same event. As long as all those types defined use and need the same properties, they can be put together. For example, adding, updating and removing a customer all need a customer property and it makes sense to create only one event for all those types (assume you have a class Customer):

```
public class CustomerEvent extends Event {

        public static const ADD:String = "addCustomerEvent" ;

        public static const UPDATE:String = "updateCustomerEvent" ;

        public static const DELETE:String = "deleteCustomerEvent" ;

        public var customer:Customer;

        public function CustomerEvent(type:String, bubbles:Boolean= false, cancelable:Boolean= true) {

        super(type, bubbles, cancelable);

        }

}
```

Bubbling property

When an event has it "bubbles" property set to false, only listeners added explicitly to the object that dispatched the event will be notified when the event is dispatched.

If you use the < Dispatcher > tag in your views or the Mate Dispatcher class in your other classes and PopUps, whether you set the event to bubble up or not will not make much difference, since the EventMap and other views registered will still be notified of the event.

However, if you use the view's dispatchEvent() function (every Display Object is a Dispatcher) and dispatch the event as you would normally do:

```
var myEvent:CustomerEvent = new CustomerEvent(CustomerEvent.ADD);
dispatchEvent(myEvent);
```

the aforementioned rules will apply and other views and the event map will not be notified unless you set the

event property bubbles to true. Just so that you don't have to remember this every time you create an event:

```
new CustomerEvent(CustomerEvent.ADD, true)
```

you can set this property as true by default in your constructor:

```
public function CustomerEvent(type:String, bubbles:Boolean= true, cancelable:Boolean= true)
```

and then this code will work fine:

```
var myEvent:CustomerEvent = new CustomerEvent(CustomerEvent.ADD);
dispatchEvent(myEvent);
```

Built-in Events

An EventHandlers block can listen to any event that bubbles up or that has been dispatched by using the Mate Dispatcher. Any Flex built-in event can also be used as long as it bubbles up. For example, an EventHandlers block can listen to FlexEvent.APPLICATION_COMPLETE event.

dispatchEvent() vs Dispatcher tag

If you only need to dispatch an event, in most cases, you can simply use dispatchEvent() function that is available in every display object (ie: views). In those cases, you would simply instantiate an event, and then dispatch it using the dispatchEvent(myEvent) function. For example:

```
var myEvent:CustomerEvent = new CustomerEvent(CustomerEvent.ADD);
dispatchEvent(myEvent);
```

If you need additional functionality, particularly when you need to receive responses after you dispatched the event, you will need to use the Dispatcher tag. See ResponseHandler, ServiceResponseHandler, ResponseAnnouncer, and ServiceResponseAnnouncer.

Unless you need that additional functionality provided by the Dispatcher tag, it is recommended to use dispatchEvent() as a way of decoupling your application from Mate.

The Event Map

The Event Map should be placed in an mxml file by itself and added to the Application file. Typically, an application will create many event types, so it is recommended to have several Event Maps, grouping related event types in each. For example, you may have an event map that handles all Login-related events (login check, logout, forgot password, login success, etc) and a different event map for all the event that relate to updating a user's information (user add, user delete, user update, etc).

The component that defines the event map should extend from EventMap.

Also see Where should the Event Map go?

Generators

For every generator attribute, you can use a shorter way of specifying the class name. Say you have a class called com.mydomain.MyWorker that you want to instantiate in an EventHandlers block. Instead of specifying the class name as a string: *generator="com.mydomain.MyWorker"*, you can use a binding:

```
<MethodInvoker generator="{MyWorker}" ..... />
```

Note that you also need to have an *import com.mydomain.** statement at the top of your EventMap.

This allows you to jump to the class if you are using Flex Builder. You can press the command key (Mac) or the Ctrl key (Windows) and click on the generator class (MyWorker in the example) and it will take you to the class definition.

It is important to note that this binding will be executed only once because we are binding to class, which doesn't change during the life of the application.

Service Invokers

Service Invokers (WebServiceInvoker, HTTPServiceInvoker, and RemoteObjectInvoker) let you create a service instance in the event sequence. This works great for small projects, prototyping and "I just need this service call" type of projects. As a best practice, however, all services should be defined in a separate file. This file can be called "Services.mxml" and be placed in its own services folder or in the business folder. In order to call one of those re-defined services, you can use the same tags (WebServiceInvoker, HTTPServiceInvoker, and RemoteObjectInvoker), but instead of defining all the specific properties (WSDL address, endpoint, etc), you simply use the *instance* attribute, specifying what already created service instance to use.

Using the *instance* attribute allows you to reuse services instead of defining them in your event handlers. You can create an instance of the services in your event map, and then reference it anywhere in your map:

```
[Bindable] private var services:Services = new Services();
```

```
<RemoteObjectInvoker instance="{services.myService}" ... >
```

In order for this work, all services (RemoteObject, HTTPService or WebService) in your Services.mxml file must have an id assigned to them. You will then use that id when assigning an instance for your RemoteObjectInvoker, HTTPServiceInvoker or WebServiceInvoker tags.

For the above example, your Services.mxml file must contain a RemoteObject tag:

```
<mx:RemoteObject id="myService" ... />
```

No result and fault handlers are needed for this tag.

A sample Services.mxml file:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Object xmlns:mx="http://www.adobe.com/2006/mxml">

   <mx:RemoteObject id="myService" destination="ColdFusion" source="someCFC" />

</mx:Object>
```

As a middle-ground solution, you can also specify the service within your event map and reference to it by specifying the instance within the invoker tag.

For example, in your event map, you will have these two set of tags:

```
<mx:RemoteObject id="myService" ... />

<EventHandlers type="myEventType">
   <RemoteObjectInvoker instance="{myService}" ... >
</EventHandlers>
```

Modules

Mate works with Flex Modules. Depending on your needs, you can have an EventMap within the main application and other event maps included within the modules themselves.

Regardless of where the event maps are placed, however, events dispatched anywhere (in any module) will trigger EventHandlers listening for those event types in all event maps (those included in the main application plus those event maps included in modules that have been loaded).

It is recommended that any event that represents some action that can be shared among modules be handled in an EventMap placed in the main application. In that way, your modules can dispatch events knowing that they will be handled by a shared event map. Any event that is specific to a module, should be placed in an event map that is placed in the module's main file. When the module dispatches this specific event, its own event map will handle it.

Under this configuration, you would have:

- Main application
- MainEventMap.mxml
- Module A
- ModuleAEventMap.mxml

Lastly, there could be events handled by both the main map and the module's map. If you wish to specify the order by which each event map will handle the event, you can use the priority attribute of the EventHandlers tag.

How to

Tutorials and how-to guides.

Extending Mate

In the weather widget example , we use the Yahoo! Astra Web API that contains utilities classes that make it simpler to retrieve weather data from Yahoo! Weather. This API has a class called **WeatherService.** To retrieve weather information, you simply write the following code:

```
var weatherService:WeatherService = new WeatherService();
weatherService.getWeather(location, unit);
```

When the weather service returns with the information, it dispatches an event:

WeatherResultEvent.WEATHER_LOADED

When it encounters an error, it dispatches an error event:

WeatherErrorEvent.INVALID_LOCATION

But we would like to be able to handle these two results as a list of handlers in our resultHandlers block by making a call to the weather service and receiving each type of event in its own handlers list, much like we do when calling a WebService.

Pseudo code for this would look like this:

```
<WeatherLoader><!-- calls weather service -->

   <resultHandlers>
      <!-- handle the result here -->
   </resultHandlers>

   <faultHandlers>
      <!-- handle the fault here -->
   </faultHandlers>

</WeatherLoader>
```

Creating this wrapper for the WeatherService class lets us include it in our EventHandlers and MessageHandlers and let us easily handle those results and faults.

The WeatherService also needs the location for which we want to retrieve the weather information and the unit

of measure we want to use (ºC or ºF). To pass that information we can either enter that information in the <WeatherLoader> tag itself or by using the <SmartProperties> inner tag. Because the location information will be changing depending on user interaction (ie: if the user enters his/her zip code in the text input), we should use the <SmartProperties> inner tag because it will allow us to retrieve that data from the event that triggered the call.

The pseudo code for that will look like this:

```
<WeatherLoader location="92614" ><!-- calls weather service -->

   <resultHandlers>
      <!-- handle the result here -->
   </resultHandlers>

   <faultHandlers>
      <!-- handle the fault here -->
   </faultHandlers>

</WeatherLoader>
```

Implementing the tag

Our tag will be called WeatherLoader. We first need to create a class with that name that will extend from AbstractServiceInvoker and will implement IAction. We need to implement IAction to be able to place the tag inside EventHandlers blocks. While it is not required to extend from AbstractServiceInvoker, doing so will provide us with the result and fault inner-handlers.

This class will also need public properties for the data we needed: the location and the unit of measure. Those properties will be populated by the SmartProperties tag.

```
package com.asfusion.weather.mate.extensions
{
 import com.asfusion.mate.sequenceItems.*;
 import com.yahoo.webapis.weather.WeatherService;
 import com.yahoo.webapis.weather.events.*;

 public class WeatherLoader extends AbstractServiceInvoker
 implements IAction
 {
  public var location:String;
  public var unit:String;
  public var url:String = 'http://weather.yahooapis.com/forecastrss';
  // the service that we are wrapping
  private var weatherService:WeatherService ;
 }
}
```

The constructor

In the class constructor we'll instantiate the weather service we are wrapping. We'll also set the property "currentInstance" to point to "this". currentInstance is defined in the class AbstractAction. It is the object that will receive the property values specified in the <SmartProperties> tag. Because our class has the location and

unit properties, we'll set the currentInstance to our own class instance.

```
public function WeatherLoader() {

        weatherService =    new  WeatherService();

        currentInstance =    this;
}
```

The run method

Every handler needs a "run" method that gets called when the handlers list is executing each of its tags. We'll override this method to do what we want our class to do when our tag is called. In this method we will make the call to the WeatherService. In this method will also create the inner handlers for the result and fault.

Inner handlers blocks are created when an event is dispatched. In the case of a RemoteObject, for example, the RemoteObject dispatches the ResultEvent when the result gets back. When that event is dispatched, the tags contained in resultHandlers tag are executed. Those events can contain data. The ResultEvent contains a result object with the contents of what the server returned. In our resultHandlers, we access that object by using the {resultObject} smart object. But we could also access that event directly by using the {currentEvent} smart object.

Because we are now creating a custom tag for our own events, we need to specify what event will trigger the resultHandlers execution and what event will trigger the faultHandlers execution. We also need to specify who is dispatching those events.

The WeatherService class dispatches aWeatherResultEvent.WEATHER_LOADED when the result gets back, and a WeatherErrorEvent.INVALID_LOCATION when there is an error. We'll use those two events to trigger our result and fault inner handlers executions. We do so by creating those inner handlers as follows:

```
this.createInnerHandlers(scope, WeatherResultEvent.WEATHER_LOADED, resultHandlers);
```

and

```
this.createInnerHandlers(scope, WeatherErrorEvent.INVALID_LOCATION , faultHandlers);
```

The method createInnerHandlers is defined by theAbtractServiceInvoker class. The scope argument contains the scope within which this tag is called (the scope is defined by the EventHandler tag that contains this action item). The third argument is the inner handlers block that should be started when the event is received. The names "resultHandlers" and "faultHandlers" are defined by the AbstractServiceInvoker class, but you could create your own.

At the end of the method, we make the call to the WeatherService:

```
weatherService.getWeather(location, unit);
```

The complete code for the run method would look like this:

```
override  protected function run(scope:IScope):  void  {
// specify that the dispatcher of the result and error event is the weatherService object
      innerHandlersDispatcher = weatherService;
        if (this.resultHandlers && resultHandlers.length > 0){
           this.createInnerHandlers(scope, WeatherResultEvent.WEATHER_LOADED, resultHandlers);
        }
        if (this.faultHandlers && faultHandlers.length > 0){
           this.createInnerHandlers(scope, WeatherErrorEvent.INVALID_LOCATION , faultHandlers);
        }
        weatherService.getWeather(location, unit);
}
```

Using the tag

We can include our tag in an EventHandlers list, getting the location and unit information from the event that was dispatched. When we receive the weather result, we call WeatherManager.setWeather() method. This method receives the Weather object that was retrieved by the service. The event that triggers the resultHandlers is of type WeatherResultEvent.WEATHER_LOADED as we specified in our run method of our custom tag class. This event contains a Weather object in the "data" property. To access that data, we need to use the SmartObject "currentEvent", which represents the WeatherResultEvent that triggered the inner handlers.

The code:

```
<EventHandlers type="{WeatherEvent.GET}">
   <extensions:WeatherLoader> <!-- make the call to the service -->
      <SmartProperties location="{event.location}" unit="{event.unit}" />
      <extensions:resultSequence>
      <!-- receive the results contained in the currentEvent.data property
(WeatherResultEvent contains a data property) -->
         <MethodInvoker generator="{WeatherManager}" method="setWeather"
arguments="{currentEvent.data}" />
      </extensions:resultSequence>
      <extensions:faultSequence>
```

```
        <!-- receive an error event. The error event contains a data property that
contains the reason for the error -->
        <MethodInvoker generator="{WeatherManager}" method="handleFault"
arguments="{currentEvent.data}" />

      </extensions:faultSequence>

   </extensions:WeatherLoader>

</EventHandlers>
```

Diagrams

One-way communication: from views to business logic

Two-way communication: Dispatcher and ResponseHandler tags

One-way communication from business logic to views: Listener tag

Two-way communication via model: Using an adapter

Two-way communication via model: Using view injection

Core classes

Actions diagram

PDF versions for download

One-way communication: from views to business logic

Two-way communication: Dispatcher and ResponseHandler tags

One-way communication from business logic to views: Listener tag

Two-way communication via model: Using an adapter

Core classes

Actions diagram