Revised edition of *Flex 3 in Action*

FLEX 4 IN ACTION

Tariq Ahmed
Dan Orlando
WITH John C. Bland II
AND Joel Hooks

SAMPLE CHAPTER

MANNING

Flex 4 in Action

by Tariq Ahmed
and Dan Orlando

with John C. Bland II and Joel Hooks

Chapter 11

brief contents

PART 1 APPLICATION BASICS

PART 3 THE FINISHING TOUCHES..441

11 Events

This chapter covers

- Working with events
- Listening for events
- Dispatching events
- Creating custom events

Events are central to understanding how to work with Flex and how to create compelling, well-designed applications. Events are a powerful tool and one of the main features of the Flash Player.

Events are deceptively simple, but at the same time, they're an extremely powerful communication mechanism for your applications. We've mentioned several times that Flex is built around an event-driven framework. Events are the central nervous system within your applications, with information flowing in and out of them, up and down, and side to side. We don't mean to be melodramatic, but if you don't understand events, Flex becomes a terrible chore and ActionScript 2.0 begins to look sexy again.

Let's begin by introducing the event system and describing its all-important role in a Flex application. Next we explore integrating native and custom events in an application while gaining an understanding of event nuances.

11.1 The event system

When working with web technologies such as ColdFusion, .NET, ASP, PHP, PERL, and Python, information is transmitted on a *request* and *response* basis, or what is called *synchronous requests*. In this model, remote servers wait for requests from clients (users) and then gather and process information relevant to the request. They build the appropriate response and send the data back to the client, at which point the transaction is over—until the client requests the next transaction. Figure 11.1 shows a request from your laptop, over the internet, to the web server (where the site is held), and back to your computer. Between steps 1 and 2 there is latency, or a delay, because of the time it takes to travel from your location to the location of the server. The same goes for step 3 to step 4.

Flash Player employs a different paradigm. Instead of sending out requests and receiving responses, Flash Player makes requests and then listeners patiently wait and *listen* for asynchronous events. When a listener hears an event, it performs the task it was designed to do and then waits again for the next event to occur.

Figure 11.2 shows an example of a profile manager with a form and the resulting information as two separate components. In phase 1, the top graphic, the "Your information" pod registers as a listener for update events from the "Update your information" component. At this point nothing happens until the update event is dispatched, and the rest of the application is still available for interaction. Once the user enters some information and clicks the Update button, an event is dispatched and anything listening for this event will be notified, as shown in phase 2, the bottom graphic. Notice that all of this interaction happens within the application and without returning to the server.

In Flash Player, events are constantly fired in response to a variety of user inputs and system notifications. The main instigator for these events is the user. By clicking a mouse button, moving the mouse, or selecting an item from a drop-down menu, the user is unwittingly setting off events, which trigger the application to respond accordingly.

With Flash Player applications, the difference is in the user-transparent operations that take place in the background. This gives you the ability to fire off tasks asynchronously, in parallel, none of which are dependent on each other, allowing for transparent activities to go on behind the

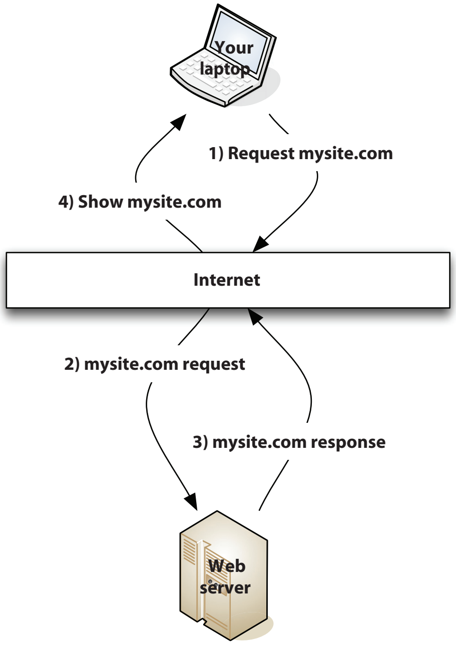


Figure 11.1 Example of a synchronous request

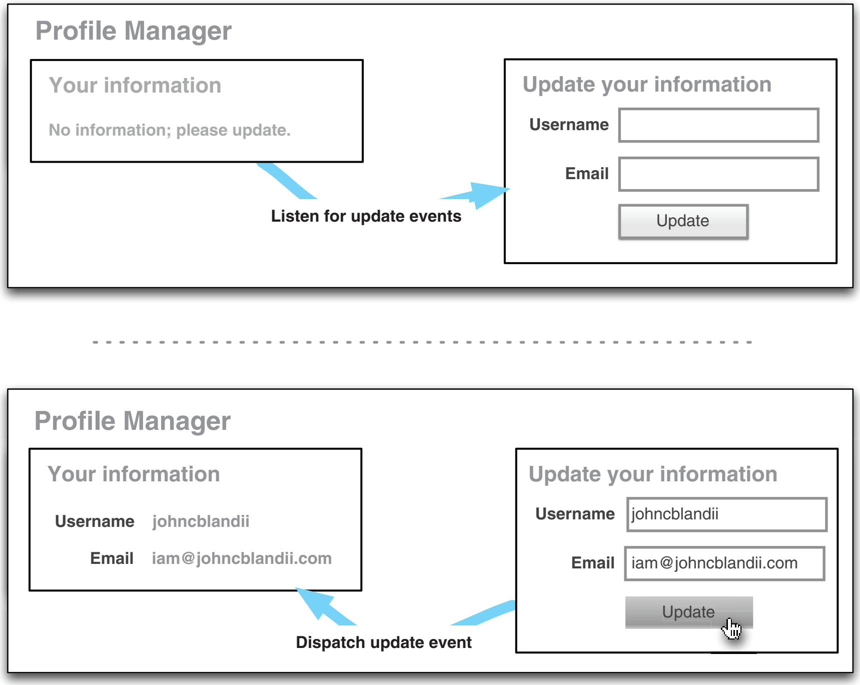


Figure 11.2 Example of event listening

scenes while your application interacts with and provides feedback to the user in real time (versus having code execute synchronously, or sequentially, while the user waits for it to complete).

In addition, when requests for new information are made, they're sent to the target server, and the application carries on with other tasks or patiently waits. The response will be processed whenever the result comes back, whether that happens to be in five seconds or five minutes.

In contrast, when making a request to a web server, an HTML page viewed using a web browser shows the page in a vertical manner from the top of the page to the bottom of the page, unless JavaScript is used to asynchronously update the page after the page has loaded.

We'll get into how this works in a moment; before we do, let's look at how the Flash Player event system is similar to another system you already know well: "I'll call you. Don't call me."

11.1.1 Event system—the Hollywood Principle

Components and classes, also known as dispatchers or targets, need to communicate and pass data to each other through the event system. To illustrate this more clearly, we can draw a parallel between the event system and the Hollywood Principle, which states, "Don't call me. I'll call you."

Think of a person calling you asking to borrow money. You tell them you'll contact them with an answer after checking your bank account. This is the Hollywood Principle, which is nothing more than an asynchronous request, in tech terms. Consider the following example.

Your Flex application is supposed to load data from the server, but while the data loads you want to show some cute, animated message to the user. The data-loading component would be you on the phone, the application would be the person in need, and the cute, animated message would be you dancing to "Footloose" in your mirror while checking your bank balance.

The request has been made. The amount of time between the request and the resulting call back is undetermined, so you could be dancing for a while, but once you've checked your account and have an answer, you're ready to inform the caller of your answer.

This is exactly how Flex applications are developed and why events are critical: Dispatchers receive requests and dispatch events accordingly but not necessarily immediately.

11.1.2 Event-delivery system

It's important to know the pathway events take within a Flex application, because this pathway determines which components receive the events and which don't.

Events originate from the dispatcher, traverse the display tree vertically to the application root and then to the stage, and are sent back down to the dispatcher, as shown in figure 11.3.

The event goes through the parent tree (and any components that are specifically listening to it), which has implications for which components receive notifications about events. For example, as shown in figure 11.3, a component's parent typically receives event notifications; children and siblings don't receive notifications.

From an application perspective, when a component dispatches an event, that event can either *bubble* or not bubble. If the event bubbles, it traverses up the parent

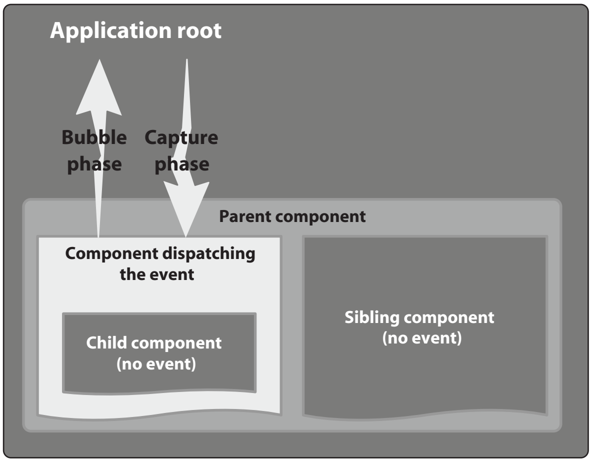

Application root
Bubble phase
Capture phase
Parent component
Component dispatching the event
Child component (no event)
Sibling component (no event)

Figure 11.3 The event flow from component to root and back to component

chain to the application root, passing by every parent in the chain. Each parent can listen for application events at its own level and rebroadcast those events as needed, stop the propagation, or call methods to take specific action.

This is the real power behind the Flash Player event system: the ability to create custom events and pass them around. Tying into the event system allows for maximum decoupling of logic and maximizes the components that can use that logic, which in turn affords maximum code reuse.

Now that you've learned a bit about how the application passes events around, let's break down the event's journey from start to finish by exploring sending and receiving events.

11.2 Sending and receiving events

A Flash Player event is made up of the following core properties (see table 11.1).

Table 11.1 Core event properties

Property	Description
`Event.target`	Event dispatcher.
`Event.currentTarget`	Component currently containing and inspecting the event or the dispatcher.
`Event.type`	A string name that identifies the type of event, such as a click event (clicking a button), a mouse event (moving the mouse), or a select event (selecting an item). Events come in many types, and each type includes unique items, but each event has the generic types mentioned here.
`Event.eventPhase`	Current phase of the event.

> **NOTE** In the Target phase `currentTarget` has the same value as `target`, the dispatcher, but in the Capturing and Bubbling phases `currentTarget` is different.

Each portion of the event journey—from dispatcher to parents to stage and back again—can be divided into phases. Events have only three phases, depending on where they are in the process, as shown in table 11.2.

> **NOTE** The Bubbling and Capturing phases travel through parents but never through the children of the dispatcher.

Table 11.2 The different event phases

Phase	Description
Capturing	Event travels from the stage through the parents to the dispatcher.
Bubbling	Event travels from the dispatcher through the parents to the stage.
Target	Occurs only when the event has reached its target object or the dispatcher and relates only to the one object, target, or dispatcher.

You can determine which phase you're in by using the event object's `eventPhase` property. This property contains an integer that represents one of the following `Event` constants:

- `Event.CAPTURING_PHASE:uint = 1`
- `Event.AT_TARGET:uint = 2`
- `Event.BUBBLING_PHASE:uint = 3`

Best practice

When you're referring to or monitoring these phases, you can use either the number or the constant, although it's considered best practice to use constants wherever possible to make your code easier to read and manage.

Other events and custom classes can contain other properties as well, as you'll see in section 11.3, but at minimum they include these properties.

Let's examine what happens with a simple click event generated by pressing a mouse button, as shown in listing 11.1. Copy and paste listing 11.1 into a new file named mxmlAndScriptBlock.mxml and then run it.

Listing 11.1 Event listening with MXML and an ActionScript event listener

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark">
  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      protected function onClick(event:Event):void{
        Alert.show(event.target.label + " clicked", "Event Test");
      }
    ]]>
  </fx:Script>
  <s:Button label="Button 1" click="onClick(event)" />
</s:Application>
```

When you click the button, a click event is generated and the `onClick` listener is called. In the `onClick` listener, you show an alert. The alert's message is the target item's label, `event.target.label`, with `" clicked"` appended.

NOTE Remember, `event.target` references the dispatcher, so `label` is a property of the target, which in this case is a `Button` component.

Listing 11.1 uses a script block to manage the click event. This can also be done without a script block. Listing 11.2 uses inline MXML to show the same alert.

Listing 11.2 Event listening with inline MXML

```
  <s:Button label="Button 1">
    <s:click>
      <![CDATA[
        Alert.show(event.target.label + " clicked", "Event Test");
      ]]>
    </s:click>
  </s:Button>
```

Listing 11.2 uses the same button but makes the event listening occur inline. This has advantages and disadvantages. Ultimately, it boils down to your preference, but keep in mind that an inline event listener can't be used by multiple dispatchers. In the case where multiple Button components need to call the same method, this approach would fail miserably.

Although these examples do the event adding in MXML, you aren't restricted to working in that environment. You can do the same thing in ActionScript by using the addEventListener() function.

11.2.1 Adding event listeners in ActionScript

Using the addEventListener() function provides more fine-grained control over the events and is the only way in ActionScript to listen for dispatched events. One huge reason for adding event listeners in ActionScript is that event listeners added in MXML can't be removed. We cover this in more detail later, but keep it in mind for now as a big win in the ActionScript-approach column.

If you need to listen for an event in the capture phase, rather than the bubbling and target phases, you must add the listener using the ActionScript method. As a handy reference guide, we've included the main attributes of an event listener in table 11.3.

Let's take the previous MXML application from listing 11.1 and use ActionScript to register the listener. Listing 11.3 shows how you add an event listener on a previously instantiated display object using ActionScript.

Table 11.3 Event listener properties and method arguments

Property	Type	Description
type	String	(Required) The type of event for which to listen. You can define the event type as a String or use best practices and use the event type constant defined on every event object.
listener	Function	(Required) The function to call when the event is dispatched.
useCapture	Boolean	(Optional) The phase in which to listen. If true, the listener listens for the event during the capture phase. The default value is false (uses the bubbling or target phase).
priority	Integer	(Optional) When the listener is called. The higher the number, the sooner it's called. The value can be negative; the default value is 1.
weakReference	Boolean	(Optional) How quickly the event listener object is picked up and destroyed by the garbage collector. true means it's discarded sooner. The default value is false, which prevents garbage collection from destroying the listener (performance at the cost of memory).

Details on using weakReference

weakReference is a hot topic in the ActionScript community pertaining to the proper way of managing application performance and memory. Many developers rely heavily on using weakReference, whereas others rely on removing all listeners. The arguments on both sides are solid, and it all boils down to personal preference. Find out what works best for you and your applications.

Relying on weakReference isn't always ideal. This waits for the garbage collector to pick up the "trash" for you. In most cases, this is fine, but your objects may not disappear instantly as you desire.

Ideally, every event listener you add should be removed before destroying the object. This provides a clean entry and exit for the object without having to play roulette with your memory.

The best practice is to use both.

Listing 11.3 Click event example using ActionScript

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               applicationComplete="init()">
    <fx:Script>
    <![CDATA[
      import mx.controls.Alert;

      protected function init():void{
        button1.addEventListener(MouseEvent.CLICK, onClick);

      protected function onClick(event:Event):void{
        Alert.show(event.target.label + " clicked", "Event Test");
      }
    ]]>
  </fx:Script>
  <s:Button id="button1" label="Button 1" />
</s:Application>
```

Sets onClick to run when click event occurs

As demonstrated in listings 11.1 and 11.3, the main difference between the MXML and ActionScript methods for adding a simple event listener involves the use of addEventListener.

When the button is clicked, it dispatches a click event, which activates and passes the click event to onClick(). With the ActionScript method, you need an intermediary function to add the event listener to the button.

In listing 11.3, this is handled by init(), which is called when the Application issues the applicationComplete event. This event is the last event dispatched in the Application startup and is called when the Application is fully initialized and added to the display list.

NOTE If you add an event listener for the capture phase, passing `true` as the third parameter of `addEventListener`, of a button's click event, it listens only during the capture phase. Remember, the capture phase is the way down from the top-level application, as shown in figure 11.3. If you need to listen to both the capture phase and the bubbling phase, you must add a second event listener, omitting or passing `false` as the third `addEventListener` parameter.

You've probably already gathered this, but most actions in Flex have corresponding events for which you can listen by using the event listeners; you can then respond as needed. This is the communication and nervous system of your Flex application. Even setting variables can cause events to be broadcast. This type of event dispatch is called *binding*.

NOTE When adding listeners, a serious consideration for `weakReference` is required. If you think the object will ever need garbage collection and you aren't going to explicitly remove all event listeners from the object, use `weakReference`. Don't use `weakReference` on local objects (objects created in a method and then destroyed) because the garbage collection occurs automatically, which could cause your listeners not to fire if they're garbage collected before your expected event.

11.2.2 Binding events

Binding in Flex is carried out in the event system. When you bind a variable, you're establishing a dedicated listener that picks up on change events issued from the variable or object to which it's bound (for more about binding, refer to chapter 3).

Whenever you create a binding to a variable, you register an event listener to respond to any changes that occur in that variable. When binding in MXML, the updating takes place behind the scenes, as demonstrated in the following listing.

Listing 11.4 MXML binding

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:HorizontalLayout />
  </s:layout>
  <fx:Script>
    <![CDATA[
      [Bindable]
      protected var _labelText:String = "Label before event";
    ]]>
  </fx:Script>

  <s:Button id="myButton" label="Change Label!">
    <s:click>
      <![CDATA[
        _labelText = "Label " + Math.round(Math.random()*10);
      ]]>
    </s:click>
```

`[Bindable]` means watch this variable

```
  </s:Button>
  <s:Label id="myLabel" text="{_labelText}"/>
</s:Application>
```

Compare the code in listing 11.4 to what's required to accomplish the same thing in ActionScript (listing 11.5). This ActionScript version relies on a class called ChangeWatcher, which monitors any changes in the value of a property to which you have it bound. If a change occurs, ChangeWatcher triggers the necessary events to watch that value. It's much like an event listener object in that it listens for specific events from a property.

Listing 11.5 ActionScript binding using ChangeWatcher

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               applicationComplete="init()">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      import mx.events.PropertyChangeEvent;
      import mx.binding.utils.ChangeWatcher;

      protected var _watcher:ChangeWatcher;

      protected function init():void{
        toggleWatch();
      }
      protected function toggleWatch():void{
        if(_watcher && _watcher.isWatching()){
          _watcher.unwatch();
          toggleButton.label = "Watch";
        }else{
          _watcher = ChangeWatcher.watch(inputField,"text",onTextChange);
          toggleButton.label = "Stop Watching";
        }
      }
      protected function onTextChange(event:Event):void{
        myLabel.text = inputField.text;
      }
    ]]>
  </fx:Script>
  <s:Button id="toggleButton" label="Watch Text" click="toggleWatch()"/>
  <s:TextInput id="inputField" text="start text"/>
  <s:Label id="myLabel" />
</s:Application>
```

Necessary imports to dynamically bind variables

Verifies the watcher is active

Removes watched variable

Tells ChangeWatcher to detect changes

This method isn't as easy as other ActionScript approaches or even the shortest ActionScript approach, but it's more flexible. Upon receipt of the applicationComplete event from the application, you toggle the watcher. Because _watcher is null at startup, the first if statement in toggleWatch fails, causing the else block to run. The else block binds to changes on the inputField's text property by calling the ChangeWatcher.watch() method, which acts as the factory method for ChangeWatcher; an instantiated ChangeWatcher object is returned when you call the watch() method.

This method takes inputs for the object you want to watch and a property of the watched object that's listened to (in this case, you're watching the text property of inputField). The third property specifies which function to call when this event is triggered—in this case, onTextChange().

If you look back at the addEventListener() method, you'll see that this approach acts in a similar manner. With the addEventListener() method, you're listening to the entire object; with the method presented in listing 11.5, you're watching a specific object property. Changes made to the object property trigger the event listener.

When you type anything in the text input, ChangeWatcher automatically listens for those updates and executes the listening method. Each time you press a key, you send out an event that's monitored by ChangeWatcher. As demonstrated in listing 11.5, your binding event can be as simple as copying the user input value into the myLabel component, or it can be as complex as you need it to be.

Another benefit of this implementation is the ability to remove a binding from an object. You can't remove the binding in MXML. As shown in listing 11.5, using ActionScript, when the toggleWatch is called and _watcher.isWatching() is true, the bind event is removed using _watcher.unwatch(), causing future property changes to not call the event listener.

If you look at the toggle method, the toggleButton's label is changed to reflect the current watch state of _watcher. This could be done using multiple buttons or by listening to other types of events as well. This approach is for user interface simplicity.

> **NOTE** ChangeWatcher.watch() also has a weakReference argument. Use this argument in the same instances and for the same reasons as you would addEventListener().

The BindingUtils class also allows binding through ActionScript, but it's only a wrapper around ChangeWatcher. Using BindingUtils.bindProperty or BindingUtils.bindSetter, you can set up a ChangeWatcher binding. It's a helper class to condense the lines of code necessary to bind objects in ActionScript. The following line of code is similar to calling ChangeWatcher.watch():

```
BindingUtils.bindProperty(myLabel, "text", inputField, "text");
```

Table 11.4 lists a few of the benefits of the different approaches to data binding.

Table 11.4 Benefits of using each binding approach

	MXML Binding*	ChangeWatcher	BindingUtils
Ability to toggle on/off	No	Yes	Yes
Call methods on change	No	Yes	Yes
Two-way binding	Yes	No	No

* This includes the simple {} and @{} syntax as well as using <fx:Binding />.

11.2.3 Removing event listeners

You've just seen the `unwatch()` method in action, which lets you stop monitoring a variable for changes. When using event listeners, you have the same type of capability. If an event listener was added at runtime in ActionScript, you'd be able to remove it using the `removeEventListener()` method. `unwatch()` merely uses `removeEventListener` to stop the events from dispatching. If you look at the underlying code for the `ChangeWatcher` class, you'll notice on roughly line 500 the following statement:

```
host.removeEventListener(p, wrapHandler);
```

Take a look at listing 11.6 (testingForListeners.mxml) to see how to remove event listeners.

Listing 11.6 Example of adding and removing events

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark">
  <s:layout>
    <s:VerticalLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      protected function toggleListeners():void{
        if(box.hasEventListener(MouseEvent.CLICK)){
          log("Listeners removed");
          box.removeEventListener(MouseEvent.MOUSE_OVER, onEvent);
          box.removeEventListener(MouseEvent.MOUSE_OUT, onEvent);
          box.removeEventListener(MouseEvent.MOUSE_MOVE, onEvent);
          box.removeEventListener(MouseEvent.CLICK, onEvent);
        }else{
          log("Listeners added");
          box.addEventListener(MouseEvent.MOUSE_OVER, onEvent);
          box.addEventListener(MouseEvent.MOUSE_OUT, onEvent);
          box.addEventListener(MouseEvent.MOUSE_MOVE, onEvent);
          box.addEventListener(MouseEvent.CLICK, onEvent);
        }
      }

      protected function onEvent(event:Event):void{
        log("Event triggered: " + event.type);
      }

      protected function log(text:String):void{
        logField.text = text + "\n" + logField.text;
      }
    ]]>
  </fx:Script>
  <s:Button label="Toggle Listeners" click="toggleListeners()" />
  <s:Group id="box">
    <s:Rect width="200" height="50">
      <s:fill>
        <s:SolidColor color="0x979797" />
      </s:fill>
    </s:Rect>
```

Removes event listeners

Adds event listeners

Event handler

```
  </s:Group>
  <s:TextArea id="logField" width="400" height="400" />
</s:Application>
```

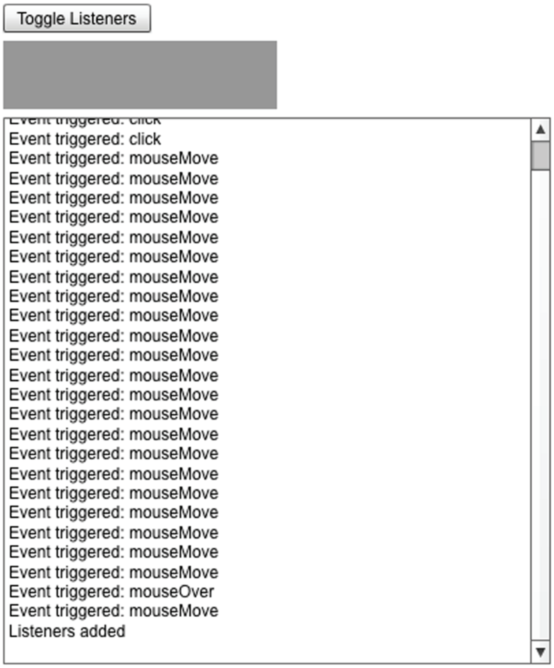
Listing 11.6 shows how to test an object to determine if a particular event listener was added and then remove the listeners or add them back accordingly so the same listener isn't added multiple times. Run the code and you'll see output similar to figure 11.4.

> **NOTE** `removeEventListener()`, in listing 11.6, uses only two arguments, but it's important to note the third parameter. The third argument is `useCapture`, which is `false` by default. When an event listener is added for the capture phase, to remove it you must pass `true` as the third argument.

`toggleListeners()` introduces a new method: `hasEventListener(type:String):Boolean`. This method checks the target object for existence of the passed-in event type. If the listener hasn't been added to the target object, the event listener is added; otherwise, it's removed, toggling the existence of the listeners.

For logging purposes, you can add and remove four different types of events. This allows you to see the different events trigger accordingly.

Knowing how to add and remove event listeners is the starting point to building a great application. Adobe didn't stop with allowing internal events. In the next section, we cover dispatching and creating custom events. This will give you ultimate control over events and enable you to dynamically determine when events are sent out and what data these events should carry.



```
Event triggered: click
Event triggered: click
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseMove
Event triggered: mouseOver
Event triggered: mouseMove
Listeners added
```

Figure 11.4 Output from listing 11.6

Removing MXML event listeners

It's important to remember that removing event listeners works only on events added using the ActionScript method of defining an event listener; listener functions added using the MXML format are permanently attached to the object. For example, you can't remove an event listener added using the MXML script shown in the following snippet:

```
<s:Button label="Toggle Listeners" click="toggleListeners()" />
```

If you think you might need to remove the event listener at some point, use the `addEventListener()` method.

11.3 Custom events

Part of what makes events so powerful is the ability to create your own custom events and use them to communicate within your application. By sending out events when data changes or when the user initiates some action, like clicking a button, you decouple an application's logic from the objects that use it. This is critical because it creates a more modular structure, allowing changes to a component without affecting other parts of your application.

Imagine dispatching an event from two separate unrelated classes. Each class has a property containing the data we need, but both classes have different property names. `ClassA` has a property named `someData` and `ClassB` has a property named `content`. When the event is dispatched, calling `event.target.someData` throws an error if the dispatcher is `ClassB`. By passing the data in the event, the listener no longer cares which target dispatched it as long as the event contains the proper data. This eliminates the listener from having to determine which class is the dispatcher, only to determine the name of the class. Also keep in mind that if the listener does use the dispatcher's property to access the data, the listener now is tightly coupled to the dispatcher, so changes to the expected property name means changes to the listener as well.

This is where and why custom events shine.

11.3.1 Dispatching custom event types

All objects implementing the `flash.events.IEventDispatcher` interface can dispatch events, which include all display objects as well as other nondisplay classes. Typically, you achieve this in your custom nondisplay classes by extending `flash.events.EventDispatcher`. You'll see this in section 11.3.2. For now, let's focus on the basics of dispatching a custom event type.

Consider the following snippet:

```
dispatchEvent(new Event("complete"));
```

It performs two distinct operations:

- A new event class is instantiated to handle and hold the information.
- The event of type `"complete"` is dispatched using the `dispatchEvent()` function call.

When dispatching events with custom types, try your best to utilize an already available event constant (`Event.COMPLETE`, `Event.CHANGE`, and so on) or create your own static constants. You won't get beat up in a dark alley if you don't, but we're aiming to write solid, reusable code, right? To do so you can create static constants on the dispatcher or, if you want to be in the cool crowd, create a custom event class with constants. Doing so will allow you to write code capable of being used in multiple applications or even in multiple places within the same application. This also allows you to pass custom data with the events.

11.3.2 Creating custom events

Dispatching a custom event type is easy, but, as noted earlier, it won't get you any closer to stardom with other skilled developers or, more importantly, help reduce your application maintenance. Custom events give you full control over what data is sent with an event, the event types, default values, and so on.

To demonstrate, let's create a simple real-world example of adding event dispatching to a custom nondisplay class. Our application will load an XML file and display the results in a list. The focus is on the event dispatching though, not the user interface.

Start by creating a new ActionScript class named `DataLoader` in a packaged named `net` (see figure 11.5). This class will handle all of the data loading and parsing. Copy the code from listing 11.7 into the new class.

Listing 11.7 `net.DataLoader` class

```
package net{
  import events.ContentEvent;

  import flash.events.ErrorEvent;
  import flash.events.Event;
  import flash.events.EventDispatcher;
  import flash.events.IOErrorEvent;
  import flash.events.SecurityErrorEvent;
  import flash.net.URLLoader;
  import flash.net.URLRequest;

  import mx.collections.ArrayCollection;

  public class DataLoader extends EventDispatcher{
    protected var _loader:URLLoader;

    public function DataLoader(){
      super();

      _loader = new URLLoader();
      _loader.addEventListener(Event.COMPLETE, onComplete);
      _loader.addEventListener(IOErrorEvent.IO_ERROR, onError);
      _loader.addEventListener(SecurityErrorEvent.SECURITY_ERROR, onError);
    }

    public function load(url:String):void{
      _loader.load(new URLRequest(url));
    }
```

Extend `EventDispatcher` class

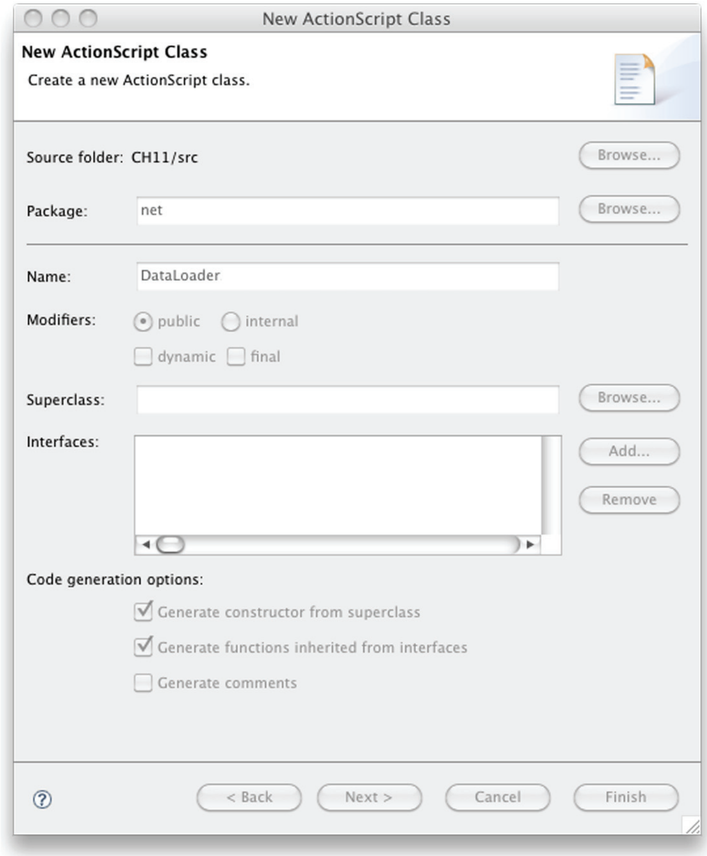
```
    protected function onComplete(event:Event):void{
      var users:ArrayCollection = new ArrayCollection();
      for each(var user:XML in XML(_loader.data).user){
        users.addItem(user.@name + " - " + user.@site);
      }

      var ev:ContentEvent = new ContentEvent(ContentEvent.DATA_BACK);
      ev.users = users;
      dispatchEvent(ev);
    }

    protected function onError(event:ErrorEvent):void{
      var ev:ContentEvent = new ContentEvent(ContentEvent.DATA_ERROR);
      ev.error = event.text;
      dispatchEvent(ev);
    }
  }
}
```

Create and dispatch complete custom event

Create and dispatch error custom event



New ActionScript Class

Create a new ActionScript class.

Source folder: CH11/src

Package: net

Name: DataLoader

Modifiers: ● public ○ internal

□ dynamic □ final

Superclass:

Interfaces:

Code generation options:

☑ Generate constructor from superclass

☑ Generate functions inherited from interfaces

□ Generate comments

Figure 11.5 Use the New ActionScript Class Wizard to create the DataLoader class.

The DataLoader is simple. In the constructor an instance of the URLLoader is created with events tied to the instance to monitor the Event.COMPLETE, IOErrorEvent. IO_ERROR, and SecurityErrorEvent.SECURITY_ERROR events. From here the class does nothing until the load() method is called, which uses the loader to load the content.

Once the content is loaded, the onComplete() method handles the data parsing; then you get into dispatching the custom event:

```
var ev:ContentEvent = new ContentEvent(ContentEvent.DATA_BACK);
ev.users = users;
dispatchEvent(ev);
```

Notice that the only difference from dispatching the events as you have before is the assignment of the users property. The event type is referenced from a static constant of the custom event, and you're still passing the event object to dispatchEvent().

The onError() method dispatches an event in the same way. You create the event, passing in the custom type, in this case ContentEvent.DATA_ERROR, set the error property, and dispatch the event. Typically you want the onError method to redispatch the ErrorEvent using the clone() method. We discuss ways of improving this class at the end of this section.

Create a new ActionScript class named ContentEvent in the events package, and copy listing 11.8 into the class.

Listing 11.8 events.ContentEvent–The custom event for content

```
package events{
  import flash.events.Event;

  import mx.collections.ArrayCollection;

  public class ContentEvent extends Event{
    public static const DATA_BACK:String = "dataBack";
    public static const DATA_ERROR:String = "dataError";

    public var users:ArrayCollection;
    public var error:String;

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

Event constants used for event references

Custom data properties

This is a bare-minimum custom event class. It's important to note the constants. These are the event types, and the names are completely up to you. Normally better names would be chosen, like COMPLETE and ERROR, but it's a great way to introduce event-naming constructs.

The format of event names should be in all caps with an underscore (_) between words. The values of the constants are also completely up to you, but camel case is suggested. We'll cover why in the metadata section.

Event name conflicts can and do occur. If ContentEvent had an event named COMPLETE (versus DATA_BACK) and the DataLoader dispatched the event, the listener could also be triggered by dispatching Event.COMPLETE. In these cases some developers choose to name their events with full names, for example, events.ContentEvent.DATA_BACK. This breaks the metadata constructs and disables MXML event access in some cases, but when done right, it completely ensures the listeners are called only for the specific event and no other events. This is especially important when listening to the Bubbling and Capture phases or when using Flex frameworks like Mate (http://mate.asfusion.com) or Swiz (http://swizframework.org).

Best practice

When building custom events, it's best to leave the constructor alone. Some developers like to change the constructor arguments to include type but leave out bubbles and cancelable. This will only confuse other developers new to your code.

One thing the ContentEvent class is missing is the clone() method. We previously noted that the class is the bare minimum, and it is. All of your custom events should always override the Event.clone() method. If the method isn't overridden and the event is cloned, the custom data won't make it into the clone, which means the clone won't match the original event.

All the method does is create a new instance of itself and set the new instance properties to match the current properties, essentially a clean copy of the event. Here's the clone() method as it would be implemented in the ContentEvent class:

```
override public function clone():Event{
  var event:ContentEvent = new ContentEvent(type, bubbles, cancelable);
  event.users = users;
  event.error = error;
  return event;
}
```

At this point the DataLoader class is complete. For now the class does exactly what you need: load data and dispatch complete and error events. The next step is to build the application.

Create a new MXML application in the (default package) named UsersView.mxml and copy listing 11.9 into the file.

Listing 11.9 UsersView application

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               width="300" height="200"
               applicationComplete="init()">
  <s:layout>
    <s:BasicLayout/>
  </s:layout>
  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      import events.ContentEvent;
      import net.DataLoader;
      protected function init():void{
        var dataLoader:DataLoader = new DataLoader();
        dataLoader.addEventListener(ContentEvent.DATA_BACK, onData);
        dataLoader.addEventListener(ContentEvent.DATA_ERROR, onError);
        dataLoader.load("data/content.xml");
      }

      protected function onData(event:ContentEvent):void{
        userList.dataProvider = event.users;
      }

      protected function onError(event:ContentEvent):void{
        Alert.show(event.error, "Error!");
      }
    ]]>
  </fx:Script>
  <s:List id="userList" width="100%" height="100%" />
</s:Application>
```

Create listeners for `DataLoader`

Initiate load request

Set list data to result

Show alert on errors

The application starts with the `applicationComplete` event, as shown previously, to call the `init()` method. This is where you instantiate a local instance of the `DataLoader` class, add event listeners for the `ContentEvent.DATA_BACK` and `ContentEvent.DATA_ERROR` events, and then tell it to load the `data/content.xml` file, which is shown in the following snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
     <user name="John C. Bland II" site="http://www.johncblandii.com" />
     <user name="Tariq Ahmed" site="http://www.dopejam.com" />
     <user name="Dan Orlando" site="http://www.danorlando.com" />
     <user name="Joel Hooks" site="http://www.joelhooks.com" />
</users>
```

Once the data load is complete, the `onData` method is called, and you update the `userList` component to display the results. That's it. Your display is super simple, and all of the underpinnings are handled in separate classes. The beauty here is the `DataLoader` can change to loading content via internal dummy data, Flash Remoting (AMF), SQLite (via Adobe AIR), and so on, and your `UserView` application won't have to be touched beyond the URL to load the content.

You can't leave out the `onError` method. Even though the `DataLoader` listened for two events (`IOErrorEvent.IO_ERROR` and `SecurityErrorEvent.SECURITY_ERROR`), your application had to worry about only one: `ContentEvent.DATA_ERROR`. When an error occurs, the `onError` method shows a simple alert displaying the `event.error` property as the message of the alert. Again, the underlying code can change to display any type of message, but your application stays the same. `DataLoader` could choose to use localized text, display a generic message, or display an error from the server. The possibilities are endless, but the changes to your application are nil.

Now that you've seen how to dispatch custom events, let's look at how to improve our events with metadata.

11.3.3 Adding event metadata to custom dispatchers

An important part of completing the dispatch of events—particularly when using custom components and classes—is adding metadata to dispatchers, to take advantage of code hinting in Flash Builder.

It's not necessary to add metadata, but doing so adds convenience by giving you the ability to see events as properties in MXML or to get code hints in ActionScript. Listing 11.10 shows how to add metadata to the DataLoader class from listing 11.7.

Listing 11.10 Event metadata DataLoader class

```
package net{
  ...
  [Event(name="dataBack", type="events.ContentEvent")]
  [Event(name="dataError", type="events.ContentEvent")]
  public class DataLoader extends EventDispatcher{
    ...
  }
}
```

Event metadata for all errors dispatched by the DataLoader

The name attribute is required and refers to the property value, not the static constant of the class. Don't use DATA_BACK as the name. Instead, use the value of DATA_BACK, which is dataBack. This is where constant values are critical for code hinting. If the constant's value isn't in CamelCase, Flash Builder won't properly interpret which static constants are available from the event class referenced in the type attribute.

The type attribute references the event class. If the event class is flash.events.Event, the type attribute isn't required.

Why is this important? We're glad you asked! Figure 11.6 shows the before and after screenshots for code hinting in ActionScript.

To demonstrate the effectiveness for an MXML implementation, create a new MXML application named UserView2.mxml and copy listing 11.11 into the file.

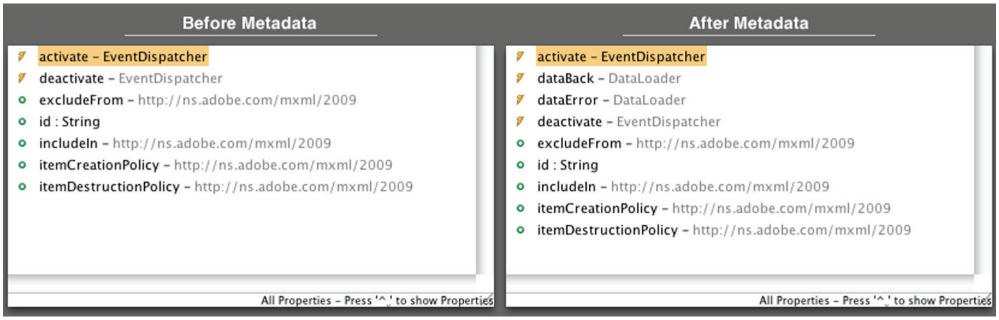

Before Metadata
activate – EventDispatcher
deactivate – EventDispatcher
excludeFrom – http://ns.adobe.com/mxml/2009
id : String
includeIn – http://ns.adobe.com/mxml/2009
itemCreationPolicy – http://ns.adobe.com/mxml/2009
itemDestructionPolicy – http://ns.adobe.com/mxml/2009
All Properties – Press '^ ' to show Properties
After Metadata
activate – EventDispatcher
dataBack – DataLoader
dataError – DataLoader
deactivate – EventDispatcher
excludeFrom – http://ns.adobe.com/mxml/2009
id : String
includeIn – http://ns.adobe.com/mxml/2009
itemCreationPolicy – http://ns.adobe.com/mxml/2009
itemDestructionPolicy – http://ns.adobe.com/mxml/2009
All Properties – Press '^ ' to show Properties

Figure 11.6 Before (left) and after (right) the custom metadata

Listing 11.11 `UserView` implementing `DataLoader` in MXML

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               xmlns:net="net.*"
               width="300" height="200"
               applicationComplete="init()">
  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      import events.ContentEvent;
      protected function init():void{
        dataLoader.load("data/content.xml");
      }

      protected function onData(event:ContentEvent):void{
        userList.dataProvider = event.users;
      }

      protected function onError(event:ContentEvent):void{
        Alert.show(event.error, "Error!");
      }
    ]]>
  </fx:Script>
  <fx:Declarations>
    <net:DataLoader id="dataLoader" dataBack="onData(event)"
                    dataError="onError(event)" />
  </fx:Declarations>
  <s:List id="userList" width="100%" height="100%" />
</s:Application>
```

`DataLoader` implemented in MXML

The main change is that `DataLoader` is now instantiated in MXML with the `dataBack` and `dataError` events shown as MXML properties. This isn't possible without metadata on the `DataLoader` class.

It's important to note how to add metadata in an MXML component. The exact same event metadata is used with a slight change, as shown in the following code:

```
<fx:Metadata>
  [Event(name="eventName", type="EventClass")]
</fx:Metadata>
```

NOTE Adding class-level metadata also improves your documentation when using ASDoc.

Using events with custom dispatchers as we've presented in this chapter lets you retain loose coupling. Your components don't need to know about each other, and the parent windows can control their behavior and manage interactions. You'll find out more about custom components in chapters 17 and 18.

Our last stop on the event train is how to stop the event, pun intended.

11.3.4 Stopping event propagation

During any event phase (capture, target, or bubbling), you can use the event's `stopPropagation()` and `stopImmediatePropagation()` methods to discontinue the event from broadcasting to any other components. The two methods are virtually identical, differing only in whether other event listeners on the same component are allowed to receive the event.

For example, if the `event.stopPropagation()` method is used on an event, it discontinues propagation after all other event listeners on a given component have finished responding to the event. If you were to use the `event.stopImmediatePropagation()` method, event propagation would be terminated before it was delivered to any other events, even if they were listening on the same component.

When used in conjunction with the `priority` attribute of the event listener (set when adding the event listener), you can create a function to be the first responder to the event. This can be an effective gating mechanism to evaluate an event and cease propagation if necessary to any other event listener on a given component (see the following listing).

Listing 11.12 Stopping propagation

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
               xmlns:s="library://ns.adobe.com/flex/spark"
               applicationComplete="init()">
  <fx:Script>
    <![CDATA[
      import mx.controls.Alert;
      public function init():void{
        button.addEventListener(MouseEvent.CLICK, onClick);
        box.addEventListener(MouseEvent.CLICK, onParentClick);
      }
      public function onClick(e:Event):void{
        Alert.show("AS event. Calling stopPropogation.","First Event");
        e.stopPropagation();
      }
      public function onButtonClick(e:Event):void{
        Alert.show("MXML Click Event.", "Second Event Listener");
      }
      public function onParentClick(e:Event):void{
        Alert.show("You should never see this alert.", "Parent Event");
      }
    ]]>
  </fx:Script>
  <s:HGroup id="box" width="100%">
    <s:Button id="button" label="Fire Event" click="onButtonClick(event)"/>
  </s:HGroup>
</s:Application>
```

Adds event listener with high priority

Stops propagation— second event fires

Event never reaches parent

You interrupt the propagation of the event at the button level. Normally, this event would travel through all the functions, triggering all three alerts. But after the first event, you call the `stopPropagation()` method. When you run the example, the first two events—those listening directly to the button—will run, but the parent event listening to `box` won't receive the event and therefore won't run.

If you change the `stopPropagation()` method to `stopImmediatePropagation()`, you'll see only the first alert. The `stopImmediatePropagation()` method terminates any and all delivery of the event beyond the first event listener. Discontinuing the propagation of events is an effective way to handle your event flow, depending on the circumstances. This is true when you have custom components sending out custom events.

You've learned the event flow (phases), how to start and stop listening (adding and removing event listeners), how to create custom events, and how to build your own event dispatchers. Dive into events as much as you can. They're crucial to the completion of great applications and key portions of the available Flex frameworks.

11.4 Summary

This chapter focused on the events system and how this system is used in Flash Player as well as Flex specifics, namely binding. The events system is arguably the most important upgrade from ActionScript 2.0 to ActionScript 3.0.

If you're coming from a web development background, learning asynchronous events is a paradigm shift from the standard request/response model of creating web applications. The events system is the core of this shift. Everything you do through Flex in some way touches upon the events system. When you key into an application's events, you can free your objects to behave independently, to the point where they don't need awareness of the application.

Now that you've tackled the events system, we'll move on to discuss application navigation, building on your event knowledge while continuing to improve your application's structure, functionality, and portability.

WEB DEVELOPMENT

FLEX 4 IN ACTION

Ahmed • Orlando • Bland • Hooks

Free ebook
SEE INSERT

Flex has grown from just a way to build Flash apps into a rich ecosystem, and Flex 4 introduces new UI components, better performance monitoring, and speed enhancements to the compiler.

Flex 4 in Action is a comprehensive tutorial that introduces Flex to web designers and developers. It starts with the basics—forms and data—and moves through core concepts like navigation, drag-and-drop, and events. Even if you're new to Flex, this book is all you'll need to make your apps pop using the new Spark components, data services, charting, special effects, and more.

What's Inside

- How to architect your applications
- Use charting to build interactive dashboards
- Improve productivity with network monitoring and unit testing
- Give your apps a unique look with themes and skins
- And much more

Readers of this book need basic development skills, but no previous experience with Flex.

Tariq Ahmed is an RIA engineer and Flex community evangelist. **Dan Orlando** is an RIA architect, specializing in Flex and AIR. **John C. Bland II** is an independent Flex, ColdFusion, and mobile developer. **Joel Hooks** is a Flash Platform developer and ActionScript expert.

"*The* desk reference for all things Flex 4"
—John Griffin, Overstock.com

"No question is left unanswered, no facet unexplored."
—Peter Pavlovich, Kronos Inc.

"A great book for both beginners and experienced Flex developers."
—Kevin Schmidt
Adobe Systems, Inc.

"The lessons are memorable, witty, and very relevant."
—Zareen Zaffar, Amcom

"Completely demystifies building rich user interfaces."
—Rick Wagner, Acxiom Corp.

"What you need to be flexible 4 your job!"
—Rick Evans, SAS

For online access to the authors and a free ebook for owners of this book, go to manning.com/Flex4inAction

ISBN 13: 978-1-935182-42-9
ISBN 10: 1-935182-42-0
54999
9 781935 182429

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]