# Flash Article

## Data Binding in Macromedia Flash MX Professional 2004


### by Aral Balkan

BitsAndPixels


If you have developed data-rich applications in Delphi or .NET, chances are you have already seen (or at least heard of) data binding. Flash developers got their first introduction to data binding with the release of Macromedia Data Connection Kit and Macromedia Firefly components. If you are a Firefly user and are comfortable with the concepts behind data binding, you can skip ahead to the What's Changed Since Firefly section, where I cover the differences between Firefly and Macromedia Flash MX Professional 2004 data-aware components. Everyone else, read on…

Data binding, quite simply, lets you bind a property of one component to a property of another component, so that when one changes, the other changes as well. Although it seems like a very simple concept (and in reality, it is), it automates a great many of the tasks that you, as a developer, have to take care of when building an application. I could go on and on about the origins of data binding but the simplest way to understand it is to see it in action. So let's jump right in and bind two components together using data binding.

## Requirements

### Macromedia Flash MX Professional 2004

- Try
- Buy

## Example: Data Binding Between Two UI Components

The fictitious discount airline, noFlights Online, has a novel business plan and needs a way for their online users to check for flights on its intended day of departure. Before users can do that, of course, they must enter their date of departure. The easiest way for them to do that is through a calendar control. Recruited by the Outrageous Markup Staffing Agency (another fictitious company), you are sent to noFlights with the mission of creating this amazing tool!

**Figure 1.** Finished calendar application.

1. Drag a DateChooser component onto the Stage. (DateChooser is a calendar component in version 2 in Macromedia Component Architecture speak.) Using the Property inspector, give the component instance the instance name **flightDateChooser**.
2. Drag a label component onto the Stage and give it the instance name **infoLabel**.
3. Give the label some initial text to display by setting its text property in the Property inspector to: **Select a Date**.

   OK, so now you have two components on screen. When the user clicks on the calendar to select a date, display the selected date as text in the label. Simple enough, right? Without data binding, you would have to listen to a change event on the calendar, ask it for its currently selected value, format it for display, and write it out in a text field. You can achieve the same result much faster using data binding:

4. Select the flightDateChooser instance on the Stage.
5. Open the Component Inspector panel by selecting Window >Development Panels > Component Inspector or by pressing Alt-F7. Click the Bindings tab.
6. In the Bindings tab, click the Add Binding icon (the plus sign).

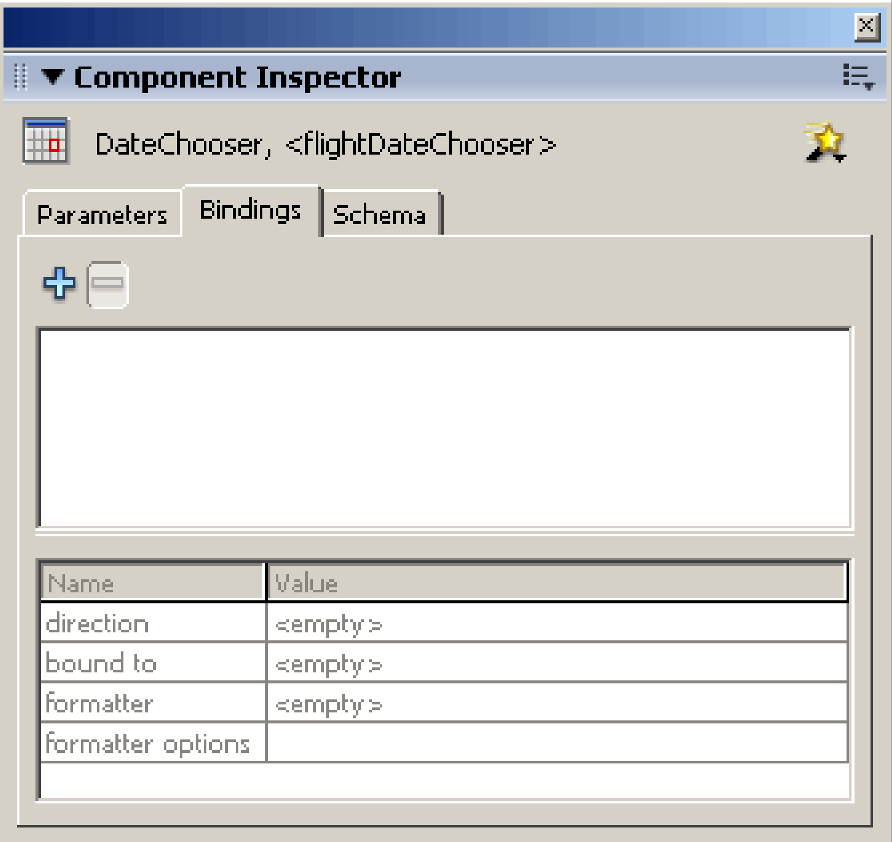



**Figure 2.** The Component Inspector panel.

7. The Add Binding dialog box appears and shows you the component properties for data binding. Since you are interested in updating your Label component when a user selects a date in the calendar, bind from the selectedDate property. Select the **selectedDate** property (Figure 3) and click OK to finalize your selection.

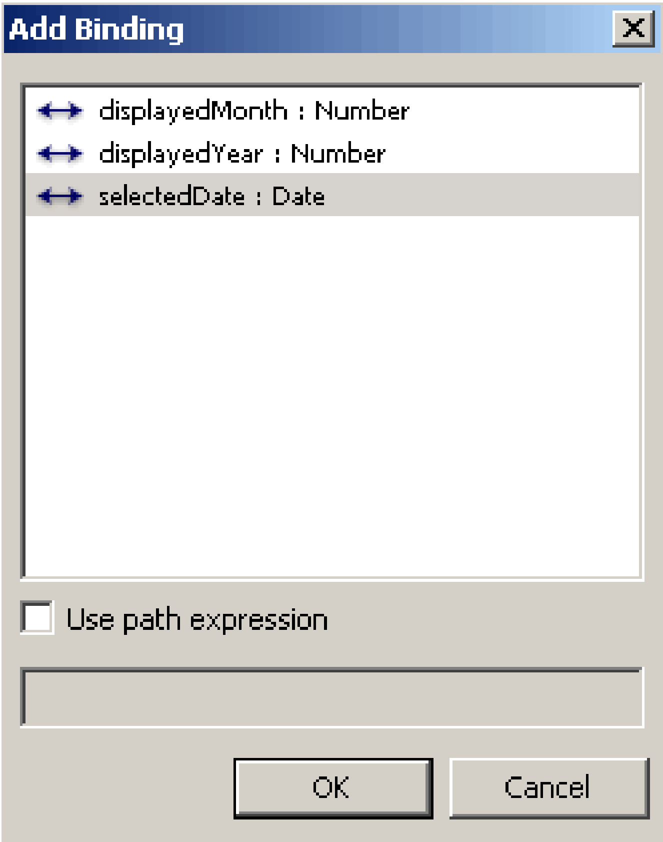



**Figure 3.** The Add Binding dialog box.

8. Looking in the Component Inspector panel, see how Flash has added a binding for the selectedDate property. By default, it is also selected as the only binding available and you can see that it has some other settings, such as direction, bound to, formatter and (grayed out) formatter options.

   At this point, you have specified that you want to bind the selectedDate property but you have not specified within Flash where you want to bind it to. To do this, click the Value cell next to "bound to." When you do this, the cell highlights and a magnifying glass icon appears. Click the same cell again to display the Bound To dialog box.

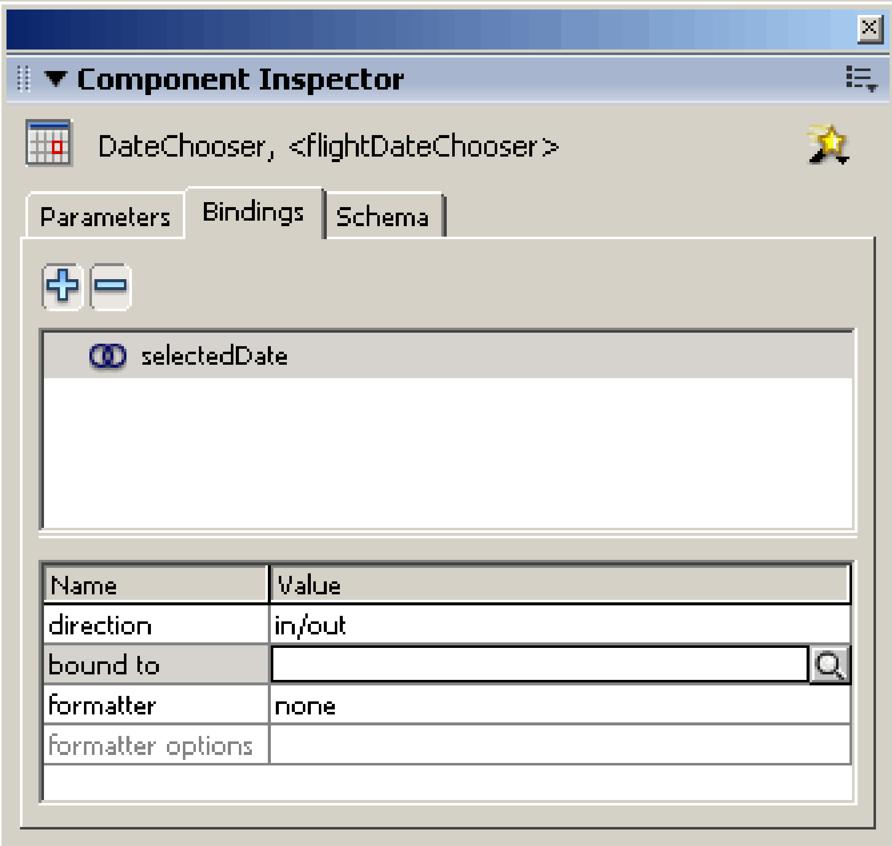



**Figure 4.** The Component Inspector panel, with the bound to parameter selected.

9. The Bound To dialog box lists all of the components you currently have in your movie in the Component path pane. You can tunnel down to find each component's bindable properties. In this instance, you want the selected date to display in the Label component. Therefore, click the infoLabel Label instance and select its text property from the Schema location pane. Click OK to finalize your selection.

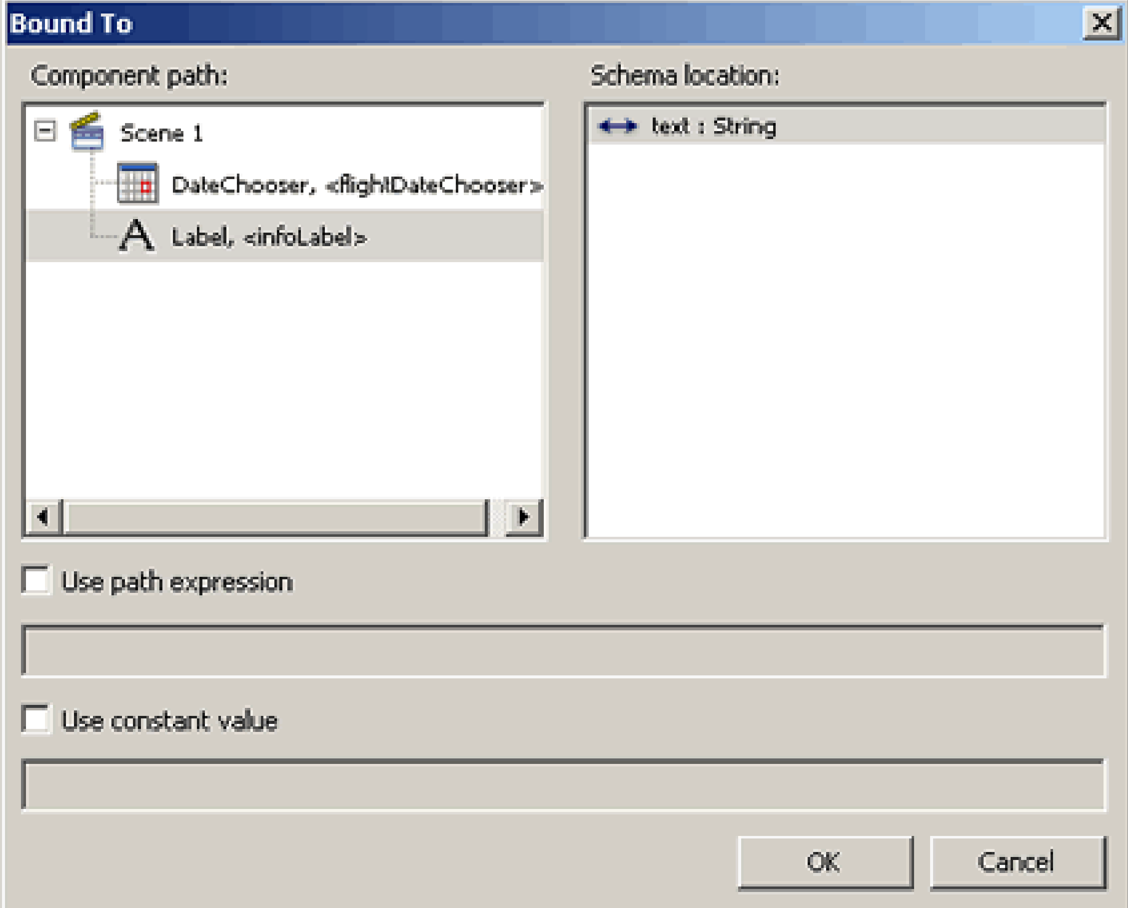



**Figure 5.** The Bound To dialog box with the infoLabel component selected.

10. Great—now that you've bound the selectedDate property of the DateChooser component to the text property of the Label component, you should see the selected date in the Label component whenever you select a date in the DateChooser. Now you can try it out! Test the movie and click on some dates in the DateChooser.

    First the good news: It works! The text in the Label really updates when you select a date (and how easy was that to set up?)

    Now the bad news: It's ugly! No human being (except those of who us like to do binary math in our heads and can actually beat a computer at chess) would want to look at a UTC timestamp. We need to format the date so it displays in a user-friendly manner. Fortunately, Macromedia Flash gives us an easy way to do just this.

11. Click the infoLabel label instance to select it and look in the Bindings tab of the Component Inspector panel. You need to add a formatter.
12. Click the text binding entry in the Bindings panel to select it. Click the empty value cell next to formatter twice to display the formatter pop-up menu and select the **Date** formatter from the list. See how the formatter options row displays the default format template for the Date formatter (MM/DD/YYYY HH:NN:SS) now.
13. Now you don't really want the time to display, so click the value cell for formatter options and

change the template to **MM/DD/YYYY**. When you're done, the Bindings tab for your Component Inspector panel will look like Figure 6.

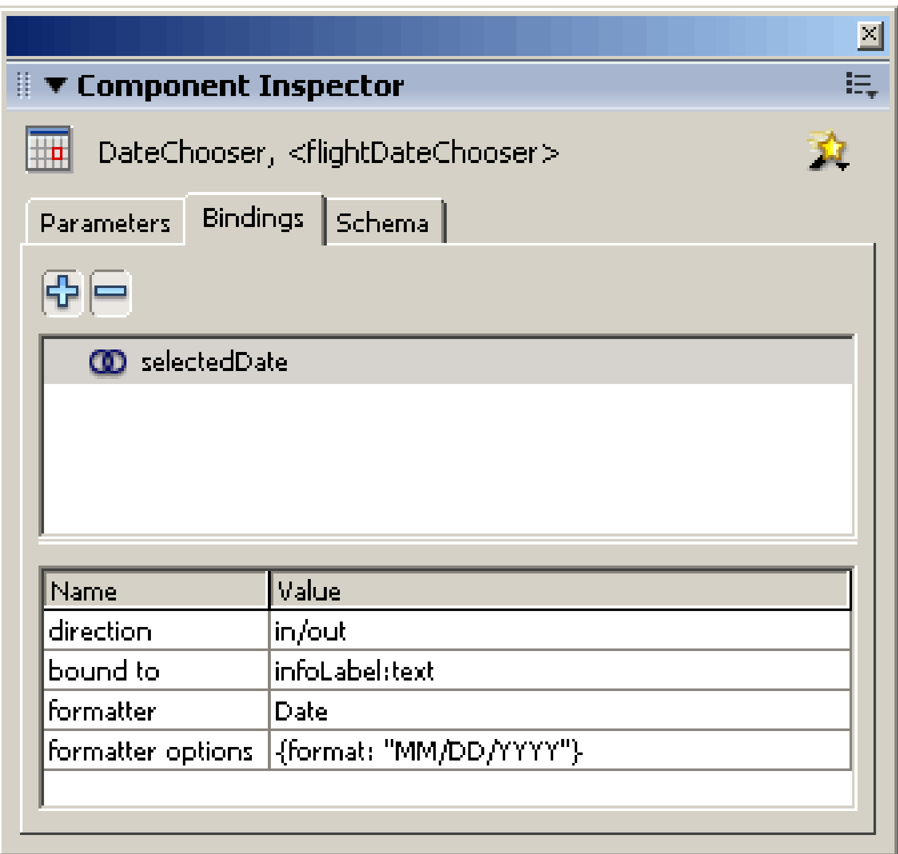



**Figure 6.** Bindings panel with date formatting settings.

14. Now, test the movie...ah, much better!

Congratulations! Not only have you created your first, albeit, overly simplistic application with data binding, but you've even used a formatter. And all this without writing a single line of ActionScript. (It is possible to do all of this entirely through ActionScript and create bindings at runtime, but that topic is out of the scope of this introductory article.) In the next example, you will go further than simply binding components together and learn about one of the data-only components, the DataHolder. Later in the article, I will also cover other types of formatters, and you will even write your own custom formatter.

But first, for you Firefly users upgrading to Flash MX Professional 2004, let's quickly review what has changed. If you have never used Firefly, you can skip the next section.

## What's Changed Since Firefly?

Towards the end of the Macromedia Flash MX days, Macromedia released a set of data components called Firefly. These components form the data-aware components in Macromedia Flash MX 2004 Professional. Although the data-aware version 2 components in Flash MX 2004 Professional share many

of the same fundamental concepts as the Firefly components, Macromedia has made numerous enhancements to them and has fully incorporated them into the v2 of the Macromedia Component Architecture. Although the general workflow remains the same, there are several important differences, which I highlight in this section.

## Goodbye Custom UI, Hello Component Inspector

The new components no longer feature Custom User Interfaces. The Firefly components used tabbed Custom UIs, which, although effective, suffered from usability issues inherent in all Custom UIs, such as lack of multilevel undo and keyboard-based cut and paste.

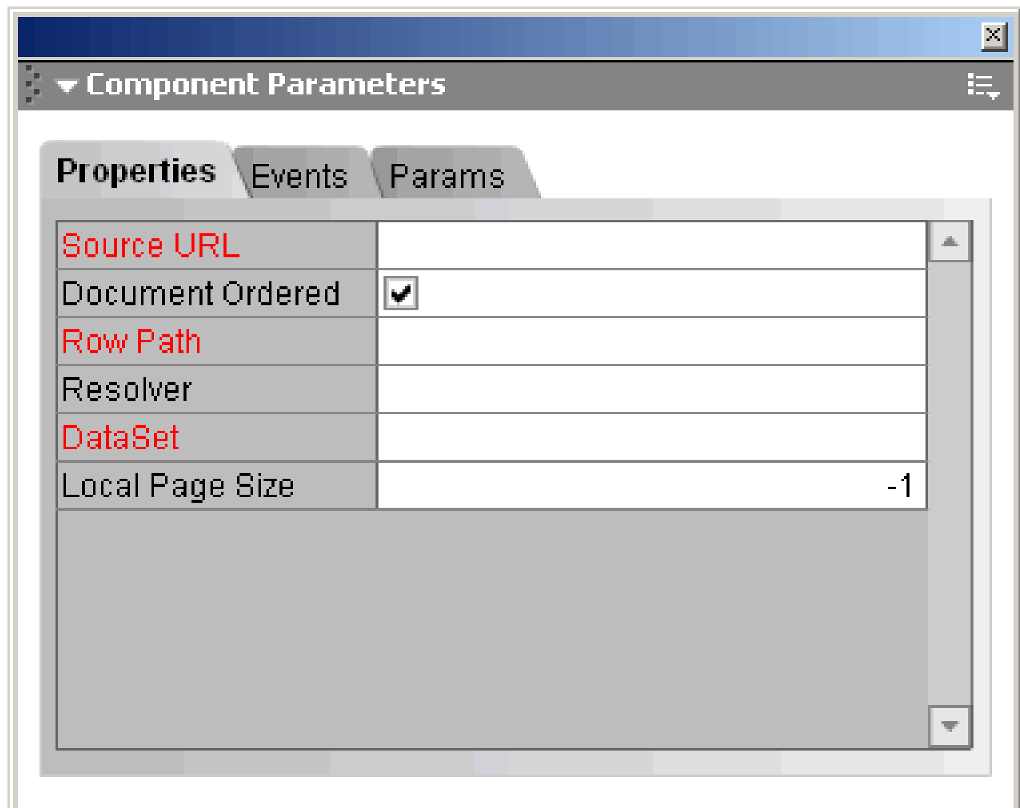



**Figure 7.** Firefly XML Connector Custom UI with the Properties tab active.

Instead of the Custom UI, we now have the Component Inspector, a single panel you use when you set the properties of any v2 component.

There are both advantages and disadvantages to this new approach. It is of course better to have a consistent, single location for setting component properties. Also, since the Component Inspector panel is part of the Flash authoring environment (IDE), it benefits from multilevel undo and all of the keyboard shortcuts and usability enhancements available to the IDE. However, one side effect of this change is that you may have to write a bit more code in some instances, such as when setting up event listeners. (The Component Inspector panel does not include the Events tab in Firefly Custom UIs.)

## Version 2 Components and Flash Remoting

Firefly shipped with a plug-in (Connector/Resolver pair) for Flash Remoting. This is now notably missing among the plug-ins (now simply referred to as connectors and resolvers) currently available with Macromedia Flash MX Professional 2004. Before you jump to any conclusions, however, let me put your mind at peace: No, Macromedia Flash Remoting has not been deprecated and yes, you can use Flash Remoting with data binding and the new data-aware v2 components.

### Transfer Objects

Although you can see most of the differences between Firefly and the v2 components instantly, perhaps the biggest change of all has taken place under the bonnet, in the very way that Flash manages data. Instead of working with pure data, the v2 components use transfer objects (based on the core J2EE Transfer Object pattern). This means, for example, that each item in a DataSet is an instance of a given custom class with methods and properties. These methods, in turn, can mirror the methods available for the transfer object on the server side. By default, if a class is not specified, Flash creates an instance of the Object class.

In addition to the changes I'm presenting here, Macromedia has added many improvements and new features since the Firefly days and I will be looking at some of those in the following sections.

### Data Binding with the Data Holder

In the first example, you saw how you could use data binding to bind UI components together. Although this is sufficient for simple functionality, to create more complex interactions it's nice to be able to bind all UI components in a screen to a single component that acts as a data repository. There are two such components available, the DataSet and the DataHolder. In this section, you'll be using the simpler of the two, the DataHolder.

The DataHolder does not have a visible presence in your application (although you drag the component onto the Stage to add it to your application, it will not be visible when you run your application.) The purpose of the DataHolder is to act as a central holding area for data and to fire off events when this data changes.

Again, let's jump right in with an example and see the DataHolder at work.

## Example 2: Data Binding Using a Data Holder

The Organic Bakery, a fictitious bakery specializing in certified organic bread, has asked you to create an e-commerce solution for them. Since they want the best experience for their customers, they've hired a firm that uses agile development methodologies along with usability design and testing. You happen to work for the firm as a developer—lucky you! Today, your task is to spike out (quickly test) the suitability of the version 2 components (and data binding) for the shopping cart facility. You decide to

concentrate on the Add To Cart feature and keep the spike simple with a single product. Your graphic designer friend in your "caves and common" seating arrangement, seeing what you're working on (and with too much time on her hands), whips up an image and a design which you reluctantly agree to use!

**Figure 8.** Organic Bakery (a fictitious company): Add to Cart screen showing a DataHolder component with the priceLabel instance selected.

Use the final layout of the FLA, illustrated in Figure 8, as a guide throughout this example.

1. Drag three Label components onto the Stage and give them the instance names **priceLabel**, **vatLabel**, and **totalLabel**.
2. Enter default text for each of the labels using the Property inspector. In the example above, you can see that I set them as price label, vat label, and total price label, respectively.
3. Drag a NumericStepper component on the stage and give it the instance name **quantityStepper**.
4. You can add standard static text fields to the left of the Label and NumericStepper components as **Price**, **Quantity**, **VAT** (value-added tax, a purchase tax charged in most of Europe, as well as other places), and **Total Price** as well as any graphic embellishments the designer in you feels compelled to add. (Although, remember, this is a spike so you normally wouldn't care at all about its graphic design.)

   Note that the Label component is a component. Do not confuse it with a regular text field. You are using the Label components so you can bind data to them and you're using standard static text to "label" them. (I know, it can get confusing.)

5. Now for the fun part: You want the TotalPrice field to update each time the user selects a different quantity of bread to buy using the NumericStepper. To do this, multiply the price by the quantity, calculate the VAT due, and add the two results together to arrive at the TotalPrice. As you can see, this is a somewhat more complex requirement and you can't just bind one component to another to realize it. Enter the DataHolder!

Drag a DataHolder instance on the stage and give it the instance name **myDataHolder**.

6. You're going to bind each of the UI components to the DataHolder and use of the DataHolder's events to calculate the TotalPrice each time the quantity changes. To begin with, you need to set up the schema for the DataHolder to tell it what properties it should expect. This also makes these properties available so you can bind the UI components to them.

   Click myDataHolder on the Stage to select it and bring up the Component Inspector's Schema tab.

7. In the Schema tab, you should see that there is a default property, called Data (an Object) that has already been created for you. Although you can, you don't have to use this property; instead, use the Add a Component Property button (the one with the large plus sign icon) to add four new properties. As you add a property, it highlights the field name value. Enter the following field names for the new component properties: **price**, **quantity**, **vat** and **totalPrice**. (The order of the properties in the Schema list is not important.)
8. Set the data type attribute of the price property to Number, as shown below in Figure 9.

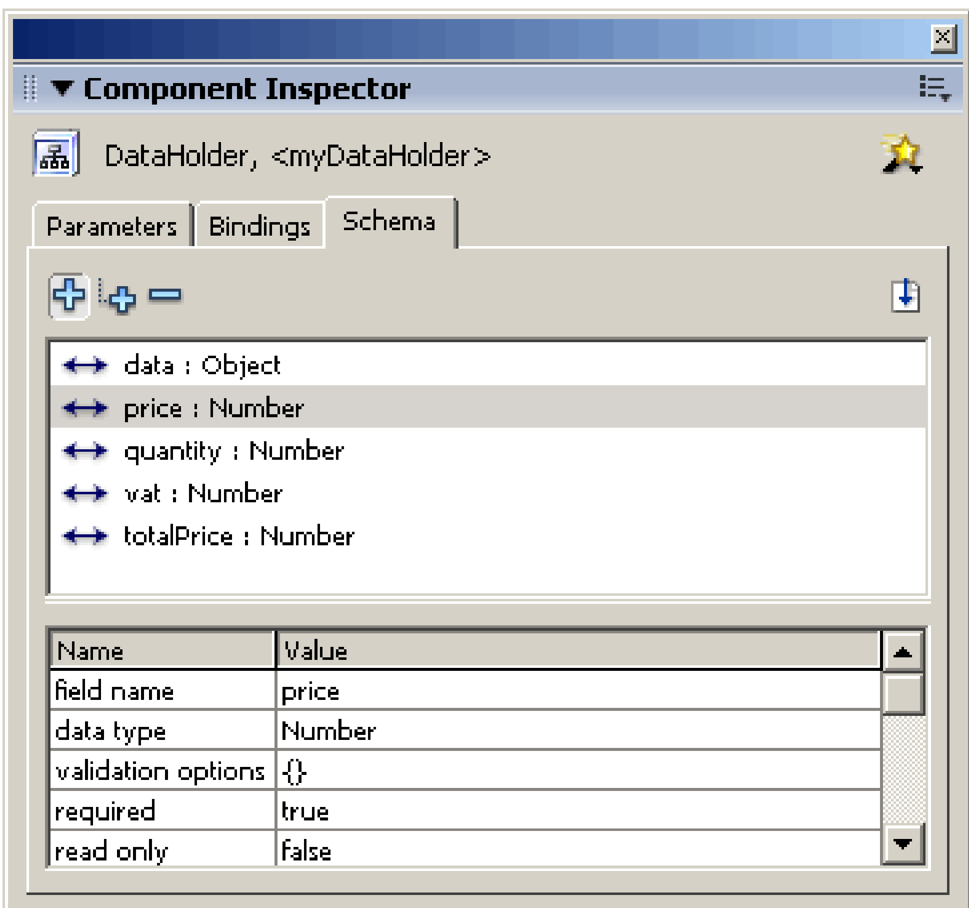



**Figure 9.** Component Inspector panel showing the price property of myDataHolder.

9. Similarly, set the data type attributes of the vat, quantity, and totalPrice properties to **Number**.

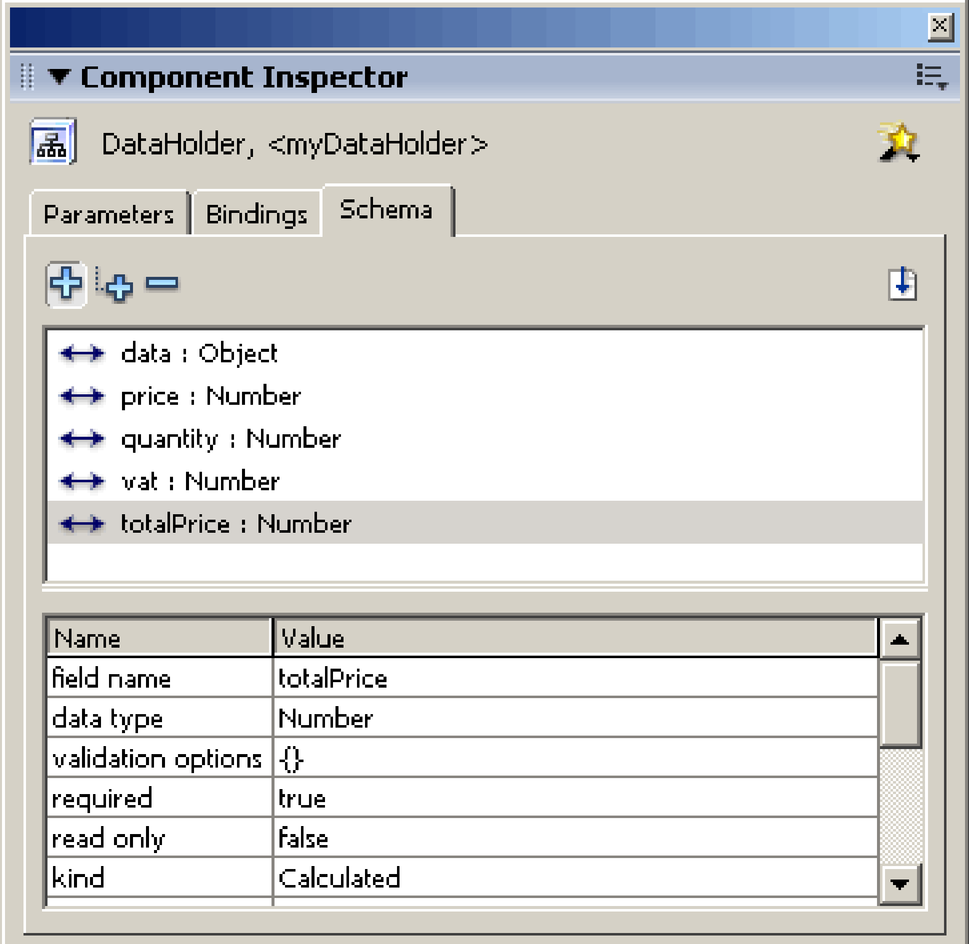



**Figure 10.** Component Inspector panel showing the totalPrice property of myDataHolder.

10. Now that you've set up the schema of the DataHolder, create the bindings from the UI components to the DataHolder. Let's start with the priceLabel Label component.

    Click the priceLabel Label component on the Stage to select it and, using the Bindings tab of the Component Inspector panel, create a binding from its text property to the price property you just created in the DataHolder. Set up the Number Formatter and set it to **2** for the decimal place precision, as shown below (this is so we can display the price in a format the customer is used to, such as £1.69).

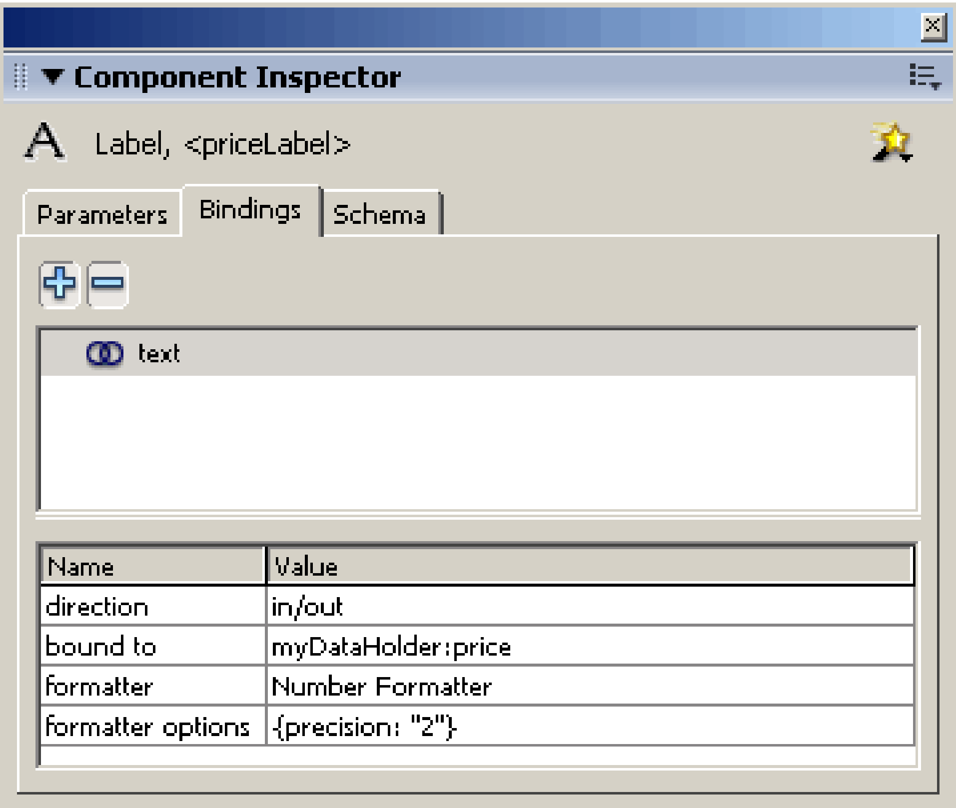



**Figure 11.** The Component Inspector panel shows the Bindings settings for the priceLabel Label component.

11. Similarly, create a binding from the value property of the NumericStepper to the quantity property on the DataHolder.

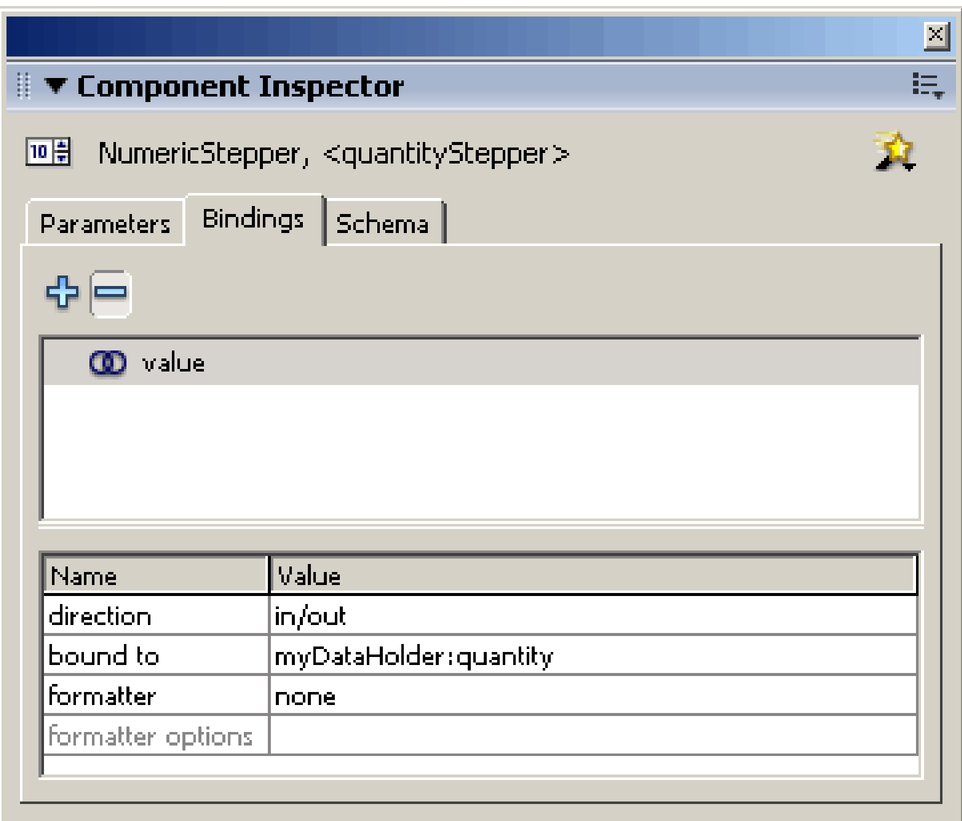


**Figure 12.** Bindings settings for the quantityStepper NumericStepper component.

12. Create a binding from the text property of the vatLabel Label instance to the vat property of the DataHolder.

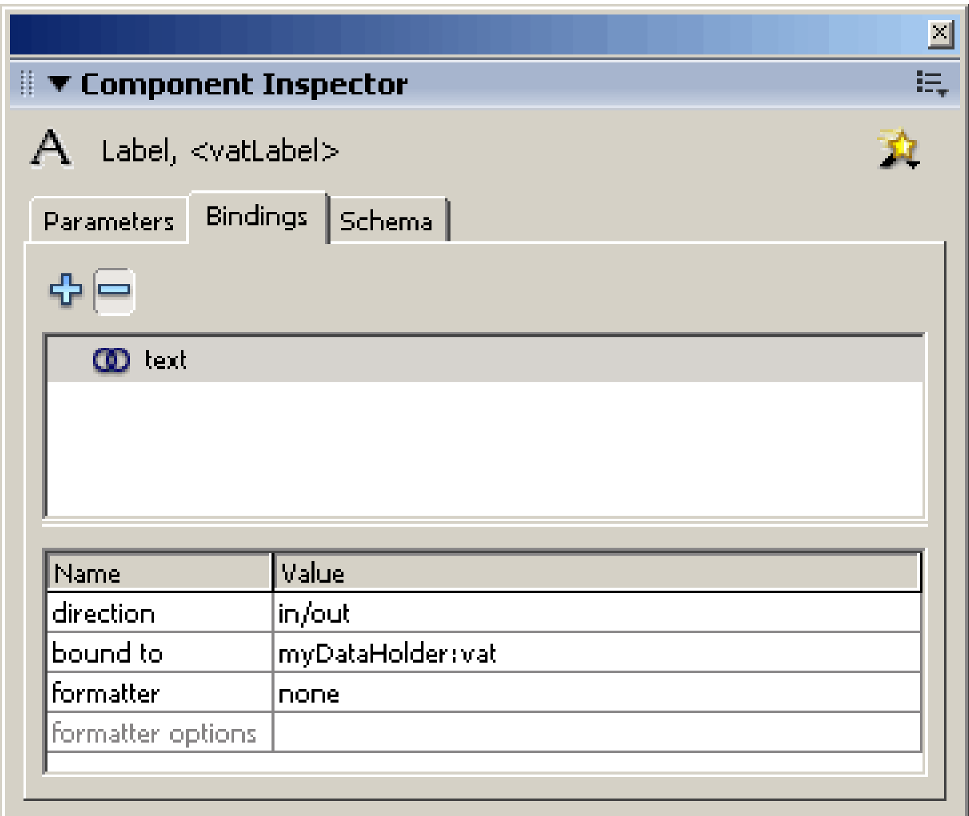



**Figure 13.** Bindings settings for the vatLabel Label component.

13. Finally, create a binding from text property of the totalLabel Label to the totalPrice property of the DataHolder.

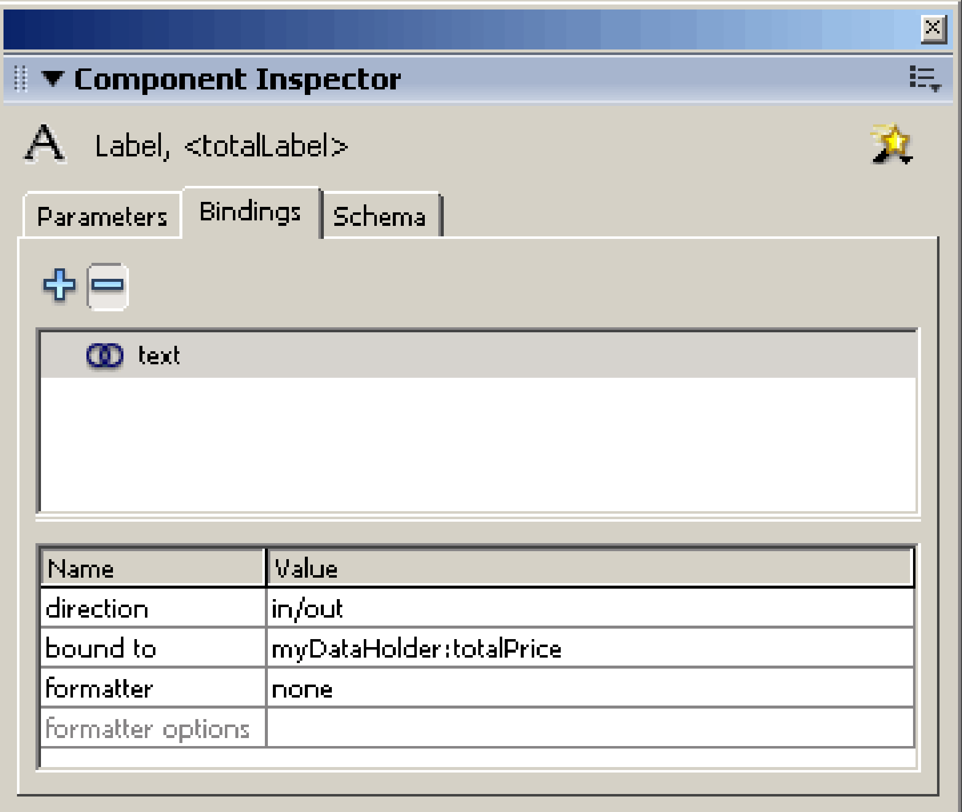



**Figure 14.** Bindings settings for the totalLabel Label component.

14. Now all of your bindings are in place. However, you still need to do two things: One, you need to set the initial values for price and vat, and two, you need to listen for the event broadcast by the DataHolder when the quantity stepper changes.

    So what is this event? Simple, it is the same as the name of the property, in other words, the event, in this case, will be called *quantity*. This is how the DataHolder works: When one of the properties in its schema changes, it broadcasts an event with the name of the said property. Elegant, no?

    To capture the event and set up the initial values on the DataHolder, you need to type some code. Create a new layer called Actions. Enter the following script on Frame 1 of the new layer:

```
// turn on the debugger (that's two underscores before dataLogger) _global.__dataLogger =
new mx.data.binding.Log(); // onQuantityChange listener function onQuantityChange
( eventObj : Object ) : Void { // save a local reference to the event object's target //
(which points to our DataSet) var myDataHolder = eventObj.target; // get the price,
quantity and VAT from the DataSet var price = myDataHolder.price; var quantity : Number =
myDataHolder.quantity; // ok, I don't think there's really 17.5% VAT on bread, // but what
the heck! var VAT : Number = myDataHolder.vat; // calculate total price, including VAT var
grossPrice : Number = price * quantity; var vatOnPrice : Number = grossPrice * VAT / 100;
var totalPrice : Number = grossPrice + vatOnPrice; // set the totalPrice calculated field
in the DataSet myDataHolder.totalPrice = totalPrice; } // listen for the quantity event
myDataHolder.addEventListener ( "quantity", onQuantityChange ); // setup initial values for
data holder myDataHolder.price = 0.69; myDataHolder.vat = 17.5; myDataHolder.quantity = 1;
```

Go ahead, test the movie, and play around with the quantity stepper to see the Total Price update in

response to your actions. Although this is great, you can go one step further. For one thing, in the screenshot at the start of the example, you can see that there's a static text field with a Pound Sterling symbol (the unit of currency in the United Kingdom) in front of the priceLabel Label. You could do the same thing for the Total Price label, but where's the fun in that? Instead, follow the directions in the next section to create a custom formatter to append the Pound Sterling symbol to the label when updating the totalPrice property in the DataHolder.

## Using a Custom Formatter

Custom formatters are very useful for managing your data display. In this case, you are going to use one to prefix the totalPrice Label component with a Pound Sterling symbol and to display the numeric portion of the data in currency format (to 2 decimal places.)

Continuing where you left off in the previous example, you're going to specify a custom formatter for the totalPrice property in the DataHolder:

1. Select the DataHolder instance on stage and bring up the Component Inspector panel's Schema tab.
2. Click the totalPrice property to select it and choose Custom Formatter as its formatter. In the formatter options, enter the class name MyFormatter**.**
3. Create the MyFormatter class. Select File > New and select ActionScript File to create a new ActionScript document. Save it as **MyFormatter.as** in the same directory as your FLA.
4. Enter the following script in the MyFormatter.as file:

```
class MyFormatter { function MyFormatter () { trace ("MyFormatter initiated!"); } function
format () { // get the value var value : Number = arguments [ 0 ]; // format the number to
2 decimal points var formattedValue:String = formatPrecision ( value, 2 ); // add the
currency symbol formattedValue = "£" + formattedValue; // return the formatted value return
formattedValue; } function formatPrecision ( num:Number, precision:Number, splitCharacter:
String ) : String { // formats a number to given precision // thanks to bokel for the
original algorithm on which this method is based var precision:Number = (precision = Math.
abs(precision)); if( precision == 0 ) { // no decimal points var returnValue = String
( Math.round(num) ); return returnValue; } if ( splitCharacter == null ) { // setup default
split character splitCharacter = "."; } // calculate precision var returnValue:String =
String ( Math.floor(num) + splitCharacter + Math.floor(num * Math.pow( 10, precision)).
toString().substr(-precision) ); return returnValue; } }
```

5. So what's going on here? It's actually all quite simple. After you specify a custom formatter class, the data binding system calls its format method automatically whenever the property updates. Macromedia Flash passes the value of the changed property (in this case, the totalPrice property) as the first argument to the format method. You then convert the value to two decimal places (using the formatPrecision method) and return the formatted value.

   Test the movie and play with the NumericStepper. See how the movie now updates the Total Price area, formatting the value as currency with a Pound Sterling symbol at the head.

**Figure 15.** Interface with custom Total Price formatting.

### Using a Custom Validator

When creating a form that has TextInput or TextArea components, you often want to validate the input of your users immediately after they entered it. This enhances usability, as it lets you catch erroneous input before the users submit a form. You can warn them about it without breaking their flow.

The components in Macromedia Flash MX Professional 2004 help you do this with validators. Some data types have their own built-in validators, such as Numbers, for which you can choose an upper limit and a lower limit to validate against. For data types without their own validators or to validate to criteria not supported by the default validator, you can create a custom validator by setting the data type of your property to Custom and specifying a class to use as your custom validator.

## Example: Custom Validator

Now you'll modify the Organic Bakery example to add a custom validator. Since it does not make sense to validate the NumericStepper component (though you can and should set its limits in the Component Inspector panel's Parameters tab so that it does not let the user select an invalid quantity), you're going to start by adding a new field to your form. Follow the instructions below, using the screenshot below of the completed app as your guide:

**Figure 16.** Interface with a custom e-mail validator.

1. Enter some text in a new static text field to guide the user: **If you would like to learn the origins of this bread, please enter your e-mail address below:**
2. Drag a TextInput component onto the Stage and give it the instance name **emailText** in the Property inspector.
3. Create a regular text field to label the TextInput component. The text should read **E-mail:.**
4. If the user enters an invalid e-mail address, you will want to bring it to his attention, but you don't want to break up his user experience flow. The best way to do this is to flag the invalid field on the form itself. (The worst way is to show an alert window, thus ruining the user's flow. In a real application, there would also be a check when the user submits the form that would then alert the user in a stronger manner.)

   Create a dynamic text field and give it the instance name **invalidEmail_txt**.

5. Double-click the dynamic text field you just created and type the message you want the user to see when he or she enters an invalid e-mail address, for example, "Invalid!"

6. Create a new property in the DataHolder and map it to the text property of the emailText TextInput component instance.

   To do this, click the DataHolder on the Stage to select it and bring up its Schema tab in the Component Inspector panel. Add a new component property and give it a field name of **email**. Set its data type to Custom and set the classname of the Custom Validator (next to validation options) to **MyCustomValidator**.

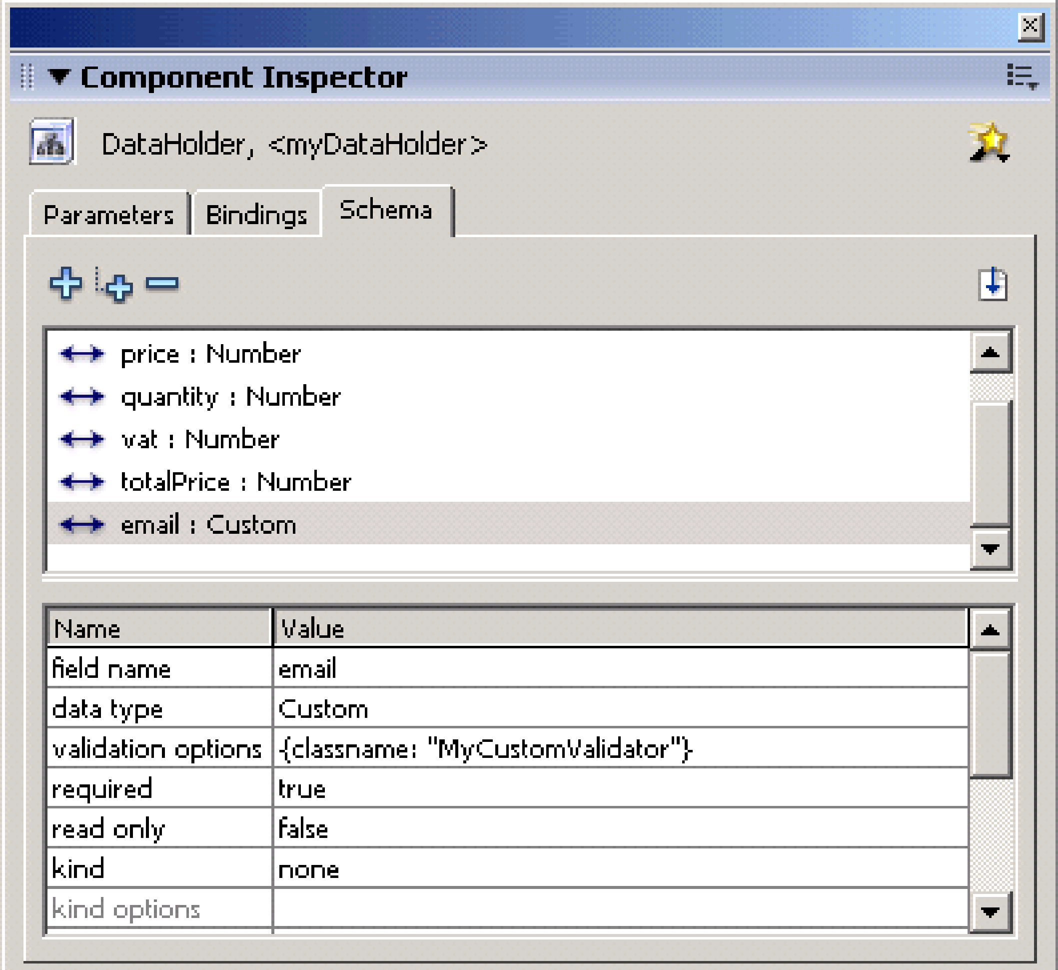



**Figure 17.** Component Inspector panel showing the email property of the myDataHolder instance of the DataHolder component.

7. Before you write the custom validator class, create the binding between the TextInput component and the DataHolder.

   With the DataHolder component still selected, open the Bindings tab in the Component Inspector panel. Use the Add Binding button to add a binding from the email property of the DataHolder to the text property of the emailText TextInput component instance.

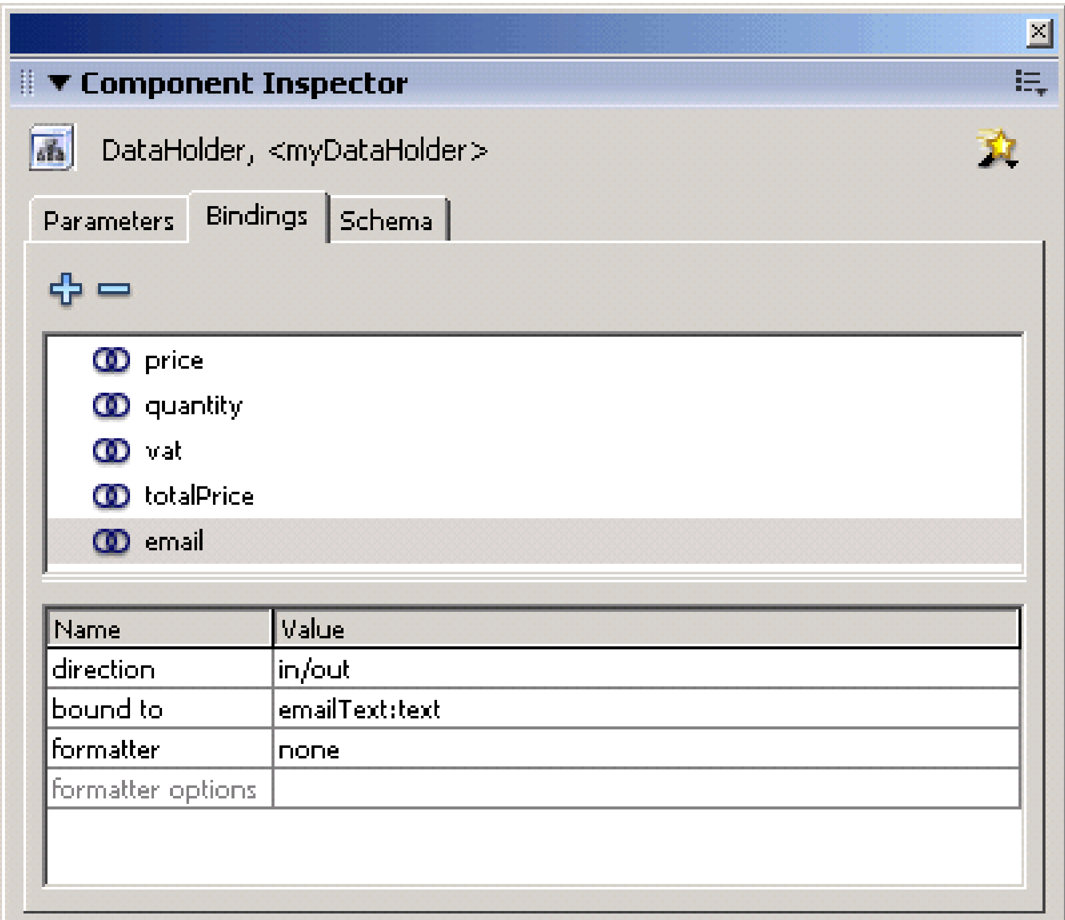



**Figure 18.** Bindings tab showing the email property of the DataHolder component bound to the emailText property of the TextInput component.

8. Create a new ActionScript document, go to File > New > ActionScript File, and save it as **MyCustomValidator.as** in the same directory as your FLA. Enter the following code in that file and save it:

```
class MyCustomValidator extends mx.data.binding.CustomValidator { function
MyCustomValidator () { trace ("MyCustomValidator initiated!"); } function validate () { //
simple syntax check the entered email address var address:String = arguments [ 0 ]; var
atFirstIndex:Number = address.indexOf("@"); var atLastIndex:Number = address.lastIndexOf
("@"); var dotFirstIndex:Number = address.indexOf("."); var dotLastIndex:Number = address.
lastIndexOf("."); // validation flag var emailValid:Boolean = true; // @ check emailValid =
atFirstIndex != -1; emailValid = emailValid && atLastIndex != -1; emailValid = emailValid
&& atFirstIndex == atLastIndex; // . check emailValid = emailValid && dotFirstIndex != -1;
emailValid = emailValid && dotLastIndex != -1; emailValid = emailValid && dotFirstIndex ==
dotLastIndex; trace ("email is: "+emailValid ); if ( ! emailValid ) { this.validationError
( "Email not valid" ); } } }
```

9. A custom validator has to extend the class mx.data.binding.CustomValidator. This is the class that contains the method we call to raise a validation error. The code above is very simple. (It's not a very good candidate for a real-world e-mail validator, even for syntax checking alone! But it's good for this example.) It checks to see that there is both an at sign (@) and a period (.) in the entered string and, if not, it calls the `validationError()` method to throw a validation error.

Now that you have a custom validator, you need to add a listener to the DataHolder to listen for valid and invalid events and to hide (or display) the "Invalid" notice next to the e-mail field accordingly. First, add the code to create and activate your `onInvalid` event listener. Add the following code to the script in Frame 1 of your FLA:

```
function onInvalid ( eventObj : Object ) : Void { var INVALID_EMAIL_MSG:String = "Email not valid"; var invalidProperty:String = eventObj.property; var errorMessages:String = eventObj.messages; switch ( invalidProperty ) { case "email": // define loop limit var errorMessagesLength:Number = errorMessages.length; for ( var i = 0; i < errorMessagesLength; i++ ) { // current error message var errorMessage = errorMessages [ i ]; // check the message to see if its our email error if ( errorMessage == INVALID_EMAIL_MSG ) { // email error invalidEmail_txt._visible = true; } } break; default: trace ("Unknown invalid property: " + invalidProperty + "(" + errorMessages[0] + ")" ); } } myDataHolder.addEventListener ( "invalid", onInvalid );
```

10. The `onInvalid` listener gets called with an event object (`eventObj`). You first check to see that the property that triggered the invalid event is the email property and, if so, you loop through the array of error messages to see if your error message is among them.

    **Note:** Since your custom validator currently only raises one error, this is not necessary, but in real life your validators will probably raise more than one error. This is a good example of how you would handle such a situation.

    Finally, when you're convinced that the e-mail is invalid, you turn on the visibility of the dynamic text field next to the email field to display the warning "Invalid!"

    If you stopped here, the "Invalid!" notice would never disappear, even if the user were to correct the text in the email field. To close it, you must listen for the valid event and make the dynamic text field invisible.

    Enter the following code underneath the previous snippet:

    ```
    function onValid ( eventObj : Object ) : Void { if ( eventObj.property == "email" ) { // ok, a valid email was entered // make sure we hide the invalid email sign invalidEmail_txt._visible = false; } } myDataHolder.addEventListener ( "valid", onValid );
    ```

11. Finally, you want to hide the "Invalid!" message when the form first displays, so add the following line into the frame script in the first frame of the FLA :

    ```
    // hide invalid email alert at startup invalidEmail_txt._visible = false;
    ```

12. Test the movie. Try entering some text without an @ sign in the e-mail field and tab out of the field. The "Invalid!" notice will light up.

## Conclusion

The version 2 Macromedia Flash MX Professional 2004 Component Framework sports lots of nice new features, such as CSS Styles and a more standards-based event system. One of the biggest new features is data binding, inherited from the Firefly components introduced with Flash MX.

As you have seen in the examples here, using data binding to tie components together can be a real time-saver when developing Rich Internet Applications. Data binding saves you from having to program that functionality yourself.

In this introductory tutorial, you learned how to tie UI components together as well as how to use a common data repository, the DataHolder, to handle more complex interactions. You also saw how to use formatters to affect the display of data, including how to write your own custom formatter.

There is a lot more about data binding that just wasn't possible to cover in this tutorial. Going forward, look into the other features provided by data binding such as the various connectors and resolvers (for connecting to external data) and the validation options. A central component not covered here is the DataSet, which you can think of as the big brother of the DataHolder. Unlike the DataHolder, the DataSet keeps change information about the data and you can use it in with the connectors and resolvers (not to mention the new Screens/Forms functionality) to create efficient applications that work with external data.

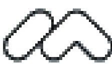
Also, it is possible to tie Flash Remoting RecordSets to the v2 components, but that, again, falls outside the scope of our tutorial.

I hope you have enjoyed this introduction and that it has whetted your appetite to learn more and use data binding in Macromedia Flash MX Professional 2004.

---

### About the author

Aral Balkan is a user interface and usability consultant who has been using Macromedia Flash for as long as he can remember. Alongside managing his London-based new media consultancy business, Bits And Pixels, Aral is also active in the Macromedia Flash community. He is the Director of Education Content on the Macromedia Flash resource site Ultrashock.com and co-director of the London Macromedia User Group. Aral also has taught graduate and undergraduate-level multimedia and web design classes in the United States and looks forward to one day finding enough time to do some further teaching in the United Kingdom. He is a published author, having contributed to Flash MX Most Wanted Components (Friends of Ed), along with tutorials for his community blog, onRelease.org, Ultrashock.com, and Macromedia Developer Center. His latest passions include Agile development methodologies, patterns, user-centric product development, and usability engineering. One day, he hopes to get his head around Java!



Company | Site Map | Privacy & Security | Contact Us | Accessibility | Report Piracy | Send Feedback

©1995-2003 Macromedia, Inc. All rights reserved.
Use of this website signifies your agreement to the Terms of Use.

Search powered by Google