Data integration

Adobe® Flash® CS3 Professional provides a flexible, ActionScript™ 2.0 component-based architecture and object model for connecting to external data sources, binding data to user interface (UI) components, and managing what data is displayed and how it's updated at the source.

The Adobe website has many tutorials on creating rich Internet data applications in Flash. For downloadable examples and tutorials that use the data components, see "Additional resources" on page 389.

This article begins with an overview of data integration, provides a quick example you can walk through to become familiar with how data integration works, provides general workflows, and then explains data binding, which is the core functionality of the Flash data integration architecture, and the other layers in the Flash data integration architecture.

NOTE	The Flash data binding layer supports ActionScript 2.0 only. If you are using ActionScript 3.0, you will need to manually program all of your application's data interactions.

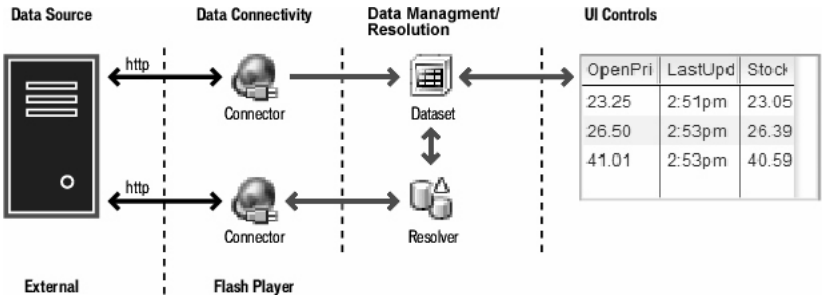
There are four main layers in the Flash data integration architecture:

- The data binding layer provides a way to map data elements to properties of Flash ActionScript 2.0 data components, which can then be mapped to UI components. In other words, you bind to a data source and then select the elements you need to display in your application and to update the source. Flash also integrates objects such as formatters and encoders to let you control how data is propagated and formatted among components. See "Data binding" on page 393.
- The data connectivity layer provides connector components that let you connect to an external data source to send and receive data. You can connect to a variety of sources, such as web services and XML. For more information, see "Data connectivity" on page 411.
- The data management layer provides a component that enables intelligent supervision of common data operations, such as editing, sorting, filtering, aggregation, and translation of changes. For more information, see "Data management" on page 418.

- The data resolution layer provides resolver components that can translate changed data into a format that is consumable by an external data source. In addition, these components can accept and translate updates from an external data source so that they can be consumed by a Flash client. For more information, see "Data resolution" on page 426.

When you integrate external data into a Flash application, you connect to the external data, select different elements of the data schema that you need for your application, and bind them to component fields within your application. You manage how the data is displayed in your application and how it's updated on the server.

The following image depicts the flow of data within a Flash application and identifies the different elements that comprise the Flash data architecture. Data binding is represented by the red arrows between the components. As shown in the diagram, you will need to set up data bindings between properties of UI controls and properties of a DataSet component; between the DataSet component and a connector component; between the DataSet component and a resolver component; and between a resolver and a connector component.



Typically, you add data components to the Stage in a Flash document. (See "Workflows for using the data components" on page 392 and in each component entry in Components Help.) The data components have no visual appearance in a runtime application. If you prefer, you can also create and access the data components through ActionScript code, although you may still need to perform some tasks through the Flash interface. To work with data binding classes in ActionScript instead of in the Flash interface, see "Making data binding classes available at runtime" in the *Components Language Reference*.

The following table can help you decide what components you need to use in your Flash data application.

Data source	Use this connector	Use this resolver
web service/SOAP	WebServiceConnector WebService classes (not a component)	XUpdateResolver WebService classes (not a component)

Data source	Use this connector	Use this resolver
XML document	XMLConnector	XUpdateResolver
SQL data	WebServiceConnector	RDBMSResolver

Flash is a client-side technology. To create a Flash application that integrates with a data source, you will need to implement server-side code as well. Building and exposing business logic on the server is the job of a server developer and is best implemented using products that are specifically designed for that task, such as ColdFusion®, J2EE Application Servers, and ASP.NET. For information on server-side tasks and other tasks that might best be addressed by a database administrator, see "Advanced topics in data integration" on page 430.

Additional resources

The following table outlines additional resources that are available for learning to use the data integration components in Flash.

Component	Data tutorials on the Flash Tutorials page	Data tutorials on the Developer Center
WebServiceConnector	Web Service Tutorial: Adobe Tips www.adobe.com/go/learn_fl_tutorials	Tip of the Day, Part 2, www.adobe.com/go/learn_fl_tipoday_pt2 Building a Google Search Application, www.adobe.com/go/learn_fl_google_search
XMLConnector	XML Tutorial: Timesheet www.adobe.com/go/learn_fl_tutorials	Timesheet Tutorial: Bike Trips Sample, www.adobe.com/go/learn_fl_xmlconnector Data Integration Using ASP, www.adobe.com/go/learn_fl_flashpro_asp
XUpdateResolver	XUpdate Tutorial: Update the Timesheet www.adobe.com/go/learn_fl_tutorials	---
RDBMSResolver	---	Time Entry Application, www.adobe.com/go/learn_fl_time_entry Data Integration Using ASP, www.adobe.com/go/learn_fl_flashpro_asp Using the RDBMSResolver Component to Update a Database, www.adobe.com/go/learn_fl_delta_packet

Creating a simple application

The following example walks you through creating a simple data integration application, which can help you understand the concepts and steps involved.

In the example, you create a simple application that loads and displays a dinner menu. You load an XML file, which you'll use both as a data source and as a sample of the data source's schema (structure). The UI consists of a data grid, into which the XML data is loaded, and a button that loads the data. Data binding is supported only between components that exist in Frame 1 of the main Timeline, Frame 1 of a movie clip, or Frame 1 of a screen; in this example, the components all reside in Frame 1 of the main Timeline.

NOTE	Note that all of the namings in this example are case-sensitive.

To create the dinner menu application:

1. Download and decompress the Samples zip file from www.adobe.com/go/learn_fl_samples and navigate to the DataIntegration\DinnerMenu folder to access the sample file, dinner_menu.xml.
2. Copy the dinner_menu.xml file into a new folder.
3. In Flash, create a new Flash document and save it as **dinner_menu.fla** in the folder you created in the previous step.
4. Create the user interface, which consists of two ActionScript 2.0 components—a button that triggers data retrieval and a data grid to display the data:
   a. From the Components panel, open the User Interface category and add a DataGrid instance named `menu_dg` to the Stage with width **540** and height **240**.
   b. Also from the Components panel, add a Button component instance named `loadData` below the data grid labeled **Load Data**.
5. Add the data components—an XMLConnector component to connect to the dinner_menu.xml file and a DataSet component to bind that data to the data grid:
   a. Add an XMLConnector component instance named `xmlConn`.
   b. Add a DataSet component instance and name it `menu_ds`.

   The data components do not have to be on the Stage; they don't appear at runtime.
6. Set parameters for the XMLConnector component: select the XMLConnector component instance and, in the Component inspector, click the Parameters tab, enter `dinner_menu.xml` for the URL, and select `receive` for the direction. (Because the XML file is in the same folder as the FLA file, the fully qualified path is simply the XML filename.)

7. Load a sample of the data source's schema: with the XMLConnector instance still selected, in the Component inspector click the Schema tab and follow these steps:
   a. Select `results : XML` from the Schema tab top pane.
   b. Click the Import a Schema from a sample XML file button.
   c. Select the dinner_menu.xml file from the dialog box that appears.

   The XML file's schema structure appears in the Schema tab.
8. Expose the XMLConnector's `array` property for data binding and bind it to the DataSet's `dataProvider` property. With the XMLConnector component selected, follow these steps:
   a. On the Bindings tab of the Component inspector, click the plus (+) sign and in the dialog box, select `food:array`.
   b. On the Bindings tab again, click Bound To, click the magnifying glass icon, select DataSet, and select `dataProvider:Array`.

   Each time you create a binding, you perform at least these two basic steps.
9. Populate the data grid with the XML data by binding the XML data—through the DataSet component—to the data grid. Select the DataSet component and click the Bindings tab. You see the binding to the `xmlConn` instance that you just added. Now, add two new bindings:
   a. Bind the DataSet's `dataProvider` property to the DataGrid's `dataProvider` property: click the plus (+) sign, select the `dataProvider:Array` property, click Bound To, click the magnifying glass icon, select DataGrid, then select the `dataProvider:Array` property. Select `out` for the direction.
   b. Bind the DataSet's `selectedIndex` property to the DataGrid's `selectedIndex` property: click the plus (+) sign, select the `selectedIndex:Number` property, click Bound To, click the magnifying glass icon, select DataGrid, then select the `selectedIndex:Number` property.
10. Set up the button to load data into the data grid. Click layer 1 in frame 1 of the Timeline and open the Actions panel. Add the following code to the first frame:

```
form = new Object();
form.click = function(eventObj){
  xmlConn.trigger();
}
loadData.addEventListener("click", form);
```

11. Save and test the application. Click Load Data. The data from the XML file is loaded into the DataGrid.

You've just created your first data integration application, with data dynamically loaded from an XML file. To add more functionality to this application, see “Creating an indexed binding” on page 406.

Workflows for using the data components

This section provides a high-level overview of the steps required to create a Flash application that can dynamically interact with an external data source. You can find instructions and examples to complete each step throughout the rest of the article.

There are two general workflows: one for connecting to web services or XML documents as your data source, and one for connecting to an external database.

Workflow for data source from web services or XML documents:

1. Get the URL of your external data source:
   - A web service.
   - An XML document.
2. Add components to the Stage:
   - Add a connector component.
   - Add a DataSet component, which you will bind to your data source and UI components.
   - Add UI components that will display data to users, such as a DataGrid component.
   - Add a resolver component.
3. Set up the connector component:
   - Set component parameters.
   - Set component properties on the Schema tab.
4. Bind the connector component to DataSet component.
5. Set up the DataSet component:
   - Set component parameters.
   - Set component properties on the Schema tab.
6. Bind the UI component to the DataSet component.
7. Set up the resolver component:
   - Set component parameters.
   - Set component properties on the Schema tab.
8. Bind the resolver component to the DataSet component.

9. Add additional UI components and code for the resolver functionality (that is, for adding, editing, or deleting data records).
10. Bind UI components to resolver components.

Workflow for an external database (non-XML or not a web service):

1. Set up your data source; for example, in the ColdFusion environment, set up a ColdFusion DataSource component to connect to your data source.
2. Add components to the Stage:
   - DataSet component.
   - UI component for data display, such as DataGrid.
   - Resolver component.
3. Bind the DataSet component to the UI component for data display.
4. Set up DataSet component:
   - Set component parameters.
   - Set component properties on the Schema tab.
5. Set up a connection to your data; for example, you could set it up through a ColdFusion component with Flash Remoting services and your own ActionScript code.
6. Bind the resolver component to the DataSet component.
7. Set up the resolver component:
   - Set component parameters.
   - Set component properties on the Schema tab.
   - Write ActionScript code using methods of a resolver component class.
8. Add additional UI components and ActionScript code for the resolver functionality (that is, for adding, editing, or deleting data records). Bind UI components to resolver components.

Data binding

Data binding lets you map the properties of one component to another component. A binding is simply a statement that says "When property X of component A changes, copy the new value to property Y of component B."

For rich Internet applications, you can map data from external data sources to Flash components. The external data source is represented in your application by a component; items in the data source's schema are represented as properties of the component. You can define component properties to meet your business needs; these properties, which contain dynamic data that you want to manipulate, are referred to as *bindable* properties.

The most powerful use of data binding in Flash is to define the flow of data between UI components, data management components, and connector components that access external data sources such as web services, XML documents, and relational databases.

In the Flash interface, you bind data by using the Bindings and Schema tabs of the Component inspector. Although you need to understand how bindings and schemas work in Flash, your connector component is usually the first component you need to set up, because it brings in the schema for your data source; see "Data connectivity" on page 411.

Data binding is supported only between components that exist in Frame 1 of the main Timeline, Frame 1 of a movie clip, or Frame 1 of a screen.

You can also create runtime bindings by writing ActionScript code. For more information, see "Data binding classes" in the *Components Language Reference*.

A simple binding example

The following procedure provides a simple illustration of how data binding connects one UI component to another. In the example, the value properties of component instances `stepper1_nm` and `stepper2_nm` are bound to each other, and the value properties of `stepper3_nm` and `myInput_txt` are bound to each other. In a real-world application, you would most likely import a schema, define additional bindable component properties, and create multiple bindings between data components and UI components.

To connect UI components to create data binding:

1. Add a NumericStepper component to the Stage, and name it `stepper1_nm`.
2. Add another NumericStepper component, and name it `stepper2_nm`.
3. With `stepper1_nm` selected, open the Component inspector, and click the Bindings tab.
4. Click the Add Binding (+) button to add a binding.
5. In the Add Binding dialog box, select Value, and click OK.
6. In the Name/Value section at the bottom of the Bindings tab, click the Bound To item under Name, and click the magnifying glass icon across from the Bound To item under Value.
7. In the Bound To dialog box, select component `stepper2_nm` under Component Path, and click OK.

8. Select Control > Test Movie. Click the Up and Down buttons on component `stepper1_nm`.

   Each time you click the buttons on `stepper1_nm`, the `value` property of `stepper1_nm` is copied to the `value` property of `stepper2_nm`. Each time you click the buttons on `stepper2_nm`, the `value` property of `stepper2_nm` is copied to the `value` property of `stepper1_nm`.

9. Return to editing the application.

10. Add another NumericStepper component and name it `stepper3_nm`.

11. Add a TextInput component called `myInput_txt`.

12. Repeat steps 4-7 and bind the `value` property of `stepper3_nm` to the `text` property of `myInput_txt`.

13. Select Control > Test Movie. Type a number in the text input field, and press Tab.

    Each time you enter a new value, the `text` property of `myInput_txt` is copied to the `value` property of `stepper3_nm`. When you click the Up and Down buttons on `stepper3_nm`, the `value` property of `stepper3_nm` is copied to the `text` property of `myInput_txt`.

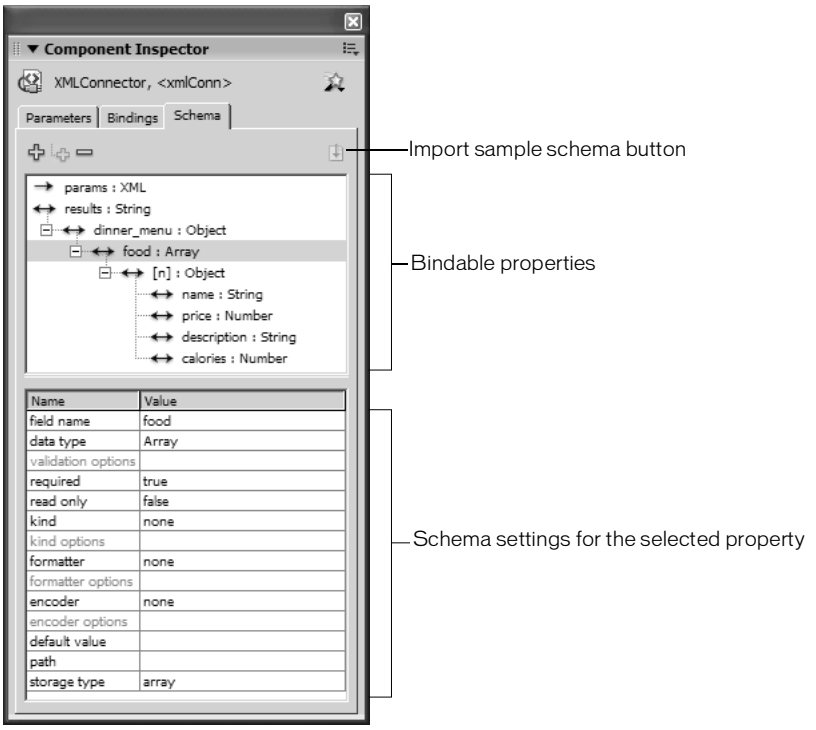
For more tutorials that show you how to create data bindings, see www.adobe.com/go/learn_fl_data_integration.

Working with schemas in the Schema tab

The Schema tab in the Component inspector lets you view and edit the schema for each data-related component in your application. The Schema tab lists the component's *bindable properties*, which are properties to which you can bind that commonly contain dynamic data. All components have properties, but by default, to reduce UI clutter, the Schema tab shows only properties that commonly contain dynamic data. (You can, however, bind to any property by either adding it to the Schema tab or using ActionScript code. For more information, see "Working with bindings in the Bindings tab" on page 402.)

The Schema tab also lists properties' data types, their internal structure, and various special attributes. The data binding engine needs this information for each component to handle your data correctly.

The following illustration shows the Schema tab for the XMLConnector component used in "Creating a simple application" on page 390. The top pane shows the bindable properties for the `xmlConn` instance, with the `food:Array` property selected, and the bottom pane shows the settings for the `food:Array` property.



A component's schema describes the structure and type of data but is independent of how the data is actually stored. For example, the results from a WebServiceConnector component or an XMLConnector component could have identical schemas, even though the web service results are stored as ActionScript data structures (objects, arrays, strings, Boolean values, and numbers), and the XMLConnector component results are stored as XML objects. When you use data binding to access fields within a component's schema, you use the same procedure regardless of how the data is stored.

A component identifies which of its properties are bindable. These bindable properties appear in the Schema panel as top-level schema items (component properties). A component property can have its own internal schema that defines additional schema fields that can be bound to other component properties within your application; for example, when you introspect a WSDL for a WebServiceConnector component. The WSDL definition describes the parameters and the results for a web service. The WebServiceConnector component contains two bindable properties (`params` and `results`). When the WebServiceConnector component introspects the WSDL, Flash automatically creates the schema for the `params` and `results` properties so it mirrors the schema defined within the WSDL.

There are several ways to define the schema for a component. Here are the most common ways:

- For an XMLConnector component, you can import an XML sample file to define the schema. See "Connecting to XML data with the XMLConnector component" on page 415.
- For a WebServiceConnector component, you can import the WSDL for a web service to define the schema. See "Connecting to web services with the WebService connector component" on page 412.
- For a DataSet component, which is typically the intermediary component between your connector components and UI components, you define the schema using the Schema panel. See "Adding a component property to a schema" on page 397 and "Adding a schema field to a schema item" on page 399.
- For UI components, the schema is predefined within the component. You can modify the schema to create additional bindable properties, as shown in "Adding a component property to a schema" on page 397.

Adding a component property to a schema

You typically add component properties to a schema for the following reasons:

- To make an existing component property bindable. You can make any component property bindable if you add it to the schema.
- To define the fields of a DataSet component to describe expected data fields. Most commonly, you need to define the data type for an expected field, but there are numerous other properties you can set. For more information, see the examples in "Accessing the data" on page 423 and "This section discusses advanced topics, such as refinements you make to schema settings and information for developers who need to write server-side code to interact with Flash data applications.Schema item settings" on page 430.

The following example illustrates how you make an existing component property bindable by adding the component property to the component's schema. In the example, you create an application that uses a CheckBox component to indicate whether a TextInput component is editable. Because the TextInput component's schema does not initially contain the `editable` property, you add the `editable` property to the schema to bind it to the CheckBox component.

To add a component property to a schema to make the property bindable:

1. Add a TextInput component and a CheckBox component to your application, and give them instance names.
2. Select the TextInput component, and click the Schema tab on the Component inspector.
3. Click the Add a Component Property (+) button at the upper left of the Schema tab to add a component property.
4. In the Schema Attributes pane (the bottom pane of the Schema tab), enter **editable** for the field name value and select Boolean for the data type value.
5. Click the Bindings tab, and click the Add Binding (+) button to add a binding.
6. In the Add Binding dialog box, select the `editable` property, and click OK.
7. In the Binding Attributes pane at the bottom of the Bindings tab, click the Bound To item under Name, and click the magnifying glass icon across from the Bound To item under Value.
8. In the Bound To dialog box, select the CheckBox component under Component Path, and click OK.
9. Select the Checkbox component on the Stage, and click the Parameters tab in the Component inspector.
10. Select Control > Test Movie. To test the functionality, type a value into the TextInput component and then deselect the CheckBox component. You should now be unable to enter text into the TextInput component.

Adding a schema field to a schema item

When you use a DataSet component, you manually enter the schema for the component. You might need to add schema items, which are essentially component properties (see “Adding a component property to a schema” on page 397). You may also need to add additional fields within a schema item to provide a deeper level of bindable detail. For more information, see “This section discusses advanced topics, such as refinements you make to schema settings and information for developers who need to write server-side code to interact with Flash data applications.Schema item settings” on page 430.

To add a schema field to a schema item:

1. In the Schema tab, select the schema item to which you want to add a field.
2. Click the Add a Field Under the Selected Field (+) button.

   A new field is added as a subfield of the selected property.
3. In the Schema Attributes pane, enter a value for Field Name. Fill in the other attributes as appropriate.

   There are three possible scenarios based on the type of schema item:

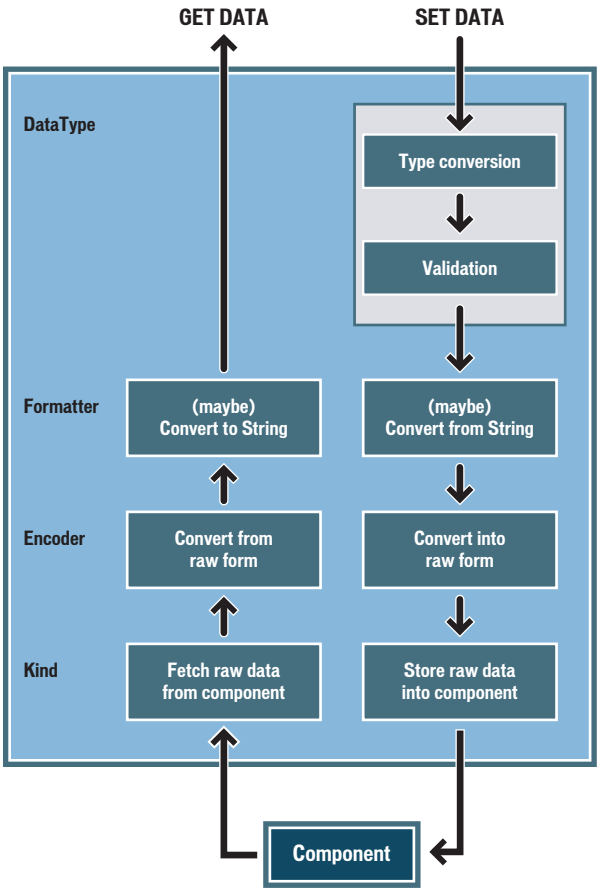
   - Schema item of type Object, which can have subfields, attributes, or both. Attributes are preceded with @ in the list.
   - Schema item of type Array, which has one subfield called [n] representing the index of the array, which can be of any type (including Object, String, and so on).
   - Schema item of other types (such as Boolean, String, Number), which don’t have subfields but can have attributes. Attributes are preceded with @ in the list.

About handling data types in data binding

Your data source’s schema is represented on the Schema tab in the Component inspector. Every item in the schema has many attributes that you can configure in the lower pane of the Schema tab. In particular, four attributes control the handling of data types as data flows in and out of Flash applications. These four attributes are Data Type, Encoder, Formatter, and Kind.

You might not need to change the settings of these attributes from the default values. However, in situations where you’re working with complex data types, you might need to change the values of these attributes so Flash receives and outputs data in the correct format. See “When to edit schema item settings” on page 442.

The following illustration shows the runtime process of the data binding engine. The four attributes that handle data types are shown in the illustration and discussed in the following text.



Kind When Flash wants to get data from a component, the data is fetched from the component according to the Kind setting. At this point, the data is in whatever format the component provides (the raw form of the data). For example, the XMLConnector component always provides data as a string, the NumericStepper component provides data as a Number, and so on.

Encoder The encoder's job is to convert this data to an ActionScript data type. For example, the string data that you get from an XML document can represent a date or a number. If data binding needs the data in string form (because it is being assigned to a text component, for example) the formatter does this conversion. If there are several bindings from a field, the formatter is used only for those bindings that are assigning to a field whose type is String.

Data type and formatter When you want to set data into a component, data binding first needs to convert the data to an ActionScript data type, which is a form that the component can read; this conversion is automatic, depending on the Data Type setting. If the data is a string and a Formatter setting exists, then the formatter converts the data from string to the specified ActionScript data type. The Data Type setting also controls whether the data binding engine inspects the data to see if it's valid and causes events to be generated accordingly. The encoder then converts the data from the ActionScript-readable form to raw form, and the kind then finally passes the data to the component.

The processing handled by these four attributes occurs when the data field is accessed with data binding. It is possible to directly access a component property from your ActionScript code, but when you do this, you're working with the raw value of the data, not the data value that results from the action of data types, encoders, formatters, and kinds. For more information, see "DataType class" in the *Components Language Reference*.

In many cases, you won't need to edit the settings that are in the bottom pane of the Schema tab. The following guidelines specify when to change the schema item settings from their default values:

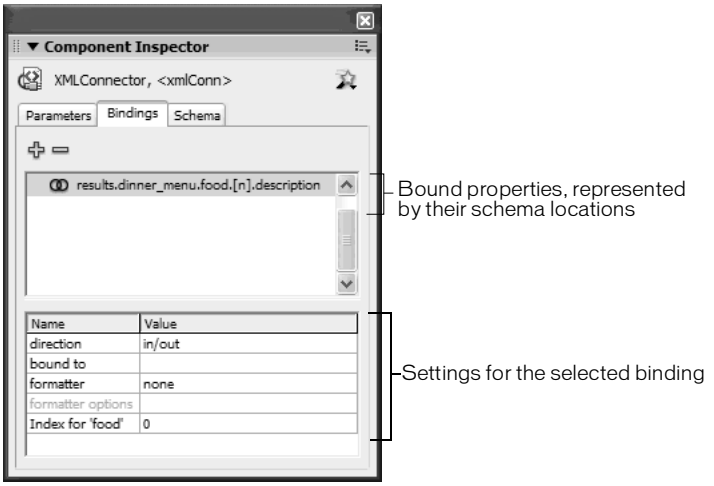
- You always need a kind. The default value for the kind setting is `none`, which is equivalent to the Data kind.
- You need an encoder when the component does not provide the data in the form you want. The most common case is the XMLConnector component, or any other component whose properties are XML data. This is because XML stores all data—including numbers, dates, and Boolean values—as strings. If you want to use the actual data instead of its string representation, you use an encoder.
- You need a formatter when you want to control how the data is converted to a string, usually for display purposes.
- You need a data type when you want data validation to occur, you want better conversion for certain data types, or both.

For more information on these schema item settings, see "This section discusses advanced topics, such as refinements you make to schema settings and information for developers who need to write server-side code to interact with Flash data applications.Schema item settings" on page 430.

Working with bindings in the Bindings tab

Once you have imported and defined schemas for your data components, as described in "Working with schemas in the Schema tab" on page 395, you can start adding bindings. You use the Bindings tab to add and remove bindings to and from components and their properties. All the bindings for a component appear here.

The following illustration shows the Bindings tab. The top pane lists the properties exposed for binding, represented by their schema location, of the component that's selected on the Stage and contains Add Binding (+) and Remove Binding (-) buttons. The bottom pane shows information about settings for the selected property, such as what it's bound to and in which direction it's bound.

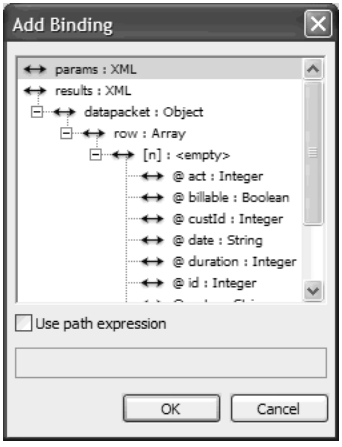


To walk through the steps of creating bindings, see "A simple binding example" on page 394. The following topics describe each step of creating bindings in more detail:

- "Adding a binding" on page 403
- "Configuring bindings" on page 404
- "Defining what to bind to" on page 405
- "Creating an indexed binding" on page 406

Adding a binding

To add a binding, click the Add Binding (+) button on the Bindings tab. The Add Binding dialog box appears.



This dialog box shows all the schema items (properties) for the selected component. You use this dialog box to select which property you want to expose for binding. Component properties appear as root nodes within the schema tree. An arrow icon represents whether a schema item has read/write access, as follows: a right-pointing arrow represents a write-only property, a left-pointing arrow represents a read-only property, and a bidirectional arrow represents a read-write property. (See "Configuring bindings" on page 404.)

To walk through the steps of creating a binding, see "Creating a simple application" on page 390, which creates a simple data application, or "A simple binding example" on page 394, which demonstrates how bindings connect two UI components.

In general, follow these steps to add a binding:

1. Select the component on the Stage for which you want a binding.
2. In the Component inspector, click the Bindings tab.
3. Click the Add Binding button. The Add Binding dialog box opens.
4. Select the property for which you want to add a binding.
5. In the bottom pane of the Bindings tab, click Bound To. The value field becomes editable.
6. Click the magnifying glass icon in the field and select the component path and schema location to bind to. See "Defining what to bind to" on page 405.

7. In the bottom pane of the Bindings tab, click Direction and select the appropriate value from the pop-up menu. See "Configuring bindings" on page 404.
8. Repeat the steps for additional components.

The schema for a component defines which schema items are bindable. However, you might need add a binding for a schema item that is not identified in the data source's schema. You can do this by selecting the Use path expression option. See "Adding bindings using path expressions".

Configuring bindings

When a property is selected on the Bindings tab, you can further define it using the options located in the bottom pane of the Bindings tab. You can specify information such as Direction and Bound To, which you'll commonly need to specify, as well more complex properties such as Formatter and Formatter Options:

Direction Shows a list of directions that can be set for a binding. You need to select a value from the list:

- In: The selected schema item is the destination of a binding. It receives a new value when the other end of the binding changes. On the Schema tab, it is represented by a left arrow.
- Out: The selected schema item is the source of a binding. Whenever its value changes, the value is copied to the other end of the binding. On the Schema tab, out is represented by a right arrow.
- In/Out: New data values are copied when either end of the binding changes value. On the schema tab, in/out is represented by a two-headed arrow.

Bound To Identifies the destination schema item (another component's property) to which this schema item is bound. You need to specify this value. See "Defining what to bind to" on page 405.

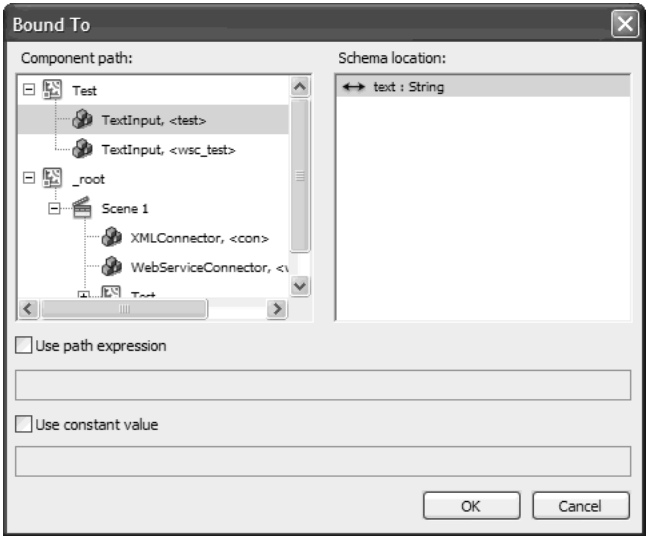
Formatter Shows a list of available formatters that determine how to display this binding. For more information, see "Schema formatters" on page 438.

Formatter Options Shows the Formatting Options dialog box. The settings in this dialog box are used at runtime to control formatting of data assigned from this schema item to the destination schema item that is defined in the Bound To property. These settings override the default formatting settings for the source schema item. See "Schema formatters" on page 438.

Index For If you create a binding for a schema item that is defined as a field of an object contained within an array, you must specify an index for the array. See "Creating an indexed binding" on page 406.

Defining what to bind to

When you expose a component property for binding, you need to define what to bind the property to. The Bound To dialog box appears when you click Bound To in the Binding Attributes pane of the Bindings tab. The Bound To dialog box includes the Component Path pane and the Schema Location pane.



The Component Path pane shows a tree of components that have properties to which you can bind. The tree is based on the current Stage editing environment:

- If the Stage shows the contents of the document root, a single component path tree appears relative to the document root.

NOTE	Component instances are displayed only if they exist in Frame 1 of the edited document root or in Frame 1 of any screen/clip whose instance exists in the edited document root. This pane shows only components, not text fields.

- If the Stage shows the contents of a movie clip being edited from the library, two component path trees appears. The first appears from the root of the symbol being edited, and the second appears from the document root, allowing bindings to instances within the document.

NOTE	Bindings to this second component tree do not appear in the Bound To instances when they are selected. They appear only as bindings from the Bound From component instance.

The Schema Location pane shows the schema tree of the component selected in the Component Path pane. This is the same information that appears in the Schema Tree pane of the Component inspector Schema tab.

You can use a dynamic value or a constant value for the Bound To property.

To use a dynamic value for the Bound To property:

1. Select a component in the Component Path pane.
2. Do one of the following actions to select a schema item for the data:
   - Select a schema item using the Schema tree located within the Schema Location pane.
   - Select Use Path Expression, select a component property from the schema tree, and enter a path expression. For more information, see "Adding bindings using path expressions" on page 444.

To use a constant value for the Bound To property:

- Select Use Constant Value, and enter a constant value, such as `3`, a string, or `true`. You can use any value that is valid for the schema item. When you use a constant value, the selected component path, schema location, and path expression are ignored. You can bind to a constant value only when the Direction attribute for the binding is set to In.

Creating an indexed binding

In the example application created in "Creating a simple application" on page 390, the data grid displays the dinner menu. The description of each food item, however, is too long to fit in the data grid. Ideally, the user could click an item in the data grid and read the full description of a food item, perhaps in a text box below the data grid. To accomplish this, you would create an indexed binding to the data array.

This section shows you how to create an indexed binding to connect a field in your data source with the selected index of another component. The most common use for an indexed binding is to the `selectedIndex` property of a UI element. When you create a binding to the index of an array, a setting for its value is dynamically added to the Schema Attributes pane; you use this setting, the `Index for` field, to specify to what to bind the index.

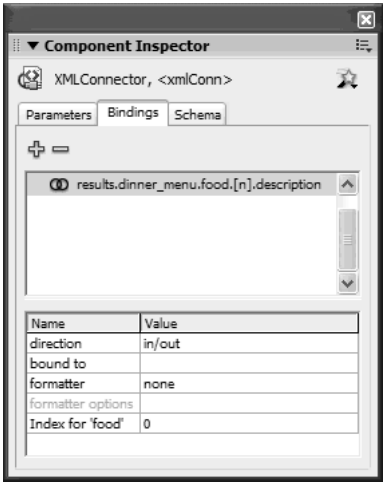
NOTE If a schema item location includes several array references such as `"foo/bar[]/abc[]/def[]"`, three `index for` settings are dynamically added to the Schema Attributes pane—one for each array that needs to be indexed.

In the following example, you add a text box to display the full description of the food item when a user clicks on the item in the data grid.

To create an indexed binding:

1. If you haven't already done so, create the example application shown in "Creating a simple application" on page 390.
2. Drag a TextArea component to the Stage and name it `myTextArea`.
3. Select the `xmlConn` instance, click the Bindings tab, click the + symbol, and select the `description:String` property, which is in the `food` array.

   Notice that on the Bindings tab, the attribute `Index for 'food'` is dynamically added, as shown in the following image; you'll fill in this value in a later step.



4. With the `results:dinner_menu:food.[n].description:String` field selected on the Bindings tab, click Bound To, click the magnifying glass icon, select `myTextArea`, and select the `text:string` property.

   The text area will be populated by the `description` property of the `food` array.

   Next, you define the index value for the `food` array, so that when the user clicks a different item in the data grid, the correct description populates the text box.

5. Click `Index for 'food'`, click the magnifying glass icon, deselect Use Constant Value, select the `menu_dg` DataGrid instance, and select `selectedIndex:Number`. The settings for the indexed binding appear in the Bindings tab, as shown in the following image:

6. Next, set the DataGrid index default value to 0 to make it available for data binding: select the `menu_dg` instance, click the Schema tab, select `selectedIndex:number`, and in the Default Value field in the lower pane, type `0`.
7. Save and test the application. Click Load Data, then click different items in the data grid.

   The text area updates with the detailed description for each food item. Each time the user selects a new item in the data grid, the index of the array is updated to show the data associated with the new item.

> **NOTE** The `index for` property appears only in the Binding attributes pane for a schema item that is the field of an object within an array.

Sometimes you might need to manually define a schema that identifies a schema item as a field of an object contained within an array. In the following example, the id, billable, rate, and duration schema fields are all considered attributes of an object contained within the row array:

```
results : XML
   datapacket : Object
      row : Array
         [n] : object
            @id : Integer
            @billable : Boolean
            @rate : Number
            @duration : Integer
```

If a binding is created for any of these items, an `index for 'row'` property appears in the Binding Attributes pane, so that an index can be specified for the row array. Flash uses the [n] schema field to identify this type of relationship. Therefore, you might need to duplicate this entry if you are manually creating a schema. To do this, you add a new schema field under the `row : Array` node and set Field Name for the schema field to `[n]`. The compiler reads this value and creates an `index for` property if it is used within a binding.

About debugging data binding and web services

Data binding is a series of actions that occur in response to events, such as the following:

- The data of a component property changes.
- A web service call is completed.
- An XML document is fetched.

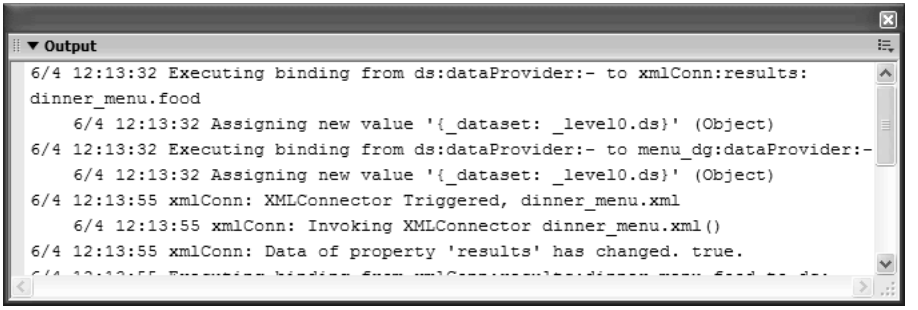
You can create a log of all actions that are performed by data binding or web services. To create the log, create a new Log object by adding the following code to the first frame in your Flash document:

```
_global.__dataLogger=new mx.data.binding.Log();  //to enable trace log
```

To turn the trace log off, use the following code:

```
_global.__dataLogger=null; //disable trace for binding
```

When you run an application that turns the trace on, a detailed log of data binding and web services events and actions appears in the Output window. The following image shows the log for the application created in “Creating a simple application” on page 390, when the code to enable the trace log is added to the first frame of the application:



The following list describes the types of things reported:

- Executing bindings
- Calling web service methods
- Fetching XML documents

- Status and result events from WebService and XML components
- Valid and invalid events from validated data fields
- A variety of errors, invalid settings, and so on

By running your application and then examining the log, you can often find out why things are not working as expected. Sometimes an error is explicitly reported—for example, a missing web service parameter. Sometimes the data is bound to the wrong component or to no component and so on. If you find that there is too much information in the log, clear the Output window by selecting Clear from the context menu, to keep the log as short as possible.

For more information, see "Log class" in the *Components Language Reference*.

Data binding in Flash Player 7 versus Flash Player 6

Bindings between components are executed based on default component events (for example, a binding between the `selectedIndex` of a DataGrid and a DataSet is executed whenever a new record is selected in the DataGrid or DataSet. After the event is generated, the binding is queued to be executed as soon as possible. This action depends on your version of Flash Player. If you publish to Flash Player 7 or higher, the binding happens immediately. If you publish to an earlier version of Flash Player, the binding is queued to the beginning of the next frame.

However, the DataSet component works only in Flash Player 7 or higher. Queuing bindings to the next frame can potentially cause issues with components, such as the DataSet, that provide their own events for accessing data that may become out of sync with data binding. To avoid these issues, Adobe recommends that you publish to Flash Player 7 or higher when using the DataSet component.

Data connectivity

You use the connector components in Flash to connect to your data source. The schema for your data source is mapped to properties of a connector component. A typical application might contain several connector components for retrieving or updating data, or both.

Before you can create data bindings, you must either set up a connector component on the Stage or create the proper mappings in ActionScript using the WebServiceConnector component class. However, it is useful to first understand how data bindings in Flash work; see "Data binding" on page 393.

NOTE	External data refers to any data that is accessible through HTTP.

Flash comes with the following connector components:

- The WebServiceConnector component, which lets you connect to the WSDL URL of a web service.
- The XMLConnector component, which lets you connect to any external data source that returns XML through HTTP (such as JSP, ASP, Servlet, or ColdFusion).

In addition to, or instead of, using these connector components, advanced developers and database administrators can use the WebServices classes to write ActionScript code that accesses remote procedure calls exposed by a server using Simple Object Access Protocol (SOAP). For more information, see "Web service classes" in the *Components Language Reference*.

NOTE	The WebService classes are accessible only through ActionScript code and are common to various Adobe products. The WebServiceConnector component has an API that is unique to Flash and lets you access the component's methods, properties, and events through the visual interface.

To help you consider what kind of connectivity architecture you should implement, see the following Adobe Developer Center articles: "Choosing Between XML, Web Services, and Remoting for Rich Internet Applications" at www.adobe.com/go/learn_fl_ria_dataservices and "Getting a Handle on Web Services" at www.adobe.com/go/learn_fl_webservices.

Connecting to web services with the WebService connector component

The WebServiceConnector component lets you introspect, access, and bind data between a remote web service and your Flash application. A single instance of a WebServiceConnector component can be used to make multiple calls to the same operation. To call more than one operation, use a different instance of a WebServiceConnector component for each operation. For example, you would use one instance to connect to a DataSet component and another instance to connect to a resolver component, as shown in the illustration in the overview at the beginning of this article.

To use the WebServiceConnector component, you need to load the web service's schema into the WebServiceConnector component. A web service's schema is defined by a Web Service Description Language (WSDL) file. The WSDL file, which is accessible through a URL, specifies a list of operations, parameters, and results that are exposed by the web service. Once the schema is loaded, you can proceed to add data bindings.

You can load and view the schema of any web service by entering the URL into the `WSDLURL` parameter of a WebServiceConnector component instance.

The following example demonstrates how to load and view the schema for a web service that provides helpful tips for different products. You add a WebServiceConnector component instance on the Stage, specify the web service to use, and view the web service's schema on the Schema tab of the Component inspector.

NOTE

This example requires an active Internet connection because it uses a public web service. If you use a web service in your application, the web service must be located in the same domain as the SWF file for your application so the application can work in a web browser. For more information, see "About data connectivity and security in Flash Player" on page 417.

1. Drag a WebServiceConnector component to the Stage and name it `tipsWSC`.
2. In the Component inspector, click the Parameters tab, if not already selected.
3. Select the `WSDLURL` parameter, and type the following URL:

   ```
   http://www.flash-mx.com/mm/tips/tips.cfc?WSDL
   ```

   When you specify a web service for a WebServiceConnector component in this way, it is automatically added to the Web Services panel and is available to any application you create.
4. Select Operation, and select the `getTipByProduct` method.

5. Click the Schema tab and view the auto-generated schema for the web service:

The Schema tab displays a schematic representation of the service that you are calling. The parameters and results structure are defined within the schema. The Tips schema states that the service expects one String parameter, `product`, when it is called; this is the write-only input, as indicated by the right-pointing arrow. The service returns a string as the result of the call; this is the read-only output, as indicated by the left-pointing arrow.

Once the web service's schema is brought into the Schema tab, the items identified within the schema can now be bound, using the Bindings tab, to a variety of UI controls to let users input values for the parameters and to get back and display the results of the web service. To see this web service in action, see the Tips sample application at www.adobe.com/go/learn_fl_samples. Download and decompress the Samples zip file and navigate to the DataIntegration\AdobeTips folder to access the sample.

For information on data binding, see "Data binding" on page 393 and "Working with bindings in the Bindings tab" on page 402.

For a common workflow and information on the properties, methods, and events of the WebServiceConnector component, see "WebServiceConnector component" and "Using the WebServiceConnector component" in the *Components Language Reference*.

Using the Web Services panel

You can view a list of web services, refresh web services, and add or remove web services in the Web Services panel (Window > Other Panels > Web Services). When you add a web service to the Web Services panel, the web service is then available to any application you create. When you drag a WebServiceConnector component onto the Stage and specify a value for the WSDLURL parameter, that web service is automatically added to the Web Services panel.

You can use the Web Services panel to refresh all your web services at once by clicking the Refresh Web Services button. If you are not using the Stage but instead are writing ActionScript code for the connectivity layer of your application, you can use the Web Services panel to manage your web services.

The following illustration shows the Web Services panel, to which several web services have been added. A web service is represented by the planet icon, and its operations appear in the tree.

To add, edit the name of, or remove a web service:

1. Click Define Web Services (the planet icon at the top of the panel).
2. To add a service, click Add Web Service, and enter the URL of the web service. Double-click an existing web service to edit its name, or select a service and click Remove to remove it.

   If you want to edit a WebServiceConnector component's schema, you can edit it from the Schema tab of the Component inspector.

NOTE	Access to a web service (as with any external data) is subject to Flash Player security features. For more information, see "About data connectivity and security in Flash Player" on page 417.

Connecting to XML data with the XMLConnector component

The XMLConnector component lets you access any external data source that returns or receives XML through HTTP. A single instance of an XMLConnector component can be used to make multiple calls to the same operation. To call more than one operation, use a different instance of an XMLConnector component for each operation. For example, you would use one instance to connect to a DataSet component and another instance to connect to a resolver component, as shown in the illustration in the overview at the beginning of this article.

To use the XMLConnector component, you load a sample of your XML document's schema into the component. The schema is the structure of the XML document that identifies the data elements in the document to which you can bind.

To load the schema, you import a sample of the XML data to which you're connecting. You can either use an actual sample of real data or, if you know XML scripting, create a sample yourself. You import that sample XML file using the Component inspector.

Be sure that the sample you use contains all the elements you want for data binding and accurately represents the real data. Different XML structures result in different schemas. For example, if your sample contains an array with only one item, Flash won't know that you need an index for that array. The array needs to contain at least two items.

To import a sample schema:

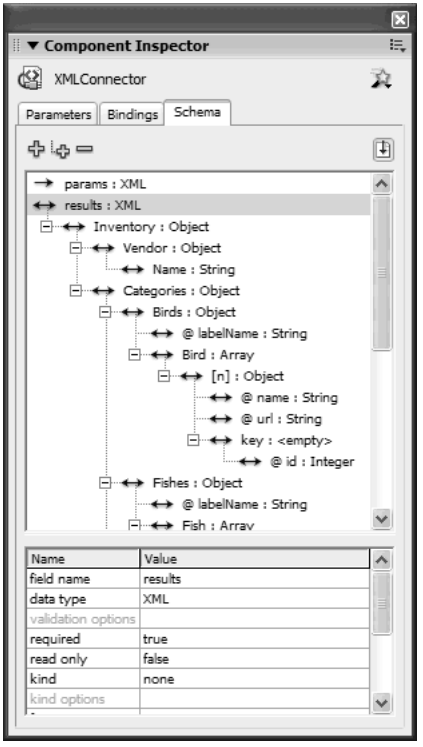
1. Locate the XML file to use as a sample.
2. Drag an XMLConnector component to the Stage.
3. Click the Parameters tab in the Component inspector and for the URL parameter, specify the fully qualified name of the XML data source.
4. Click the Schema tab in the Component inspector and select `params` or `results`, as appropriate. Select `results` if the XML sample represents the schema of the results of a call to the data source.
5. Do one of the following to import the schema:
   - Click the Import Sample Schema button in the upper right corner of the Schema tab.
   - Click the options menu control in the upper right corner of the Component inspector and select Import XML Schema from the menu.

6. In the Open File dialog box, select the file that you want to use as a sample, and click Open.

   The schema appears in the Schema tab. You can now create bindings between elements of your XML document and other component properties within your application.

> **NOTE** Some XML documents may have a structure that Flash cannot represent; for example, elements that contain text and child elements mixed together.

The following illustration shows the schema for a file named Animals.xml:



The schema tab displays a schematic representation of the structure of the XML file. It says that the `results` property of the XMLConnector component is an XML object. The root element of that object is called Inventory, which contains the elements Vendor, Categories, and so on. The Vendor element contains a single element called Name, which is a string. The Categories field contains an element called Birds, which contains the attribute `labelname`. The Birds element also contains an array of objects called Bird. Each of these objects has two attributes: `name` and `url`. It also contains a single element named key, which contains the attribute `id`. The index for the Bird array is represented by the `[n]` field.

The String and Integer fields can be bound to UI components. The Array field Bird can be bound to a DataSet component or to list-based UI components such as List, DataGrid, or ComboBox, which all use the data provider interface. Or, you can directly bind UI components to fields within certain records of the array, as shown in the example application in "Creating an indexed binding" on page 406.

A typical workflow for an application that works with data would include binding an array from the XMLConnector component to the DataSet component's `dataProvider` property. Or, you can directly bind UI components to fields within certain records of the array, as shown in the example application in "Creating an indexed binding" on page 406. In this scenario, the data set could be used to manage the data. The fields within the data set could then be mapped to any of the UI components using data binding.

For more information on the XMLConnector component, including its properties, methods, and events, see "XMLConnector component" in the *Components Language Reference*. For a common workflow using this component, see "Using the XMLConnector component" in the *Components Language Reference*.

You can also read the following tutorials on the Adobe Developer Center: "Bike Trips Sample" at www.adobe.com/go/learn_fl_xmlconnector and "Data Integration Using ASP" at www.adobe.com/go/learn_fl_flashpro_asp.

About data connectivity and security in Flash Player

Many developers are interested in using an industry standard such as SOAP web services as the data-exchange mechanism between their client and server. One reason this approach is gaining favor is the increasing number of popular servers that support exposure of logic using SOAP.

There may be cases where you want the client software to use web services that are published by third parties or hosted on servers that fall outside the Flash Player sandbox. Access to external data through any connector component is subject to the sandbox security model in Flash Player, for all Flash applications that run in a web browser. The sandbox security model restricts a Flash document from accessing data from any domain other than the one in which it originated (this includes public web services). There are a couple of ways to accomplish what you want to do, while still preserving the user security and privacy that the Flash Player sandbox provides:

- Create a policy file that is hosted on the server containing the web service to be used. For more information, see "Server-side policy files for permitting access to data" in *Learning ActionScript 2.0 in Flash* and the security tech note 14213 at www.adobe.com/go/tn_14213.

- Create an intermediary object that resides on the server to act as a bridge between your client and the public services you want to use. This approach offers several advantages:
  - Public web services can be aggregated. With this approach you can provide fail-over safety and load balancing when a request is made for data.
  - You can control the flow of data in your application. If the web service goes away or the URL is down, you can decide how to respond.
  - Data can be optimized. Multiple requests can be cached.
  - You can have custom error handling. You can determine what errors to send back to the client.
  - Data can be manipulated, converted, or combined. You can pull data from several sources and return one data packet with the combined information.

Many of the SOAP-based applications that you build will use private web services hosted on your server. After you determine the best way to implement and expose your own web services, it is easy to make public web services available to your client application. When you are in control of the server, you can offer a complete solution. The server is the ideal place for business logic that can determine the best way to respond to requests for data and the results that should be sent back to the client. This is also the most secure way to build an application. The server can provide additional processing to make sure that users have access only to certain services as well as protect the client from making calls to malicious services that can return bad data.

For more information, see the Adobe Developer Center article "Getting a Handle on Web Services" at www.adobe.com/go/learn_fl_webservices.

Data management

You use the DataSet component for applications that handle managed data. The term *managed data* refers to the ability to perform advanced operations on a local cache of data, including multiple sorts, filters, finds, and offline caching. A managed data solution requires more setup but gives you greater control over your data. In general, you should use a managed data approach for the following scenarios:

- You need to apply multifield sorts, filters, or ranges to your data.
- You are building an application that provides the ability to work offline (changes to the data are cached offline and can be applied at a later time).
- You want to receive changes from the server and apply them to your local cache of data.
- You want to create a custom transfer object implementation to complement a business class on the server.

- You plan to send updates back to an external data source using the built-in features of the DataSet and resolver components (such as automated tracking of changes to your data that can be converted into multiple formats).

For more information, see "Managing data with the DataSet component" on page 419.

If your application displays dynamic read-only data, you can use a simpler approach that does not use the DataSet component. You would instead bind the results of a connector component directly to UI components within your Flash document.

The DataSet component uses functionality in the DataBinding classes. If you intend to work with the DataSet component in ActionScript only, without using the Binding and Schema tabs in the Component inspector to set properties, you'll need to import the DataBinding classes into your FLA file and set schema properties in your code. For more information, see "Making data binding classes available at runtime" in the *Components Language Reference*.

For a tutorial that uses the DataSet component, see the Adobe Developer Center article "Flash Data Integration Using Microsoft Active Server Pages (ASP)" at www.adobe.com/go/learn_fl_flashpro_asp.

The DataSet component works only with Flash Player 7 or later.

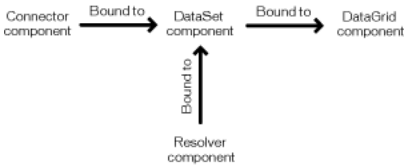
Managing data with the DataSet component

The data structure that is fundamental to data-driven applications is a table with rows and columns, or fields. To expose the fields of the current row in the table, you must define properties of a DataSet component on the Schema tab. (For an example, see the design time example in "Accessing the data" on page 423.)

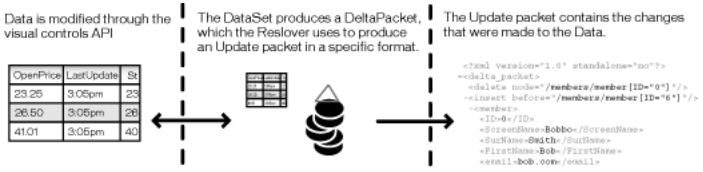
Once you have specified a schema for the DataSet component, you typically create the following bindings to or from a DataSet component:

- Bind the results of a connector component to fields of the DataSet component.
- Bind fields of the DataSet component to properties of UI components within your Flash document.
- Bind the DeltaPacket property of a resolver component to the DeltaPacket property of a DataSet component.

The following diagram illustrates the data binding that typically is needed when you use a DataSet component.



The DataSet component is used to hold and organize your data; you must use data bindings and write ActionScript code to handle updates. Changes that are made to your data through UI components can be tracked and used to generate a DeltaPacket, an object produced by the DataSet component that contains a list of changes made to data at runtime. A resolver component can then manipulate the DeltaPacket into a specific format for use by external data sources. Using the `logChanges()` method of the DataSet component, you can track both changes made to the data and methods called.The following illustration shows the flow of data through a UI component, DataSet and Resolver component, and the DeltaPacket object produced.



For a common workflow and information on how you use the methods, properties, and events of the DataSet component to manage your data, see "Using the DataSet component," "DataSet class," and "DeltaPacket interface" in the *Components Language Reference.*

The DataSet component uses functionality in the DataBinding classes. If you intend to work with the DataSet component in ActionScript only, without using the Binding and Schema tabs in the Component inspector to set properties, you must import the DataBinding classes into your FLA file and set schema properties in your code. For more information, see "Making data binding classes available at runtime" in the *Components Language Reference*.

The DataSet component works only with Flash Player 7 or later.

For more information on working with data in the DataSet component, see the following topics:

- "About loading data into the DataSet component" on page 421
- "Accessing the data" on page 423

About loading data into the DataSet component

To load data into the DataSet component, you edit the schema for the DataSet and create data bindings that can be done either in ActionScript or on the Bindings tab of the Component inspector. You need to edit the schema, in most cases, so that data appears correctly in your application. For information on editing schema, see "Adding a component property to a schema" on page 397 and "Adding a schema field to a schema item" on page 399. You can create bindings for the DataSet component in two ways:

- An array of objects bound to the `DataSet.items` property (see `DataSet.items` in the *Components Language Reference*).
- An object bound to the `DataSet.dataProvider` property. This object should implement the DataProvider interface; see `DataSet.dataProvider` property and "DataProvider API" in the *Components Language Reference.*

The objects can be sophisticated client-side objects that mirror their server-side counterparts, or in their simplest form, a collection of anonymous objects with public properties representing the fields within a record of data.

The DataSet component uses functionality in the DataBinding classes. If you intend to work with the DataSet component in ActionScript only, without using the Binding and Schema tabs in the Component inspector to set properties, you'll need to import the DataBinding classes into your FLA file and set schema properties in your code.

The following examples show different ways you can load objects into the DataSet component, using either ActionScript code or the Component inspector. The examples assume that you have specified a schema for the DataSet component on the Schema tab first; see the design-time example in "Accessing the data" on page 423.

Anonymous objects The following ActionScript example assigns an array of 100 anonymous objects to the `items` property of the `myDataSet` instance of the DataSet component. Each object represents a record of data.

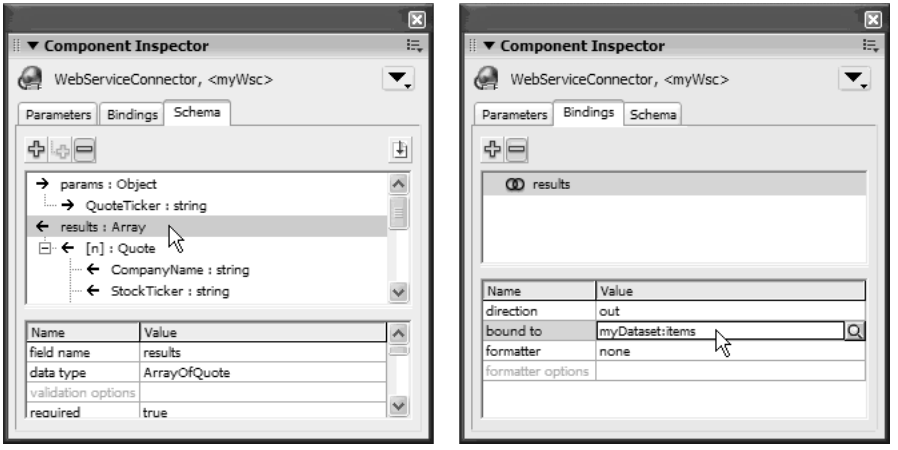
```
function loadData() {
   var recData = new Array();
   for( var i:Number=0; i<100; i++ ) {
      recData[i]= {id:i, name:String("name"+i), price:i*.5};
   }
   myDataSet.items = recData;
}
```

Remoting RecordSet The following ActionScript example assumes that you're using Flash Remoting and that you've made a remoting call that returns a RecordSet. The RecordSet object implements the DataProvider interface. The result is assigned to the `dataProvider` property of the `myDataset` component instance:

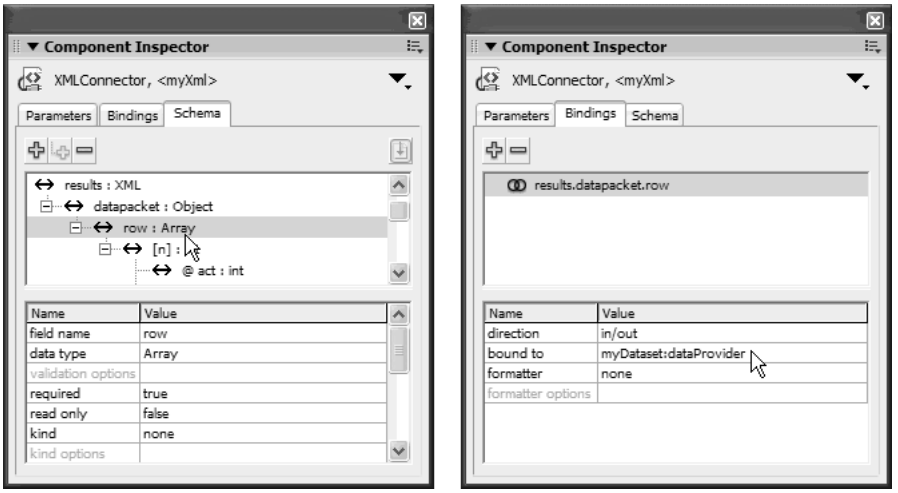
```
function getSQLData_Result(result) {
```

```
  myDataset.dataProvider = result;
}
```

Array of objects returned from a web service The following illustration shows an example of using the Component inspector to bind an array of objects returned from the web service, represented by the `myWsc` instance of the WebServiceConnector component. The illustration on the left shows the schema of the web service. The illustration on the right shows how the `results` array is bound to the `items` property of the `myDataset` component instance.



Array of objects returned from an XMLConnector component The following illustration shows an example of using the Component inspector to bind an array of XML nodes, represented with the XMLConnector component. It assumes that you have imported a schema for an XML file that contains an array of XML nodes. The illustration on the left shows the schema of the XML document, the array of XML nodes represented as an ActionScript array. The illustration on the right shows how the `results.datapacket.row` array is bound to the `dataProvider` property of the `myDataset` instance of the DataSet component.



Accessing the data

After the data is loaded into the DataSet component and the schema for the DataSet component has been defined, the data can be accessed. You can access data at runtime or at design time.

Runtime example. Accessing the data at runtime is simple. Because the data is loaded as objects, data is exposed through properties that can be referenced in code. The DataSet component has a method (`DataSet.first`) that lets you make the first item in the array the currently selected object.

The following code shows an example of accessing data at runtime. It loads an existing DataSet component instance `myDataSet` with customer information and then displays each customer's name in the trace window. Notice that the data types for the customer information—the array of objects—are added so the data displays properly:

```
//Drag DataSet component to Stage and name it myDataSet (easiest way to
  create instance and import necessary libraries)

//Creates recData which contains customer information in an array of objects
var recData = [{id:0, firstName:"Frank", lastName:"Jones", age:27,
  usCitizen:true},
             {id:1, firstName:"Susan", lastName:"Meth", age:55,
  usCitizen:true},
             {id:2, firstName:"Pablo", lastName:"Picasso", age:108,
  usCitizen:false}];

//Assigns recData to the items property of the "myDataSet" DataSet component
  instance
myDataSet.items = recData;

//Adds schema types for the expected fields
var i:mx.data.types.Str;
var j:mx.data.types.Num;

//Makes the first item the current item
myDataSet.first();

//Traces through the properties
while ( myDataSet.hasNext() ) {
  //access the data through the Dataset properties
  trace(myDataSet.firstName + " " + myDataSet.lastName);
  myDataSet.next();
}
```

Design time example Creating fields for a DataSet component at design time is another way to expose the properties of a data object. After the fields are defined, you visually bind UI controls to the data at design time. You can set additional properties (schema item settings) at design time for a DataSet field to affect the way data is encoded, formatted, and validated at runtime. For more information, see "This section discusses advanced topics, such as refinements you make to schema settings and information for developers who need to write server-side code to interact with Flash data applications.Schema item settings" on page 430.

To set up binding to this data at design time, you create persistent fields for the DataSet component that represent the properties of the object. The following procedure shows an example of how you would access the same customer information data at design time. You bind the `recData` array of objects to the items property of the DataSet component in ActionScript, as in the runtime example. Then, you bind `DataGrid.dataProvider` into `myDataSet.items` using the Component inspector.

To access data at design time:

1. Drag a DataSet component onto the Stage. Name it **myDataSet**.
2. Select a layer in the Timeline, and press F9 to open the Actions panel. Type the following code:

```
var recData = [{id:0, firstName:"Frank", lastName:"Jones", age:27,
  usCitizen:true},
            {id:1, firstName:"Susan", lastName:"Meth", age:55,
  usCitizen:true},
            {id:2, firstName:"Pablo", lastName:"Picasso", age:108,
  usCitizen:false}];
myDataSet.items = recData;
```

3. With the DataSet component selected, click the Schema tab of the Component inspector, and click the Add a Component Property (+) button.
4. Set the value for Field Name to **firstName** and leave the Data Type as String.
5. Create three more component properties for the other name/value pairs in the code: field name = lastName, data type = String; field name = usCitizen, data type = Boolean; and field name = age, data type = Integer.
6. Drag a DataGrid component onto the Stage, and name it **myGrid**.
7. Select the DataGrid component, and click the Bindings tab of the Component inspector.
8. Click the Add Binding (+) button to add a new binding. Select `dataProvider:Array`.
9. Click Bound To, select the DataSet component, and select its `dataProvider:Array` property.
10. Click Direction and select In.
11. Save and test the application.

    The data contained within the data set appears in the data grid.

The ability to make use of dynamic component properties that are added to the Schema tab at design time is a special feature of the DataSet component. The DataSet component uses the field name of these properties to map them to the properties of the object or array of objects. The settings that are applied to these properties at design time are then used by the data set at runtime.

If you do not create persistent fields for the DataSet component and you bind it to a WebServiceConnector component or an XMLConnector component that defines a schema, the DataSet component tries to create the correct fields based on the connector component's schema, which might not work. For more information, see "Managing data with the DataSet component" on page 419.

NOTE | Persistent fields that are defined for a DataSet component take precedence over the schema for a connector component.

Data resolution

The resolver components let you convert changes made to the data within your application into a format that is appropriate for the external data source that you are updating. The resolver components can also receive updates from an external data source and convert them into a format that is appropriate for the DataSet component to receive them.

Flash includes the following resolver components:

- XUpdateResolver component for XML data sources
- RDBMSResolver component for relational databases

Typically, you use the resolver components with the DataSet component. When a user edits data in your application, the data is captured in the DataSet component. The DataSet component generates a DeltaPacket, an object that contains a list of changes made to the data at runtime. The resolver component then converts the DeltaPacket to the appropriate format (update packet). When an update is sent to the server, the server should respond with a results packet containing errors or updated field values from the operations that were performed. The resolver components can convert this information back into a DeltaPacket that can then be applied to the data set to keep it synchronized with the external data source.

TIP | The RDBMSResolver component provides limited synching ability at this time.

Resolver components do not send any data from a SWF file to server-side scripts or external data sources. You need to set up this kind of data transfer. Here are the most common ways to send data outside a SWF file:

- Bind the resolver's processed data to a connector component, such as the XMLConnector or WebServiceConnector components. This connector component instance is in addition to the instance that connects your data source to a DataSet or to UI components; see the diagram at the beginning of this article.
- Write ActionScript code using the LoadVars class (see LoadVars in the *ActionScript 2.0 Language Reference)*.

- Write ActionScript code using the XML class (see XML in the *ActionScript 2.0 Language Reference*).

For more information, see "Working with External Data" in *Learning ActionScript 2.0 in Flash*.

NOTE	External data refers to any data that is accessible through HTTP.

Resolving XML data with the XUpdateResolver component

The XUpdateResolver component converts changes made to the data in your application into XUpdate statements that can be processed by an external data source. XUpdate is a standard for describing changes that are made to an XML document and is supported by a variety of XML databases, such as Xindice and XHive. You can write your own server code to handle updates, for example, in your own ASP page, Java servlet, or ColdFusion component. For more information, see the XUpdate specification at http://xmldb-org.sourceforge.net/.

The XUpdateResolver component works only in applications published for Flash Player 7 or later.

For a common workflow and information about the methods, events, and properties of the XUpdateResolver component, see "XUpdateResolver component" in the *Components Language Reference*.

You need to set the correct encoder when you use the XUpdateResolver component; for more information, see the discussion of the DatasetDeltaToXUpdateDelta encoder in "Schema encoders" on page 436.

For a tutorial that uses this component, see the XUpdate tutorial, "XML Tutorial: Timesheet," at www.adobe.com/go/learn_fl_tutorials.

Updates sent to an external data source

When a user edits data in your Flash application, the data is captured in the DataSet component. The DataSet component produces a DeltaPacket, which the resolver component uses to create an update packet. The update packet consists of XUpdate statements, which are communicated to an external data source through a connector component. These statements describe the inserts, edits, and deletes performed on the DataSet component. You can view or bind the contents of the update packet using the `xupdatePacket` property of the XUpdateResolver component.

NOTE	The information contained within the XML update packet is affected in part by the component parameter values that are assigned by the developer. For information on the XUpdateResolver component parameters, see "Using the XUpdateResolver component" in the *Components Language Reference*.

The following XML code is an example of an update packet created by an XUpdateResolver component:

```
<?xml version="1.0"?>
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/
  xupdate">
   <xupdate:insert-after select="/addresses/address[1]" >
      <xupdate:element name="address">
         <xupdate:attribute name="id">2</xupdate:attribute>
         <fullname>Lars Martin</fullname>
         <born day='2' month='12' year='1974'/>
         <town>Leizig</town>
         <country>Germany</country>
      </xupdate:element>
   </xupdate:insert-after>
</xupdate:modifications>
```

When you use the XUpdateResolver component with a DataSet, you must set the correct encoder on the Schema tab: the DataSetDeltaToXUpdateDelta encoder. This encoder is responsible for creating XPath statements that uniquely identify nodes within an XML file based on the information contained within the DataSet component's DeltaPacket. This information is used by the XUpdateResolver component to generate XUpdate statements. For more information about the DataSetDeltaToXUpdateDelta encoder, see "Schema encoders" on page 436.

In addition to client-side code and configuration, you or your server administrator also need to write server code to handle the interaction with your Flash application. For more information, see "Server-side requirements for resolving XML data" on page 446.

Resolving data to a relational database

The RDBMSResolver component creates an XML packet that can be sent to an external data source (such as ASP/JSP page, servlet, and so on). The XML packet can easily be translated into SQL statements that can be used to update any standard SQL relational database. Your development team must write the server code to parse the XML and generate SQL statements.

You can use the RDBMSResolver component to send data updates to any external data source that can parse XML and generate SQL statements against a database—for example, an ASP page, a Java servlet, or a ColdFusion component.

When an RDBMSResolver component receives a delta packet from a DataSet component, it converts it into an XML update packet, which can be communicated to an external data source through a connector component. The converted output is referred to as an update packet and consists of an optimized set of instructions that describe the inserts, edits, and deletes performed on the DataSet component. You can view or bind the contents of the update packet using the `updatePacket` property of the RDBMSResolver component.

The RDBMSResolver component works only with Flash Player 7 or later.

For a typical workflow and information on the methods, properties, and events of the RDBMSResolver class, see "Using the RDBMSResolver component" and "RDBMSResolver component" in the *Components Language Reference*.

In addition to requirements for your Flash application to resolve data, there are requirements for your server code to fulfill. For more information, see "Server-side requirements for resolving data for RDBMS" on page 447.

For a tutorial that uses the RDBMSResolver component, see the Adobe Developer Center article "Using the RDBMSResolver to Update a Database" at www.adobe.com/go/learn_fl_delta_packet.

Formatting your results

By default, the resolver components use the schema specified on the connector components to format values sent to the server. This method ensures that a date value sent from an external data source using the format "MM/DD/YYYY" is sent back to the external data source using the same format.

However, in some cases, you might find that the values you're sending to your external data source are not formatted correctly. This can occur when you don't use a connector to retrieve your data or you want to change the format of the data to be sent to an external data source. In this case, you can control the formatting by adding properties to the resolver component's schema. For instance, if you have a Boolean field called Billable in your DataSet component, its value can be formatted in an update packet as `true` or `false`. If you want it formatted as `yes` or `no`, you can create a new component property called `Billable` within the Schema tab for your resolver. Using the schema settings, you can set the data type as Boolean, the encoder as Boolean, and the encoder options as `yes` or `no`. The encoder is applied when the resolver creates the update packet, and the value for the billable field is represented as `yes` or `no`.

For more information, see "Adding a component property to a schema" on page 397.

Advanced topics in data integration

This section discusses advanced topics, such as refinements you make to schema settings and information for developers who need to write server-side code to interact with Flash data applications.

Schema item settings

This section contains details about schema item settings and how you edit them. To help you determine whether or not you need to look at schema item settings, see "When to edit schema item settings" on page 442.

The schema of a component shows what properties and fields are available for data binding. For each property or field, there are settings that control validation, formatting, type conversion, and other features that affect how data binding and the data management components handle the data of a field. The Schema Attributes pane, the bottom pane of the Schema tab, presents these settings, which you can view and edit. The following list describes the five categories of settings, according to the features they control:

Basic settings Every field or property has these basic schema settings. In many cases, these are the only settings you need to bind to a field:

- Name: Every field needs a name.
- Data type: Every field has a data type, which is selected from a list of available data types. The data type of a field affects data binding in two ways: When a new value is assigned to a field through data binding, the data type determines the rules that are used to check the data for validity. When you bind between fields that have different data types, the data binding feature attempts to convert the data appropriately. For more information, see "Schema data types" on page 440.

- Storage type: Every field has a storage type. Typically, it defaults to one of four values based on the data type of a field. The available values for storage types are simple, attribute, array, or complex.

  NOTE

  Developers almost never have to change this setting. However, there are some cases when the storage type for an attribute contained within the schema for an XML file should be set to attribute.

- Path (optional): This property identifies the location of the data for this schema field. For more information, see "Virtual schemas" on page 443 and "Setting the schema path" on page 434.

Validation settings Validation settings are applicable to any field that is the destination of a binding. You usually modify these settings when you want to control the data validation that the user inputs. To do so, you bind from the UI component to a data component, and then select appropriate validation settings for the fields of the data component. One common example is when the user input is bound to the `params` property of a connector component, such as the XMLConnector component or WebServiceConnector component. Another common example is when UI components are bound to data fields of the DataSet component.

This is how validation works: After any binding is executed, the new data is checked according to the validation rules of the destination field's data type. A component event is then generated to signal the results of the checking. If the data is valid, then the valid event is generated; otherwise, an invalid event is generated. Both components involved in the binding emit the event. You can ignore these events. If you want anything to happen as a result of these events (such as giving feedback to the user), you must write some ActionScript code that receives the valid and/or invalid events.

- Validation Options: Validation options are additional settings that affect the validation rules for this field. The settings are presented in the Validation Options dialog box, which appears when you select this item. These settings vary according to data type. For example, the String data type has settings for the minimum and maximum allowed length of the data. The XML data type has a setting to control if white space is ignored when converting from a String to XML.
- Required: This is a Boolean value that determines whether this field is required to have a non-null value. Validation fails if `required=true` but no value has been set.
- Read-Only: This is a Boolean value that determines whether this field can receive new values through data binding. If `readonly=true`, then executing any binding to this field generates the invalid event, and the field changes.

Formatter settings Formatter settings are applied when a field's value needs to be converted to a string. This is often for display purposes, such as when a DataSet field is bound to the `text` property of a Label or TextArea component. Formatter settings on a field are ignored when that field is bound to something whose data type isn't String.

- Formatter: The name of the formatter to use when converting this field to String. This is selected from a list of available formatters.
- Formatter options: these additional settings affect the formatter. The settings are presented in the Formatting Options dialog box, which appears when you select this item. These settings vary according to formatter. For example, the Boolean formatter has settings for the text that represents the `true` and `false` values.

> **NOTE** If you don't specify a formatter, then a default conversion is applied when a field's value is needed as a string.

For a complete list of formatters, see "Schema formatters" on page 438.

Kind and Encoder settings The Kind and Encoder settings are used to activate certain special features:

- Kind: The Kind setting for this field. This is selected from a list of available Kind settings.
- Kind options: Additional settings that affect the Kind setting. The settings are presented in the Kind Options dialog box, which appears when you select this item. These settings vary according to kind.
- Encoder: The Encoder setting for this field, which is selected from a list of available Encoder settings.
- Encoder options: Additional settings that affect the encoder. The settings are presented in the Encoder Options dialog box, which appears when you select this item. These settings vary according to encoder.

For more information, see "Using kinds and encoders" on page 433, "Schema kinds" on page 435, and "Schema encoders" on page 436.

Default settings These settings let you set defaults for various situations. The following list describes the uses for these settings:

- If a field's value is undefined, the default value is used whenever the value of the field is used as the source of a data binding. For example, the data fields of a DataSet component, or the `results` property of a connector component, can have an undefined value.
- When you create a new row of data in a DataSet component, the default value is used as the value of newly created records

Using kinds and encoders

Kinds and encoders are drop-in modules that perform additional special processing of the data of a schema item. They are often used with each other to accomplish common tasks. The following list describes common uses for kinds and encoders:

Calculated DataSet Fields Calculated fields are virtual fields that do not exist in the underlying data tables. Calculated fields provide developers with the ability to create and update dynamic field values at runtime. This is convenient for calculating and displaying values based on calculations or concatenations performed other fields located in a record (for instance, you can create a calculated field that combines the first and last name fields together to display the full name to a user).

To set up a calculated field for the DataSet component:

1. Select the DataSet component, and click the Schema tab in the Component inspector.
2. Click the Add a Component Property (+) button. This step adds a field to the schema.
3. Using the Schema Attributes pane, give the new component property a field name, and set its kind to `calculated`.
4. In ActionScript code, use the `calcFields` event of the DataSet component to assign this field a value at runtime.

> **NOTE** You should assign a value to a calculated field only within the DataSet component's `calcFields` event.

For an ActionScript code example, see "Schema kinds" on page 435.

Setting up schemas for XML documents In an XML document, all data is stored as a string. Sometimes you want the fields of an XML document to be available as data types other than String. The following example shows an application that pulls in data from an XML file:

```
<datapacket>
   <row id="1" billable="ON" rate="50" hours="3" />
   <row id="2" billable="OFF" rate="50" hours="6" />
</datapacket>
```

If you use this XML file to import a schema for the XMLConnector component's `results` property, it generates the following code:

```
results : XML
   datapacket : Object
      row : Array
         [n] : object
            @billable: String
            @hours : Integer
            @id : Integer
            @rate : Integer
```

Suppose you want to treat the row node as a record within a grid, and you want the `@billable` attribute to be treated as a Boolean value and show a `true` or `false` value in the grid instead of `ON` or `OFF`. Getting the data into the grid is simple: You can simply bind the row schema field to the `dataProvider` property of the grid. The following procedure describes how to get the `@billable` attribute to be treated as a Boolean value and display a `true` or `false` value.

To make the @billable attribute display a true or false value:

1. Select the XMLConnector component, click the Schema tab, and select the @billable schema field.
2. In the bottom pane of the Schema tab, set the `data type` property to Boolean.
3. Set the `encoder` property to Boolean.
4. Select Encoder Options and enter **on** for strings that represent `true`, and enter **off** for strings that represent `false`.

   The encoder now takes the XML data in its raw form (String) and converts it into an ActionScript Boolean value. Using the encoder options, it knows how to encode the string values correctly.
5. Click Formatter, and select Boolean. Select Formatter Options. You now have a choice to define how a `true` and `false` value should appear as a string.
6. Enter **True** for strings that mean `true`, and enter **False** for strings that mean `false`.

   The formatter now takes the ActionScript Boolean value and formats it into a String.

Setting the schema path

The `path` property for a schema field is an optional setting that is used in special circumstances when the schema for your component is not appropriate. Using this setting, you can create a virtual schema field (a field that exists in one location but pulls its value from another). The value of this property is a path expression that is entered in one of the following formats:

- For schemas that contain ActionScript data, the path follows the format `field [.field]...`, where `field` is equal to the name of a field (such as `addresslist.street`).
- For schemas that contain XML data, the path follows the format `XPath`, where `XPath` is a standard XPath statement (such as `addressList/street`).

When data binding is performed, Flash checks to see if there is a path expression for a schema field. If so, it uses the path expression to locate the correct value. For more information, see "Virtual schemas" on page 443.

NOTE

The path expression is always performed relative to the parent node of the schema field.

Schema kinds

A kind determines how a schema item for your component should be accessed at runtime. The following kinds come with Flash:

None The default kind. This kind is identical to the `Data` kind.

Data The schema item is a data structure, and the data field is stored within the data structure as specified by the field's schema location. This is the normal case. The data structure can be in either ActionScript or XML form.

Calculated This kind is used with the DataSet component. You can use it to define a calculated field (a virtual field whose value is calculated at runtime, based on the values of other fields). You write an event handler in ActionScript code that is invoked by the `DataSet.calcFields` event when any non-calculated field in a data set's current data record changes. The event handler must set the value of the calculated fields in that record. There is no special processing when getting or setting the value of a calculated field. For example, in the DataSet component you might define three fields, called `price`, `quantity`, and `totalPrice`. You would set the `kind` property for `totalPrice` to `Calculated` so that you can assign it a value at runtime, as shown in the following example:

```
function calculatedFunct(evt) {
      evt.target.totalPrice = (evt.target.price * evt.target.quantity);
    }
ds.addEventListener('calcFields', calculatedFunct);
}
```

See the `DataSet.calcFields` event in the *Components Language Reference*.

AutoTrigger This kind can be applied to any property of any component but is mainly useful for connector component properties. When a new value is assigned to the property through data binding, the trigger method of the component is called. For more information, see `WebServiceConnector.trigger()` and `XMLConnector.trigger()` in the *Components Language Reference*.

You can create custom kinds. The number of kinds allowed is unlimited. Kinds are defined by XML files found in the Configuration/Kinds folder that is installed with Flash. The definition includes the following metadata:

- An ActionScript class that will be instantiated to mediate access to the data
- A Kind Options dialog box

Schema encoders

An encoder determines how a schema item for your component should be encoded/decoded at runtime. Sometimes you might want a component property to have a different data type that what is actually stored inside the component. For example, an XMLConnector component results property is stored as an XML document, which contains only strings. You might want a certain field in the results to appear as a Boolean value instead.

To do this, you set the field's data type to Boolean, which tells the data binding mechanism to expect Boolean values in that field; and you set the field's encoder to Boolean, which performs the translation between the underlying string value and the Boolean value that data binding expects the property to have. See the example in "Using kinds and encoders" on page 433.

The following encoders come with Flash:

None The default encoder. No encoding/decoding occurs.

Boolean Converts data of the String type into the ActionScript Boolean type. You must specify (using the Encoder Options property) one or more strings, separated by commas, that will be interpreted as `true`, and one or more strings that will be interpreted as `false`. The settings are case-sensitive.

Date Converts data of the String type into the ActionScript Date type. You must specify (using the Encoder Options property) a template string, which works as follows:

- The template string should contain 0 or 1 instances of `"YYYY"`, `"MM"`, `"DD"`, `"HH"`, `"NN"`, and/or `"SS"`, mixed with any other combination of characters.
- When converting from date to string, the numeric year, month, date, hour, minutes, and seconds, respectively, are substituted into the template, in place of YYYY, MM, and so on.
- When converting from string to date, the string must *exactly* match the template, with the correct number of digits for each of year, month, day, and so on.

DateToNumber Converts a Date object into its numeric equivalent. The DataSet component uses this encoder for fields that are of the Date type. These values are stored within the DataSet component as numbers so that they can be sorted correctly.

Number Converts data of the String type into the ActionScript Number type. There are no authoring settings for this encoder.

DatasetDeltaToXUpdateDelta You use this encoder extracts information from a DeltaPacket and generates XPath statements that are passed to the XUpdateResolver component to generate XUpdate statements. It gets the information that is needs to generate the XPath statements from two places:

- The `rowNodeKey` property, which you must specify with the Encoder Options property (defined in the third bullet, below).
- Within the schema that was used for the XMLConnector component that originally retrieved the data.

Using this information, the encoder can generate the correct XPath statements needed to identify your data within the XML file.

The encoder options contain one property:

- The `rowNodeKey` property (String type). In order for an XML file to be updated, the file must be structured in such a way that the node that represents a record in your data set can be uniquely identified with an XPath statement. This property combines an XPath statement with a field parameter to uniquely identify the row node within the XML file and the field within the data set that makes it unique.

  In the following example, the row node represents a record within the XML file. The value of the `id` attribute is what makes the row unique.

  ```
  <datapacket>
     <row id="1" date="01/01/2003" rate="50" hours="5" />
     <row id="2" date="02/04/2003" rate="50" hours="8" />
  </datapacket>
  ```

  The XPath to uniquely identify the row node is shown in the following example:

  ```
  datapacket/row[@id='xxx']
  ```

  In this example, xxx represents a value for the `id` attribute. In a typical case, the `id` attribute in the XML file would be bound to the `id` field of the DataSet component. Therefore, the `rowNodeKey` value would be as follows:

  ```
  datapacket/row[@id='?id']
  ```

  The question mark symbol (`?`) identifies that this is a field parameter. The `id` value specifies the name of the field in the data set. At runtime, the XUpdateResolver component substitutes the value from the `id` field of the data set to generate the correct XPath for the specified record.

In the next example, the contacts node with a category attribute of Management represents the record(s) within the XML file, and the `employeeId` subnode contains the value that makes the record unique:

```
<datapacket>
   <company id="5" name="ABC tech">
      <contacts category="Mgmt">
        <contact>
          <empId>555</employeeId>
          <name>Steve Woo</name>
          <email>steve.woo@abctech.com</email>
        </contact>
        <contact>
          <empId>382</employeeId>
          <name>John Phillips</name>
          <email>john.phillips@abctech.com</email>
        </contact>
        ...
        ...
      </contacts>
      <contacts category="Executives">
        ...
        ...
      </contacts>
      ...
      ...
   </company>
</datapacket>
```

The `rowNodeKey` value for this XML file would be as follows:

```
datapacket/company/contacts[@category='Mgmt']/contact[empId='?empId']
```

You can create custom encoders. The number of encoders allowed is unlimited. Encoders are defined by XML files found in the Configuration/Encoders folder that is installed with Flash. The definition includes the following metadata:

- An ActionScript class that will be instantiated to encode/decode the data. This class must be a subclass of mx.databinding.DataAccessor.
- An Encoder Options dialog box

Schema formatters

A formatter is an object that performs bidirectional conversion of data between a raw data type and string data. The object has parameters that are settable during authoring and runtime methods for performing the conversion. The following formatters come with Flash:

None The default formatter. No formatting is performed.

Boolean This formatter formats a Boolean value as a string. You can set up Boolean options for strings that mean `true` (for example, 1, yes) and strings that mean `false` (for example, 0, no).

Compose String This formatter converts a data object to a string. You define the output format using a string template. The template is arbitrary text that can refer to the data fields in one of the following ways:

- `<field-name>`
- `<field-name.field-name>`, using dots to drill down into the data structure
- `<.>`, which represents the entire object. This can be used, for example, when the original object is a string, in which case `<.>` is simply the value of the string.

Here are two examples using the Compose String formatter. A formatter could be applied to a field that is an object with field name, quantity, and price, and the string output could read: "`You ordered <quantity> units of <name> at <$price> each.`" In another example, the formatter could be applied to a field that is a number, and you could define the string output to read: "`You have <.> messages.`"

Custom Formatter This formatter lets you specify a custom formatter by specifying a class name. The formatter ActionScript class should have the following format:

```
class MyFormatter extends mx.data.binding.CustomFormatter {
  // convert a raw value, returns a formatted value
  function format(rawValue){
  }
  // convert a formatted value, returns a raw value
  function unformat(formattedValue){
  }
}
```

Rearrange Fields This formatter creates a new array of objects based on the original array in your binding. It can only be applied to fields that are arrays. You define the fields on the new array by using a string template in the form:

```
fieldname1=definition1;fieldname2=definition2;and so on.
```

The `fieldnameN` are the names of the fields in the new array or records. The `definitionN` is one of the following:

- The name of a field in the original record
- A string, enclosed in single quotation marks ('), that contains a mix of text and tags. A tag is the name of a field in the original array, enclosed in < and >.
- A single dot (.), which represents the entire original record

For example, suppose you want to assign an array to the `DataProvider` property of a List component using data binding. The objects within the array do not have a `label` property (which the list uses if available). You could use this formatter to create a new array through data binding that replicates the objects within your original array and adds a `label` property to each object using the values you define. The following template would achieve this (this would be on a binding between your array and the List component's `DataProvider` property):

```
label='My name is <firstName> <lastName>;'
firstName=firstName;
lastName=lastName;
```

This syntax assumes that the object has two properties, called `firstName` and `lastName`. The `label` property will be added to each object within the new array.

NOTE

This formatter can be used on any binding from a component property that is of the Array type to another component property of the Array type. Also note that the Rearrange Fields formatter doesn't work if you access if in the Schema panel, but does work if you access it in the Bindings panel.

Number Formatter This formatter allows you to specify the number of fractional digits that appears when a number is converted to text.

You can create custom formatters. The number of formatters allowed is unlimited. Formatters are defined by XML files found in the Configuration/Formatters folder that is installed with Flash. The definition includes the following metadata:

- The ActionScript class that will be instantiated to perform the formatting
- A Formatter Options dialog box

Schema data types

A data type is an object that represents all the runtime logic needed to support a particular data type. A data type can be a scalar type, such as integer, string, date, currency amount, or ZIP code. It can also be a complex type, with subfields and so on. A data type can test a data value to determine if it is valid for that data type. The following data types come with Flash:

Array No validation options.

Attribute XML attribute. No validation options.

Boolean No validation options.

Custom Lets you add a custom class to check for this special kind of validation. Your code should call the validate function when the field is assigned a new value, inspect the value, and determine whether it's valid. If it is, the function should simply return. If not, the function should call `this.ValidationError("some informative message");`. The custom class must be in the classpath and formatted as shown in the following example:

```
class myCustomType extends mx.databinding.CustomValidator {
  function validate(value) {
      ... some code here
   }
}
```

DataProvider No validation options.

Date No validation options.

DeltaPacket No validation options.

Integer A validation option can be set up to define the minimum and maximum values.

Number A validation option can be set up to define the minimum and maximum values.

Object No validation options.

PhoneNumber No validation options.

SocialSecurity No validation options.

String A validation option can be set up to define the minimum and maximum number of characters.

XML Lets you specify if white space should be ignored when a string is converted into XML.

ZipCode No validation options.

NOTE	The following data types can perform validation: Custom, Integer, Number, PhoneNumber, SocialSecurity, String, ZipCode. The following data types can convert from various other data types when you assign to them: Boolean, DataProvider, Integer, Number, String, XML.

You can create custom data types. The number of data types allowed is unlimited. Data types are defined by XML files found in the Configuration/DataTypes folder that is installed with Flash. The definition includes the following metadata:

- An ActionScript class that will be instantiated for validation and type conversion
- A Validation Options dialog box
- The name of the standard formatter, which you can override using the `formatter` property
- Initial values for required, read-only, and default values

When to edit schema item settings

You can edit anything within the Schema Attributes pane, even schemas that come from an external source, such a web service WSDL file. You can always change any value for any field of any schema, with the following restrictions:

- If you change the type, all the other schema item attributes are reset to the default values for the new data type.
- If you select to completely reload the schema for a component property, you will lose all the edits that had previously been made within the Schema Attributes pane.

> **NOTE** There are several ways to reload the schema for a component property, including entering a new WSDL URL, selecting a different operation for a web service, or importing a new XML schema from a sample XML file.

When you build an application using data components and/or data binding, you need to apply schema item settings to some, but not necessarily all, fields of the components in your application. The following table summarizes the most common uses of schema item settings and can help you determine when these settings need to be edited:

Component	Property/field	Settings	When to use
Any connector	`params` (and its sub-fields)	Validation Options, Read-Only, Required	If validation is desired.
	`results` (and its sub-fields)	Formatter, Formatter Options	For fields that need formatting for display as text.
		Default value	For fields whose value is sometimes undefined.
DataSet	Any data field	Name, Data Type	You must set these for every data set field that you define.
		Validation Options, Read-Only, Required	If validation is desired.
		Formatter, Formatter Options	For fields that need formatting for display as text.
		Default Value	For fields whose value is sometimes undefined, or to specify the initial value for newly created data set records.

Component	Property/field	Settings	When to use
UI components	UI components typically don't need any changes to their schema settings.		
Any component	Any property or field	Kind, Kind Options, Encoding, Encoding Options	Various purposes, as described in "Using kinds and encoders" on page 433.
Any connector	`results` (and its subfields)	Path	To identify the location of the data for a virtual schema field.

Virtual schemas

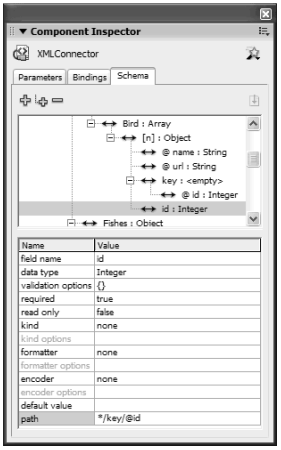
When you bind an array of data to a DataSet's `items` or `dataprovider` property, the data set only recognizes fields that are top-level items within each row of the array. It does not recognize items nested within other objects. A virtual schema lets you change how the underlying data structure is interpreted when bindings are executed. The new structure is derived using XPath statements. For more information, see "Adding bindings using path expressions" on page 444.

For example, the schema for Animals.xml file described in "Connecting to XML data with the XMLConnector component" on page 415 defines an array of objects called Bird. Each object contains two fields (`name` and `url`). They also contain a sub-element with one field called `id`. If you bind the Bird array to a DataSet component (using the `dataProvider` property) with three fields—`name`, `url`, and `id`—each item that is returned from the array is constructed in the following way, for each item in the XML file:

- Create an empty item.
- Loop through the defined schema properties, extracting the values for each property from the XML data, and assign these values to the created item. The Name and URL fields will have values.
- Provide this item to the DataSet component.

  The ID field does not exist on the item, and the DataSet component has a blank entry for each item assigned.

The solution is to create a new schema field under the object within the Bird array. The new schema field is named `id`. Every schema field has a property called `path` that accepts an XPath statement that points to the data in your XML file. In this case the XPath expression would be `key/@id`. When you get to the second bullet in the above process, data binding finds an `id` field for the object. It looks at the `path` property and uses the XPath statement to get the correct data from the XML file. The correct data is then passed to the DataSet component.



Adding bindings using path expressions

You can use path expressions for data binding in two areas:

- In the Add Binding dialog, to identify the field you are binding to
- In the Bound To dialog box, to identify the field you're binding from.

The following XPath expressions are supported:

- Absolute paths:

  `/A/B/C`
- Relative paths:

  `A/B/C`
- Node selection using node name or wildcard:

  `/A/B/C` (node selection by name)

  `/A/B/*` (node selection of all child nodes of /A/B by wildcard)

  `/*/*/C` (node selection of all C nodes that have exactly two ancestors)

- Predicate syntax to further specify nodes to be selected:

  `/B[C]` (child node syntax; selects all B nodes that have a C node as a child)

  `/B[@id]` (attribute existence syntax; selects all B nodes that have an attribute named `id`)

  `/B[@id="A1"]` (attribute value syntax; selects all B nodes that have an `id` attribute whose value is A1)
- Support for predicate comparison operators:

  =
- Support for Boolean `and` and `or` values in predicates:

  `/B[@id=1 and @customer="adobe"]`

NOTE	The following operators are not supported: "<", ">", "//".

To add a binding using path expressions:

1. In either the Add Binding dialog box or the Bound To dialog box, select Use path expression.
2. Enter a path expression to identify the schema item to which you want to bind. Path expressions are entered in the following formats:
   - For properties that contain ActionScript data, the path follows this format:

     `field [.field]...`

     In this format, `field` is equal to the name of a field (such as `addresslist.street`).
   - For properties that contain XML data, the path follows this format:

     `XPath`

     In this format, `XPath` is a standard XPath statement (such as `addressList/street`).
3. Click OK to return to the Bindings tab.

Default data binding events

When you use the Bindings tab to create a binding between two components, the binding is triggered by the default component event. If you want a binding to execute independently of the default component event (which is predetermined by Flash), you must manually refresh the binding with ActionScript code. For more information, see "ComponentMixins class" in the *Components Language Reference* (in particular, see the `ComponentMixins.refreshDestinations()` and `ComponentMixins.refreshFromSources()` methods).

In general, for UI components, the `change` or `click` events are the default events used to trigger data bindings, such as `TextInput.change`, `Button.click`, `RadioButton.click`. For connector components, the `result` event triggers the binding, such as `XMLConnector.result`.

Server-side requirements for resolving XML data

This section describes requirements that your server code must fulfill when receiving results from an XUpdateResolver component. It contains information relevant for the server administrator who is handling server-side functions for your Flash application.

After the server finishes with the update packet, either successfully or unsuccessfully, it should send back to your Flash application a results packet containing errors or additional XML updates resulting from the update operation. If there are no messages, the results packet should still be sent, but it will have no operation result nodes.

The following example shows a sample results packet for an update packet that has no errors and contains no XML updates:

```
<results_packet nullValue="{_NULL_}" transID="46386292065:Wed Jun 25
  15:52:34 GMT-0700 2003"/>
```

A sample results packet (with XML updates) follows:

```
<results_packet nullValue="{_NULL_}" transID="46386292065:Wed Jun 25
  15:52:34 GMT-0700 2003">
    <operation op="remove" id="11295627479" msg="The record could not be
  found"/>
    <operation op="update" id="02938027477">
       <attribute name="id" curValue="105" msg="Invalid field value" />
    </operation>
 </results_packet>
```

The results packet can contain an unlimited number of operation nodes. Operation nodes contain the results of operations from the update packet. Each operation node should have the following attributes/child nodes:

- `op`: An attribute describing the type of operation that was attempted. Must be insert, delete, or update.
- `id`: An attribute that holds the ID from the operation node that was sent out
- msg (optional): An attribute containing a message string that describes the problem that occurred when attempting the operation
- `field`: 0, 1, or more child nodes that give field-level specific information. Each field node, at a minimum, should have a `name` attribute, which contains the field name, and a `msg` attribute, which gives the field-level message. It can also optionally contain a `curValue` attribute, which holds the most up-to-date value for that field in that row on the server.

Server-side requirements for resolving data for RDBMS

This section describes requirements that your server code must fulfill. It contains information relevant for the server administrator who is handling server-side functions for your Flash application. It contains the following topics:

- Example of an RDMBSResolver component XML update packet
- About receiving results from an external data source

In addition to the information in this section, see the Adobe Developer Center article "Using the RDBMSResolver to Update a Database" at www.adobe.com/go/learn_fl_delta_packet.

Example of an RDMBSResolver component XML update packet

To handle server-side code, you'll need to understand the XML update packet generated by the resolver component. The information contained within the XML update packet is affected in part by the component parameter values that are assigned by the developer. For information on the RDBMSResolver component parameters, see "Using the RDBMSResolver component" in the *Components Language Reference*.

The following example shows an RDBMSResolver component's XML update packet generated with `updateMode` parameter set to `umUsingKey`:

```
<update_packet tableName="customers" nullValue="{_NULL_}"
  transID="46386292065:Wed Jun 25 15:52:34 GMT-0700 2003">
      <delete id="11295627477">
         <field name="id" type="numeric" oldValue="10" key="true"/>
      </delete>
      <insert id="12345678901">
         <field name="id" type="numeric" newValue="20" key="true"/>
        <field name="firstName" type="string" newValue="Davey" key="false"/
  >
       <field name="lastName" type="string" newValue="Jones" key="false"/>
      </insert>
      <update id="98765432101"> <field name="id" type="numeric"
  oldValue="30" key="true"/>
         <field name="firstName" type="string" oldValue="Peter"
  newValue="Mickey" key="false"/>
         <field name="lastName" type="string" oldValue="Tork"
  newValue="Dolenz" key="false"/>
      </update>
   </update_packet>
```

Elements in the XML update packet include the following:

- `transID`: An ID generated by the DeltaPacket that uniquely identifies this transaction. This information should accompany the results packet returned to this component.
- `delete`: This type of node contains information about a row that was deleted.
- `insert`: This type of node contains information about a row that was added.
- `update`: This type of node contains information about a row that was modified.
- `id`: A number that uniquely identifies the operation within the transaction. This information should accompany the results packet returned to this component.
- `newValue`: This attribute contains the new value for a field that was modified. It appears only when the field value has changed.
- `key`: This attribute is `true` if the field should be used to locate the row to update. This value is determined by the combination of the RDBMSResolver component's `updateMode` parameter, the `fieldInfo.isKey` setting, and the type of operation (insert, delete, update).

The following table describes how the key attributes value is determined. If a field is defined as a key field, using the RDBMSResolver component's `fieldInfo` parameter, it will always appear in the update packet with `key="true"`. Otherwise, the field's key attribute in the update packet will be set according to the following table:

Node type	umUsingKey	umUsingModified	umUsingAll
delete	`false`	`true`	`true`
insert	`false`	`true`	`false`
update	`false`	`true` if the field was modified; `false` otherwise	`true`

About receiving results from an external data source

This section describes requirements that your server code must fulfill. After the server finishes with the update packet, either successfully or unsuccessfully, it should send back a result packet containing errors or additional updates resulting from the update operation. If there are no messages, the results packet should still be sent, but it will have no operation result nodes.

The following example shows a sample RDBMSResolver component results packet (with both update results and change information nodes):

```
<results_packet nullValue="{_NULL_}" transID="46386292065:Wed Jun 25
  15:52:34 GMT-0700 2003">
      <operation op="delete" id="11295627479" msg="The record could not be
  found"/>
      <delete>
         <field name="id" oldValue="1000" key="true" />
      </delete>
      <insert>
         <field name="id" newValue="20"/>
         <field name="firstName" newValue="Davey"/>
         <field name="lastName" newValue="Jones"/>
      </insert>
      <operation op="update" id="02938027477" msg="Couldn't update
  employee.">
         <field name="id" curValue="105" msg="Invalid field value" />
      </operation>
      <update>
         <field name="id" oldValue="30" newValue="30" key="true" />
         <field name="firstName" oldValue="Peter" newValue="Mickey"/>
         <field name="lastName" oldValue="Tork" newValue="Dolenz"/>
      </update>
   </results_packet>
```

The results packet contains four types of nodes:

Operation nodes contain the result of operations from the update packet. Each operation node should have the following attributes/child nodes:

- The `op` attribute describes the type of operation that was attempted. Must be insert, delete, or update.
- The `id` attribute holds the ID from the operation node that was sent out
- The optional `msg` attribute contains a message string that describes the problem that occurred when attempting the operation
- Zero, one, or more `field` child nodes give field-level specific information. Each `field` node, at a minimum, should have a `name` attribute that contains the field name, and a `msg` attribute that gives the field-level message. It can also optionally contain a `curValue` attribute that holds the most current value for that field in that row on the server.

Update nodes contain information about records that have been modified since the client was last updated. Update nodes should have `field` child nodes that list the fields that are necessary to uniquely identify the record that was deleted and that describe fields that were modified. Each `field` node should have the following attributes:

- The `name` attribute holds the name of the field
- The `oldValue` attribute holds the old value of the field before it was modified. This attribute is required only when the key attribute is included and set to `true`.
- The `newValue` attribute holds the new value that the field should be given. This attribute should not be included if the field was not modified (that is, the field has been included in the list only because it is a key field).
- The `key` attribute holds a Boolean `true` or `false` value that determines whether this field can be used as a key to locate the corresponding record on the client. This attribute should be included and set to `true` for all key fields. It is optional for all others.

Delete nodes contain information about records that have been deleted since the client was updated. Delete nodes should have `field` child nodes that list the fields that are necessary to uniquely identify the record that was deleted. Each `field` node must have a `name` attribute, an `oldValue` attribute, and a `key` attribute whose value is set to `true`.

Insert nodes contain information about records that have been added since the client was updated. Insert nodes should have `field` child nodes that describe the field values that were set when the record was added. Each `field` node must have a `name` attribute and a `newValue` attribute.

Lazy decoding in the WebServiceConnector component

When the WebServiceConnector component receives multiple records of data from a web service, it translates them into an ActionScript array so they are accessible within your application. Translating multiple records of data from XML/SOAP into ActionScript native data can be a time-consuming process; large data sets become large arrays, and can take seconds or tens of seconds.

To improve performance, the WebServiceConnector component supports a feature called lazy decoding, which defers this translation. With lazy decoding, result values that are arrays are not immediately translated from XML to ActionScript. Instead, the result value passed to the user is a special object that acts similarly to an array and translates the XML data only when it is requested. The effect of this feature is to improve the perceived performance of web services by spreading the workload over a longer period of time.

To request the data, use the `myArray[myIndex]` ActionScript expression, as for any array. You must access the array using numeric indices; that is, `myIndex` must be a number. To iterate over the array, use the following statement:

```
for(var i=0; i < myArray.length; i++);
```

The expression `for(var i in myArray)` won't work in this case.

To control lazy decoding, you use ActionScript. For more information, see "SOAPCall.doLazyDecoding" in the *Components Language Reference*.

Transfer objects in the DataSet component

It is important to remember that the DataSet component is a collection of transfer objects. This differs from previous implementations of the component, when it was simply an in-memory cache of data (array of record objects). Transfer objects expose business data from an external data source through public properties or accessor methods. When you load data into the DataSet component, the data is translated into a collection of transfer objects. In the simplest scenario, the DataSet component creates and loads the data into anonymous objects. Each anonymous object implements the TransferObject interface, which is all that is required for the DataSet component to manage the objects. The DataSet component tracks changes made to the data and any method calls that are made on the objects. If methods are called on an anonymous object, nothing happens, because the methods don't exist. However, the DataSet component tracks them in the DeltaPacket, which guarantees that they will be sent to the external data source, where they can be called if appropriate.

In an enterprise solution you could create a client-side ActionScript transfer object that mirrors a server-side transfer object. This client object can implement additional methods for manipulating the data or applying client-side constraints. Developers can use the `itemClassName` parameter of the DataSet component to identify the class name of the client-side transfer object that should be created. In this scenario, the DataSet component generates multiple instances of the specified class and initializes it with the loaded data. When `addItem()` is called on the DataSet component, the `itemClassName` is used to create an empty instance of the client-side transfer object.

If you take the enterprise solution one step further, you could implement a client-side transfer object that uses web services or Flash Remoting. In this scenario, the object makes direct calls on the server in addition to possibly storing the calls in the DeltaPacket.

NOTE

You can create a custom transfer object for use by the DataSet component by creating a class that implements the TransferObject interface. For more information on the TransferObject interface, see "TransferObject interface" in the *Components Language Reference*.