Getting Started with ColdFusion MX

Building a Database Query Application with Server-Side ActionScript

Table of Contents

Introduction

This tutorial is designed to show you how to build a Macromedia Flash application that integrates with server-side functionality built with ColdFusion MX. Before you get intimidated by the prospect of having to learn a new language, you will be pleased to know that ColdFusion MX allows Flash developers to write server-side code in a language they already know: ActionScript.

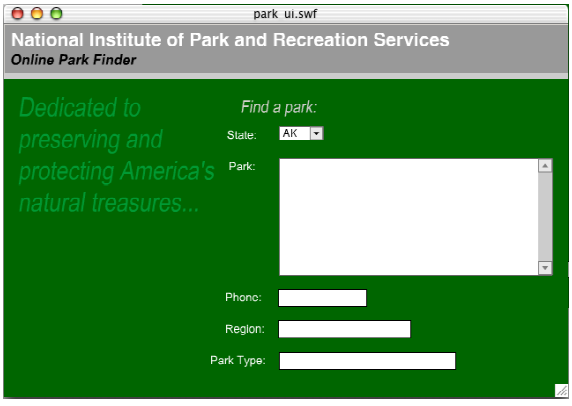
Server-side ActionScript to be precise. Server-side ActionScript uses the same basic syntax that you are used to in Flash, but rather than executing on the client inside of the Flash Player, it executes on the ColdFusion MX server – seamlessly receiving arguments from your Flash client as well as returning data structures, all without you having to write code any more complicated than a simple ActionScript function.

The Scenario

You work for the National Institute of Park and Recreation Services designing, building, and maintaining their extensive, rich, and interactive web presence. Your web site is attracting a great deal of attention, and every day your office receives more and more calls from people trying to find national parks and wildlife preserves near their homes or vacation spots. Any organization with the words "National Institute" in it name has to watch its costs. The most recent concern is the expense of maintaining an 800 number and paying people to monitor it. Being the web guru that you are, your natural instinct is to move the information online.

The plan seems sound enough, and after being promised raises on top of promotions if you can pull it off, you settle down to figure it out. Clearly, you want to maintain the rich and interactive tradition of your web site, so there is no question that the interface will be built in Macromedia Flash. It doesn't take you long to realize that designing the interface is the easy part. Getting at the park data is what has you worried. The idea of hard-coding information about the nearly 400 parks in the company Microsoft Access database - and then having to maintain that list - immediately strikes you as absurd. Fortunately, you remember reading a tutorial on Macromedia's web site that described how easy it is to get data from a database into your Flash movies using a few simple Server-side ActionScript functions.

Your first order of business is to create a fairly simple proof of concept. Thanks to the UI components available in Flash MX, throwing together a working mock-up won't take long at all. You decide that the user will be presented a list of states in a dynamically built ComboBox (by dynamically built, I mean the items in the ComboBox will be built from a list of states in the database). When the user selects a state, the names of all the parks in that state will appear in a ListBox below the ComboBox of states. Then, when a user selects a specific park name from the ListBox, a few details about that park will appear below that. You imagine an unfinished proof of concept will look something like this:



Installation Requirements

To get started, you must have the following Macromedia products installed:

- Macromedia ColdFusion MX Server
- Macromedia Flash MX
- Macromedia Flash Remoting Components
- Macromedia Flash Player 6

Trial versions of each of these products can be downloaded from the Macromedia web site (http://macromedia.com/downloads).

After Macromedia Flash MX is installed, download and install the Macromedia Flash Remoting Components. These are the files you need in order for Flash to be able to connect to your server. Don't worry – Flash won't appear any different next time you open it after installing the Flash Remoting Components. You will just have access to a few additional ActionScript objects and some documentation that wasn't there before.

You can install ColdFusion MX on your local machine or on another machine on your network. Follow the instructions provided to ensure your copy is configured correctly. It can be installed with the built-in web server or it can be configured to run with 3rd party web servers like IIS and Apache. The code in this tutorial assumes that you have configured ColdFusion MX to use the built-in web server. The ColdFusion MX built-in web server listens on port 8500, so requests should look like this:

http://localhost:8500/

If you configure ColdFusion MX to use a third-party web server, it will be configured to listen on a different port. All the code in this tutorial will still work, but where you see the port 8500, substitute the appropriate port number for your server.

What You Learned	The Flash Remoting Components allow the Macromedia Flash Player to connect to ColdFusion MX. After installing them, you will have access to new ActionScript objects that you will use to work with Flash Remoting.

As explained previously, all of the park data in our application will be coming from a database. This tutorial uses "exampleapps," a Microsoft Access database that comes with ColdFusion MX. Once the server is installed, you are ready to start writing code. Even the data source comes pre-configured.

In ColdFusion MX, a data source is a unique name that represents a set of database configuration parameters. Data sources allow you to specify all your database configuration parameters once (using the ColdFusion Administrator). Then, wherever you use the database, you simply call it by its data source name. Depending on the type of database you are configuring, these parameters might include the server the database is running on, the port the database server is listening on, the name of the database or database file, and the username and password required to interact with it. You might want to take a moment to explore the web-based ColdFusion Administrator to get a better understanding of how data sources work.

To explore the ColdFusion MX Administration, open your browser and go to the following URL: http://localhost:8500/CFIDE/administrator/

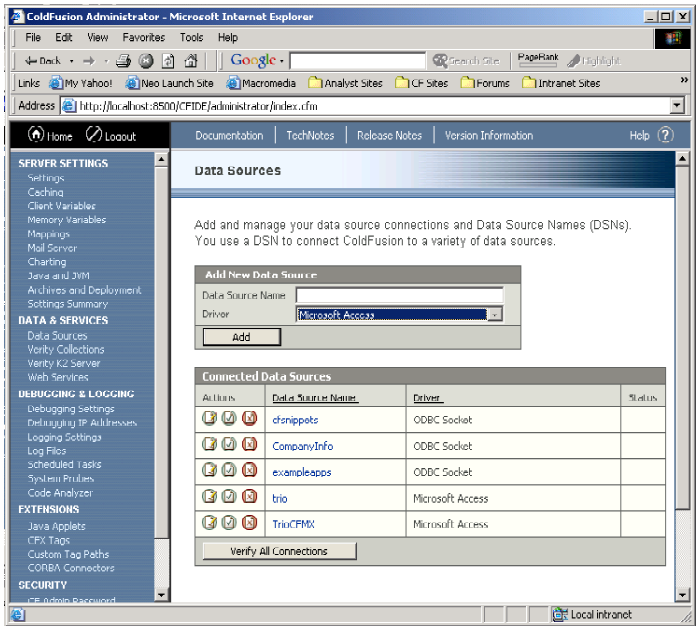
If you are not running ColdFusion on your local machine, substitute "localhost:8500" in the URL above for the server on which ColdFusion MX is installed.

You will be presented with a screen that asks you to provide the administrator password. You were asked to provide this during the installation process. If you did not install and configure the ColdFusion server yourself, ask your system administrator for the password.

Once you log in, you will see the Administrator's main menu. In the left-hand frame, under the "Data & Services" section, click on the "Data Sources" link. You should see the following three data sources already configured:

- cfsnippets
- CompanyInfo
- exampleapps

This screen allows you or your system administrator to view, add, edit, delete, and verify ColdFusion data sources. If you wanted to add your own database as a data source, this is the area where you would do so.



What You Learned	A data source is a unique name that represents a set of database configuration parameters. Using data sources allows you to keep all your database configuration parameters in one central location.

Server-Side ActionScript

As mentioned previously, Server-side ActionScript is very much like the client-side version, except that it executes on the server instead of on the client. The advantage of executing ActionScript on the server is that it gives your ActionScript code access to server-related resources, like databases. In this section, we will write four Server-side ActionScript functions. Three of these retrieve information from the exampleapps database. Once these are in place, we will then write client-side ActionScript that calls these functions to get the data from the server and deliver it to your Flash application.

Setting up Your Server-Side ActionScript Files

Server-side ActionScript resides in files with an .asr extension (ActionScript remote). They can reside in any directory underneath the wwwroot directory of your server. For this tutorial, you can either use the ParkQuery.asr file from the zip file, or you can follow

the tutorial and write your own by retyping the code. The ParkQuery.asr file should be in the following directory:

wwwroot/com/macromedia/tutorial/

This may at first seem like an odd choice for directory names, however it actually makes perfect sense. Just like using your social security number as a tax id or a driver's license number guarantees that the numbers will be unique, using your domain name as part of your directory structure guarantees that you will never have a file overwrite another file by the same name written by another party. Let's say, for instance, that you put all your Server-side ActionScript files in wwwroot/ssas. If you already had a file called ParkQuery.asr, you would run into a naming conflict if you tried to copy my ParkQuery.asr file into the same directory. Using domain names as directory structures will prevent this type of situation by keeping your namespaces safe, not to mention the fact that it will keep your code better organized. This a very common technique derived primarily from common coding practices in Java and Perl.

What You Learned	Keep your .asr files in directories with unique names. If you own a domain name, use it as a way to build a unique directory structure.

Creating Your First Server-Side ActionScript File

Go ahead and create a directory called "com" in the wwwroot directory of your server. Inside of the "com" directory, create a directory called "macromedia", and inside of "macromedia", create a directory called "tutorial". Your directory structure should look like this:

Create a file in the tutorial directory called "ParkQuery.asr" or copy the one from the zip file. Although you can use any text editor to create Server-side ActionScript files, Macromedia Dreamweaver MX includes integrated ActionScript support. You can download a trial version of Dreamweaver MX from Macromedia's web site at http://www.macromedia.com/downloads.

Remember, we are going to be creating four Server-side ActionScript functions. Three will contain database queries we will need in our application (one to build the ComboBox of states, one to retrieve the list of parks in a particular state, and one last query to retrieve a few details about a selected park), and the other will escape strings before they are passed into an SQL statement. We'll cover this last concept in detail when it comes time to write the function.

Before we write any actual functions, type the following at the top of ParkQuery.asr:

```
var queryData = new Object();
queryData.datasource = "exampleapps";
```

The first thing we are doing here is creating a global variable called queryData ("global", means that the variable is available within any function in the file since the variable declaration itself occurs outside of a function) and assigning it a single property called datasource. The datasource property corresponds to the data source already set up in

your ColdFusion MX server. As you will see, we will end up passing this object into another function which will be responsible for running the query and returning a RecordSet object. Let's go ahead and write the getStates function. getStates is responsible for returning a RecordSet of states queried from the tblParks table.

Below your variable declarations, add the following:

```
function getStates() {
    queryData.sql = "SELECT DISTINCT state FROM tblParks WHERE state IS NOT null";
    var states = CF.query(queryData);
    return states;
}
```

The first thing we do inside the getStates function is add another property to queryData called sql. This is the actual query we will run to retrieve the list of states. It is defined inside the function as opposed to outside the function because it will be different for each query. The datasource property is defined globally because it is the same for each query. In fact, as we write the other two functions, you will see how each starts out by re-defining the sql property of the queryData object so that the query will return the data specific to that particular function.

The rest of the code in our getStates function calls one of the built-in server-side ActionScript functions that are part of ColdFusion MX. The CF object is the ColdFusion server, and the query function is the function you use to execute a query against a database. The queryData object tells the server which data source to access and what query to execute. After ColdFusion executes the query, it returns a RecordSet containing the data we requested. Our function then assigns the resulting RecordSet object to the states variable, which is what gets returned by the function. At this point, the function's work is done.

What You Learned	Now you know how to create a Server-side ActionScript file with a function that can actually query data from a database and return it to a Flash movie.

You may be wondering where the RecordSet was returned to. We will get into that in more detail when we discuss the client-side ActionScript piece of this application, but for the overly curious, the RecordSet object is being serialized (represented in a way that can be sent over a network), and sent to Macromedia Flash, which will de-serialize the RecordSet and use it to create a dynamic ComboBox.

The getParks and getParkDetails functions are not much different than getStates; however, there is one thing fundamentally different. As you add the getParks function below getStates, see if you can spot the difference.

Add the following to the ParkQuery.asr file:

```
function getParks(state) {
    queryData.sql = "SELECT parkname FROM tblParks WHERE state='" +
                    state + "' AND parkname IS NOT null";
    var parks = CF.query(queryData);
    return parks;
}
```

If you noticed that that the sql property of the queryData object is defined differently, you are correct. The biggest difference between getParks and getStates is the fact that getParks takes an argument called state. As you can see, Server-side ActionScript

functions can accept arguments just like client-side ActionScript functions. This makes them more dynamic. In this case, we are retrieving a list of parks corresponding to a specific state, so the state we are interested in is passed in from the Flash client (details on that to come).

Now, let's add the last two functions (getParkDetails and escape) to the file:

```
function getParkDetails(park) {
    queryData.sql = "SELECT region, parktype, comphone FROM tblParks " +
                    "WHERE parkname='" + escape(park) + "'";
    var details = CF.query(queryData);
    return details;
}

// Escape strings so that they can be passed into a SQL statement by
// replacing single quotes (') with two single quotes ('').
function escape(str) {
    var escapedStr = new String();
    for (var i = 0; i < str.length; ++i) {
        if (str.charAt(i) == "'") {
            escapedStr += "'";
        }
        escapedStr += str.charAt(i);
    }
    return escapedStr
}
```

Save the file.

The getParkDetails function is fairly straightforward, and only differs from getParks in that it retrieves three columns of data rather than one. This means we will handle the RecordSet object on the client slightly differently (more on that to come). It also passes the "park" variable through the escape function before passing it into the query function.

The escape function searches through the string (in this case, the park name) character by character looking for a single quotation mark. If it finds one, it adds a second single quote to the string before the first. This is called SQL escaping. Since strings have to be inside of single quotes in SQL, you obviously cannot have any single quotes in the string that is inside the single quotes. If you do, the standard is to "escape" the single quotes by replacing them with two single quotes.

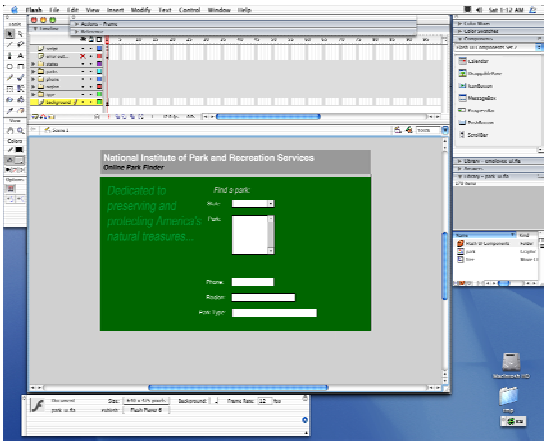
Now, we'll concentrate on the Flash movie so we can see how to make use of this powerful new server component of Flash application development.

On the Client-Side

In the previous section, we defined three server-side functions using Server-side ActionScript. Now, ColdFusion has all it needs to make them available to a Flash client. The Flash Remoting Service that is part of ColdFusion MX automatically handles all of the communication between the Flash client and the ColdFusion server. To see how it works, let's switch to the client side of our application.

The Flash file we are interested in is called park_ui.fla. There are actually two versions of the park_ui.fla included with this tutorial. park_ui.fla contains all the code from this tutorial already written. park_ui_empty.fla does not contain any ActionScript and therefore will allow you to write the ActionScript while following along with this tutorial. Go ahead and open up whichever file you choose in your Flash MX authoring environment and let's take a look at how it works behind the scenes.

As far as Flash files go, this one is pretty straightforward (remember, this is just a proof of concept – you'll dress it up later). All you will find on the stage are a few components and text fields. The topmost layer (called "script") is where we will be writing the ActionScript for this application. The layer below that contains a MessageBox component with the instance name of "errorAlert." As you will see, we will use this for displaying error messages. The rest of the layers contain a ComboBox (with the instance name of "states"), a ListBox (instance name of "parks"), and the text fields we use to display park details (with the instance names of "phone", "region", and "partType").



Before we actually start writing code, let's examine at a high level application how this application will work. The code explained below will carry out the following steps:

1. Configure a connection to the ColdFusion MX server.
2. Create objects with the callback functions the server needs to talk back to the client.
3. Invoke remote Server-side ActionScript functions, passing them the objects we created with the callback functions.
4. Use the data returned from the server to dynamically populate a ComboBox, ListBox, and a few TextFields.

Callback functions are a very useful way for Flash developers to handle some kinds of events. For instance, when you invoke a remote server-side function, there are two ways the Flash Player could handle returning the result. The movie could either stop while waiting on the response (a process is said to "block" when it pauses, waiting on another process), or the movie could go on, and when then results are returned, a callback function could automatically be called. The Flash player uses the latter method. While at first this may seem slightly more complex, it is by far the more powerful model. Using callback functions, your application can continue executing while the data Is being retrieved, so you can show the user additional information or let them continue working. We will look at callback functions in more detail once we begin to write a few.

What You Learned	Callback functions are a more sophisticated way to handle responses from servers.

Writing the Client-Side ActionScript Code

At this point, we're ready to start writing our client-side ActionScript code. The first few lines simply help set up the rest of the application by including an object we need and setting up a few variables.

Open the ActionScript editor to the first frame of the "script" layer and add the following:

```
#include "NetServices.as"

// Make sure you remove the include below in a production environment.
// It is for debugging and development purposes only because it causes
// unnecessary performance degradation.
#include "NetDebug.as"

stop();

// Define some constants.
var GATEWAY_URL = "http://localhost:8500/flashservices/gateway";
var SERVICE = "com.macromedia.tutorial.ParkQuery";
```

The first line is including the NetServices ActionScript file which you should have installed in your ActionScript "include" directory. We will get to the importance of the NetServices object soon enough; for now, just assume that it is important and must be included early on.

The next line includes the NetDebug object. NetDebug enables the "NetConnect Debugger," which you can find under the Window menu. The NetConnect Debugger is a very useful tool for debugging Flash Remoting code because it reports on the traffic between the client and the server, allowing you to pinpoint exactly where problems are occurring. Make sure you do not include the NetDebug object in your final movie because logging causes significant performance degradation. It's not an issue for development, but you wouldn't want it in your deployed application.

The following line stops the movie since we are not concerned with any type of animation.

The next two lines define a pair of "constants," variables whose values will not change throughout the lifecycle of the application (constants should always be capitalized). The first one, GATEWAY_URL, points to your ColdFusion MX server (Remember: this tutorial assumes your are running ColdFusion and Flash MX on the same computer. If you have configured things differently, you'll need to adjust the server name and/or port number accordingly).

The "flashservices/gateway" path does not indicate an actual file on your server; rather, it is an alias for the Macromedia Flash Remoting Service. The Remoting Service is a service which comes pre-installed and pre-configured with ColdFusion MX and is responsible for handling communication between Flash clients and server-side services.

The value of the SERVICE constant should look vaguely familiar to you. Remember the directory structure you created inside of the wwwroot directory on the server for the ParkQuery.asr file? On the client, you refer to services by their filenames, relative to the wwwroot directory with dots in place of slashes or backslashes. Eventually, you will see how we use the value of the SERVICE constant to actually reference remote services.

What You Learned	Server-side services are referenced from the client by their full names with dots in place of slashes or backslashes. Constants are variables expressed in all caps which are intended not to change throughout the execution of the application.

Add the next few lines to the actions panel. They help set up the connection we are going to need to make to the Remoting Service so we can invoke our remote functions.

```
// Set the default gateway on the NetServices object and go ahead and create a
// connection object.
NetServices.setDefaultGatewayUrl(GATEWAY_URL);
var con = NetServices.createGatewayConnection();
```

We make use of our first constant here by passing it in as an argument to the setDefaultGatewayUrl function on the NetServices object. The NetServices object provides us with connections to the Remoting Service, so naturally it needs to know which RemotingService to provide connections to.

The createGatewayConnection function on NetServices is a little misleading since it does not actually open a socket and create a connection between the client and the server at the time the function is called. Rather, it creates a NetConnection object, which we will later use to make the actual connection.

What You Learned	Despite its name, calling createGatewayConnection on the NetServices object does not actually establish a connection. Connections are not made to the server until the actual remote function is invoked.

Writing an Initialization Function

Most applications need to be initialized in some way. Application initialization ensures that certain pieces of code have already been run and certain objects and functions are already set up before the user starts interacting with the application. Initialization is essentially a guarantee that certain actions have occurred in your code so that you can make assumptions based on those actions.

Our Macromedia Flash application needs an initialization sequence in order to perform several actions:

- Configure the alert box that gets used to display error messages if something has gone wrong on the server, then hide it until it's needed.
- Set the size (width) of the state ComboBox so that it doesn't look too big.
- Set up an object with callback functions that we will use to query the database for a list of states to populate the states ComboBox with.
- Invoke the remote function which retrieves and returns the list of states.
- Set the rows (height) and width of the ListBox which eventually gets used to display the list of parks in a specified state.

What You Learned	Most applications need some form of initialization to prepare them for user interaction. Initialization ensures that certain variables and functions have been properly set up.

Add the following code to the actions panel below the code you have already written. Note that initialization code should always be the last ActionScript code in the panel, so all of the rest of the ActionScript we write for this tutorial will actually go *above* this initialization code.

```
// Initialize the application.
```

```
function init() {
        // Hide the error MessageBox and give it a title.
        errorAlert._visible = false;
        errorAlert.setTitle("application error occurred");
// Set the size of the state ComboBox.
states.setSize(50);

// Create a handler for the states combobox.
var resultHandler = new Object();
resultHandler.getStates_Result = function (result) {
        states.setDataProvider(result);
        states.setChangeHandler("onStateChange");
};

resultHandler.getStates_Status = handleError;

// Get the ParkQuery service from the gateway connection.
var service = con.getService(SERVICE, resultHandler);

// Invoke the getStates() function on the server.
service.getStates();

// Set the size and width of the listbox that displayes the park names.
parks.setRowCount(8);
parks.setWidth(310);
parks.setChangeHandler("onParkChange");
}

// Call the init function to initialize the application.
init();
```

The first few lines are straightforward enough. They give the MessageBox component a title and hide it until such time as we need it (hopefully never!). We then set the width of the ComboBox component to something relatively small since all it needs to display are state abbreviations. Of course, you can also configure components through the property window; however, many find it is easier to go through a single panel of ActionScript code to make changes than it is to dig through multiple levels of movie clips.

Now this is probably where the code starts to get into unfamiliar territory. Let's take a moment to examine this line in particular:

```
// Get the ParkQuery service from the gateway connection.
var service = con.getService(SERVICE, resultHandler);
```

At this point, we are actually getting a reference to our remote ParkQuery service from our NetConnection instance. The first argument to getService() is the SERVICE constant which we defined at the top of our file. The second argument, however, is something a little new.

As mentioned previously, Flash Remoting uses a callback paradigm. In other words, when you execute a remote function, the client code does not block (stop or pause) to wait for a response from the server. Rather, execution continues, and you know that your data was returned when your callback function is called.

That brings us to the second argument of the getService function, resultHandler. resultHandler is an instance of an object created especially for handling callbacks from the server. The code just above the getService function builds the resultHandler object:

```
var resultHandler = new Object();
resultHandler.getStates_Result = function (result) {
```

```
            states.setDataProvider(result);
            states.setChangeHandler("onStateChange");
};
resultHandler.getStates_Status = handleError;
```

What You Learned	When you invoke a remote function on the server, application execution does not block or pause. The movie continues to play, and your callback function gets called when then data is returned from the server.

Note: Typically, you would not create objects in this fashion as it is not the most efficient technique in cases where you need to make more than one instance of the object. Creating a generic object instance and assigning that specific instance properties and functions means that every instance created in that manner will require new memory allocation to store those functions and properties. An alternative and much more efficient method for creating objects is to create objects from classes. We'll explore this later. In this case, we can get away with this particular technique only because we are creating a single instance of the resultHandler and we know that the init function is only going to get called once.

After creating a new resultHandler object, we assign it two functions. The first function is called getStates_Result. It probably already occurred to you that getStates is the name of the first function in our ParkQuery Server-side ActionScript file. The prefix "_Result" tells the server that this is the callback function for the getStates server-side function. As you can see, the server passes back a data structure which we call "result". If you refer back to our Server-side ActionScript file, you will recall that the object that getStates returns is a RecordSet of data queried from the database. In this particular case, the RecordSet contains state abbreviations.

Although we have set up a callback function to handle the response from the server, that function will not get called until we actually invoke the remote service. That's done by the following code.

```
// Invoke the getStates() function on the server.
service.getStates();
```

Once we are inside the getStates_Result function, we know for a fact that the server has returned a RecordSet object. The question is, what do we do with it? It just so happens that a RecordSet object extends the DataProvider object, which happens to be the data type that most UI components take in their setDataProvider methods. Therefore, all we have to do is pass the RecordSet object into the setDataProvider function of the states ComboBox, and the ComboBox automatically updates itself with the list of state abbreviations. If you have ever achieved a similar effect by dynamically building an HTML select input, then you can appreciate how wonderfully simple and elegant this technique is.

The last thing we do in the getStates_Result function is set a changeHandler on the states ComboBox so that we know when the user has selected a state. By passing in the string "onStateChange", our onStateChange function (which we will review shortly) is automatically called whenever the ComboBox gets changed.

Now what if something went wrong on the server and the RecordSet object was not returned? What if the database was down, or there was a syntax error in our Server-side ActionScript code? In such a case, the second function we assigned to resultHandler gets called: getStates_Status. Rather than implementing the getStates_Status function at the moment of assignment, we assigned it to another generic error-handling function instead called handleError. handleError is a very simple function. It gets the "description"

property of the error object passed in from the server and displays the message in the errorAlert MessageBox component.

The remainder of the init method simply sets the height and width of the parks ListBox component and assigns it a changeHandler. As always, more on that changeHandler to come.

Since all of this initialization code is inside of a function (called init, a very common name for functions which initialize applications), it will not run until it is explicitly called, which we do with the very last line of code in the actions panel:

```
// Call the init function to initialize the application.
init();
```

What You Learned	Some Flash UI Components can take in a DataProvider which RecordSet extends. That means you can pass query results directly into a UI component, and the component will automatically update itself with the data from the ResultSet.

Now is probably as good a time as any to ask ourselves why we would want to go through all this trouble to build the states ComboBox dynamically as opposed to just hard-coding a list of states in the Flash file. It may be easier to hard-code the data initially, but in the long run, it always pays to take a little extra time to do things in a way that makes it easier to maintain. In your case, you wouldn't want to add and remove states every time a new entry made it into the database. Or what if this were an application that you needed to deliver to a client? How many times would they pay you to update a simple ComboBox before turning the account over to your competition who promised a maintenance-free application? The truth is that we live in a dynamic word, and the only way to truly model it and keep up with it is though dynamic processes and applications.

What You Learned	Dynamic is good, hard-coding is bad. Hard-code that which will almost never change; everything else, externalize.

Creating Classes

If you were to publish the application at this point, you should understand everything you see. You should see the states ComboBox automatically populated with all the unique states from the database, ready for a user to interact with. Before writing code to handle the event of a state being chosen, however, we need to write a few class definitions.

The first class will contain callback functions for the getParks server-side function. That means that we will create a new object from this class and pass it into the getService function of our NetConnection instance.

Above the init function definition, add the following code:

```
// Defines the ParkHandler class.
function ParkHandler() { };
ParkHandler.prototype.getParks_Result = function(result)
{
        parks.setDataProvider(result);
}
ParkHandler.prototype.getParks_Status = handleError;
var parksQuery = con.getService(SERVICE, new ParkHandler());
```

Remember when we discussed a more efficient way to create objects than the method used in the init function? Well, this is it. The function definitions above don't actually create an instance of anything; rather they define a class from which instances of objects can be made. Assigning functions and properties to the prototype property of a class makes for a more efficient use of memory since new memory is not allocated for every function and property of every instance; rather, memory is allocated at the time you assign properties and functions to the prototype property, and that same memory space is referenced for every instance you create. Using more efficient memory allocation on the client means that your movie (and any other applications the user has open) will execute more efficiently.

As you already know, the ParkHandler class contains callback functions for the getParks Server-side ActionScript function. Notice how handleError gets reused once again for handling errors (naturally), and how we call getService on our NetConnection instance to set up our service for later when we're actually ready to call it. Look inside the getParks_Result function and you will see a familiar technique for mapping a RecordSet to a client-side UI component. In this case, we are dynamically populating the parks ListBox with names of parks.

What You Learned	There are two ways of creating your own objects in Flash. Make sure you choose the way that will lead to the most efficient code. If you only need a single instance of the object, it is ok to create an generic object instance, then add functions and properties to that instance. If you need to create more than one instance, create a class by assigning properties and functions to the prototype property of the class and create instances from the class.

We have one more class definition to write since we have one more function on the server which needs callback functions. The class definition we are about to write will contain callback functions for the getParkDetails server-side function. The getParkDetails_Result function is going to be a little different than the other callback functions we have written so far because we cannot map the RecordSet directly to a UI component. Instead, we will extract the data from the RecordSet and use it to populate dynamic TextFields.

Add the following code to your actions panel above the init function definition:

```
// Defines the DetailsHandler class.
function DetailsHandler() { };
DetailsHandler.prototype.getParkDetails_Result = function(result) {
        var UNKNOWN = "(unknown)";
        var row = result.getItemAt(0);
        phone.text = (row["comphone"] == null) ? UNKNOWN : row["comphone"];
        parkType.text = (row["parktype"] == null) ? UNKNOWN : row["parktype"];
        region.text = (row["region"] == null) ? UNKNOWN : row["region"];
}
DetailsHandler.prototype.getParkDetails_Status = handleError;
var detailsQuery = con.getService(SERVICE, new DetailsHandler());
```

Let's take a closer look at the implementation of the getParkDetails_Result function. Since the details of a park are being displayed in dynamic TextFields rather than being mapped to a UI component, the code isn't quite as clean and elegant as we have seen in previous callback implementations. The good news is that this gives us the opportunity to explore a little more closely the anatomy of a RecordSet.

A RecordSet contains data structures that represents rows returned from a query. You use the getItemAt function to access individual rows. In the example above, we only expect a single row to be returned, so we know we always want the first item (which has an index of 0). As shown in the example above, once you have a reference to the row, you can access the row's data through the name of the column in square brackets. You could also access the row's data using dot notation, instead. For instance:

```
row["parktype"]
```

and

```
row.parktype
```

will return the same result. As you can see, we simply assign the three different column values to the text properties of our three different TextFields, and they automatically update themselves.

What You Learned	RecordSets are very powerful objects that model database query results. Although they may seem a little complex at first, they are very well designed data structures.

Writing Change Handlers

Let's review what happens when a user picks a state from the states ComboBox:

1. When a user selects a state, it triggers the changeHandler on the states ComboBox.
2. Our onStateChange function (which we will write momentarily) gets called and passed a reference to the states ComboBox.
3. The onStateChange function finds out which state was chosen, then calls the remote getParks function on the server, passing in the name of the chosen state.
4. The ParkHandler's getParks_Result callback function gets called when the server returns the data. The resulting RecordSet object is used to populate the parks ListBox.

Let's go ahead and write the changeHandler for the states ComboBox.

Add the code below above the init function definition:

```
// This function is called whenever a state is chosen. It updates the list of
// parks in order to correspond to the selected state.
function onStateChange(statebox) {
        var selectedState = statebox.getSelectedItem();
        parksQuery.getParks(selectedState.state);
}
```

The "statebox" argument that gets passed into the onStateChange function is a reference to the ComboBox that was just changed. To find out which item was selected, we call the getSelectedItem function on the instance of the ComboBox. getSelectedItem returns an object that contains a property corresponding to the column name of the RecordSet object that we used to build the ComboBox. "selectedState.state", therefore, returns the name of the selected state since "state" refers to the state column of the tblPark database table. As you can see, the state name is getting passed into the getParks remote function call on the parksQuery service. This is where the connection to the Remoting Service is actually made. ColdFusion executes the query on the server, and the RecordSet is returned. Where does it get returned to? An instance of the ParkHandler class, which proceeds to set the RecordSet on the "parks" ListBox component.

When a user selects a park from the "parks" ListBox, the process is almost identical, except that the onParkChange changeHandler is called instead of the onStateChange changeHandler. Before we write the changeHandler for the ListBox, let's review the process:

1. The user selects a park, triggering the parks ListBox's changeHandler which we assigned inside of the init function.
2. Our onParkChange function (which we will write momentarily) gets called and passed a reference to the parks ListBox.
3. The onParkChange function finds which park was chosen, then calls the remote getParkDetails function on the server, passing in the name of the chosen state.
4. The DetailsHandler's getParkDetails_Result callback function gets called when the server returns the data. Each column is extracted from the RecordSet object and set on one of the three dynamic TextFields.

Let's go ahead and write the onParkChange changeHandler function. Its only job will be to find out which park was chosen in the ListBox and call the getParkDetails remote function, passing it the name of the chosen park. Add the code below to the actions panel above the init function definition.

```
// This function is called whenever a specific park is chosen.  It updates
// the park detail TextBoxes in order to correspond to the selected park.
function onParkChange(parkbox) {
        var selectedPark = parkbox.getSelectedItem();
        detailsQuery.getParkDetails(selectedPark.parkname);
}
```

As you can see, the getParkDetails remote function is called on detailsQuery, and when the data gets returned from the server, the getParkDetails_Result function of the DetailsHandler will be called and passed the RecordSet containing the details we are interested in. As we already discussed, the data is extracted from the RecordSet object and set on the details TextFields.

Publishing the Application

Go ahead and publish the application to the directory we created at the beginning of the tutorial. Make sure your ColdFusion server is running on a computer that is in the same domain as the computer your Flash movie is on or else the Flash player will not be allowed to communicate with the server due to security restrictions.

Double-click on the generated HTML file to load the movie into your default browser and start exploring all the parks in the US. Congratulations! You have just built a fully functional application which integrates a rich Flash user interface with ColdFusion MX.

Wrapping It Up

You demo your application to your manager and, naturally, he is amazed. Of course, he tries to take credit for it as you demo it to your president, but your president is too smart for that and you are given the promotion you deserve. What everyone is most impressed with is the time it took you to develop a robust, working prototype. Had you opted for any other technology, the job might have taken days of set-up, configuration, reading documentation, development, and finally testing. Fortunately, Macromedia has given you enough tools that putting together a dynamic client/server application is literally as simple as snapping together a few components. This is when the possibilities really start to dawn on you.