FAST TRACK TO JQUERY MOBILE AND COLDFUSION 10

(This page intentionally left blank)

Integrating JQM with ColdFusion 10

Unit Objectives

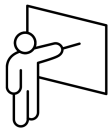
After completing this unit, you should be able to:

- Define and use AJAX/JSON(P) web services using ColdFusion 10
- Define and use REST web services using ColdFusion 10
- Using REST services with jQuery Mobile
- Implement real-time communications with HTML5 web sockets

Unit Topics

- Delivering Client-Independent Data
- Creating and Invoking REST(ful) Web Services
- Implementing Push with Web Sockets and ColdFusion 10

Delivering Client-Independent Data



Support for AJAX web services have been provided by ColdFusion long before version 10, and it has evolved into supporting/integrating with a wide variety of both data-centric and user-interface-centric components with the language.

Defining Simple AJAX Web Services

AJAX services in ColdFusion are defined as a set of server side services paired with matching client side JavaScript services for passing data between client and server in a structured fashion without the need for a full page reload.

In order to keep a clean separation of duties between the client and the server, you will use ColdFusion strictly for querying and delivering data in either XML or JSON format.

ColdFusion components lend themselves well to this architecture through its support for enabling methods to be executed via an HTTP get request and its ability to translate complex data structures into JavaScript Object Notation.

Defining URL accessible methods

ColdFusion component methods may be invoked via an HTTP `get` operation as long as their access property is set to `remote`. Data can be returned as either XML or JSON using the `returnformat` property of the cfc method.

Therefore, a ColdFusion CFC method that services AJAX requests from JQM typically resembles the following:

```
<cfcomponent>

<cffunction name="getData"
            access="remote"
            returntype="array"
            output="false"
            returnformat="json">

</cffunction>

</cfcomponent>
```

Presuming that the file containing the component was stored at http://localhost/mycomponent.cfc, you could invoke the getData method directly using the following url:

```
http://localhost/mycomponent.cfc?method=getdata
```

Converting Queries to an Array of Structures

Unfortunately, ColdFusion query results do not convert to JavaScript in a format that is very easy for JavaScript frameworks to consume. Most frameworks, including JQM, expect to receive an array of javascript objects, which is analogous to a ColdFusion array of structures. Serializing a ColdFusion query generates a multi-dimensional indexed array.

While there is no built-in function for converting a CFQUERY recordset into an array of structures, you can easily develop your own as illustrated by the following code snippet:

```
<cffunction name="query2array" access="private"
            returntype="array" output="false">

  <cfargument name="qdata" type="query" required="yes">

  <cfset local.i = "0">
  <cfset local.stdata = structnew()>
  <cfset local.thiscolumn = "">
  <cfset local.aresult = arraynew(1)>

  <cfloop from="1" to="#qdata.recordcount#" index="i">
      <cfset local.stdata = structnew()>
       <cfloop list="#qdata.columnlist#"
               index="local.thiscolumn">
          <cfset stdata[lcase(local.thiscolumn)] =
                                qdata[local.thiscolumn][i]>
      </cfloop>
      <cfset local.aresult[i] = local.stdata)>
  </cfloop>

  <cfreturn aResult>
 </cffunction>
```

Using this methodology, returning an array of structures to JavaScript enables you to reference query data in the following format:

```
query[recordnumber].columnName
```

Using the SerializeJSON method

In ColdFusion 10 the serializeJSON() method has been enhanced to return query data in a more suitable format for JavaScript web apps, eliminating the need to call a query2array converter. However, in order to call serializeJSON() explicitly, you need to modify your method signature as illustrated by the following example:

```
<cffunction name="getDataJSON"
            access="remote"
            returntype="string"
            output="false"
            returnformat="plain">

  <cfset local.q = "">
  <cfquery name="local.q">
       select firstname, lastname
       from people
  </cfquery>

  <cfreturn serializejson(q,true)>
</cffunction>
```

```
{
    "ROWCOUNT": 3,
    "COLUMNS": [
        "FIRSTNAME",
        "LASTNAME"
    ],
    "DATA": {
        "FIRSTNAME": [
            "Steve",
            "David",
            "Dave"
        ],
        "LASTNAME": [
            "Drucker",
            "Gallerizzo",
            "Horan"
        ]
    }
}
```

Figure 1: Sample Output from the getDataJSON method

Note the following:

- The `returntype` is set to `string`
- The `returnformat` is set to `plain`
- The second argument to `serializeJSON` causes it to return a Javascript object whose properties contain an array of column values.

Using the methodology described above, in JQM you would reference the data from the query using the following syntax:

```
result.DATA.COLUMNNAME[rownum]
```

Removing High Ascii Characters

Users of web content management systems have an annoying habit of occasionally copying and pasting content from Microsoft Word into their web forms. Unfortunately, this can sometimes lead to special characters (i.e. smart quotes, ellipsis) getting inserted into your database. These "high ascii bit" characters can cause your JSON data to become invalid.

Therefore, you may want to programmatically filter all high ascii bits from your query results by invoking the CleanHighAscii() method listed on the following page[1]:

```
<cffunction
	name="CleanHighAscii"
	access="public"
	returntype="string"
	output="false"
	hint="Cleans extended ascii values to make web safe">

	<!--- Define arguments. --->
	<cfargument
		name="Text"
		type="string"
		required="true"
		hint="The string to clean"
		/>

	<cfset local.Pattern = CreateObject(
		"java",
		"java.util.regex.Pattern"
		).Compile(
			JavaCast( "string", "[^\x00-\x7F]" )
			)
		/>

	<cfset local.Matcher = local.Pattern.Matcher(
		JavaCast( "string", ARGUMENTS.Text )
		) />

	<cfset local.Buffer = CreateObject(
		"java",
		"java.lang.StringBuffer"
		).Init() />


	<!--- Keep looping over high ascii values. --->
	<cfloop condition="local.Matcher.Find()">

		<!--- Get the matched high ascii value. --->
		<cfset local.Value = local.Matcher.Group() />

		<!--- Get the ascii value of our character. --->
		<cfset local.AsciiValue = Asc( local.Value ) />
```

1 Developed by Ben Nadel http://www.bennadel.com/blog/1155-Cleaning-High-Ascii-Values-For-Web-Safeness-In-ColdFusion.htm

```
                        <cfif (
                               (local.AsciiValue EQ 8220) OR
                               (local.AsciiValue EQ 8221)
                               )>

                               <!--- Use standard quote. --->
                               <cfset local.Value = """" />

                        <cfelseif (
                               (local.AsciiValue EQ 8216) OR
                               (local.AsciiValue EQ 8217)
                               )>

                               <!--- Use standard quote. --->
                               <cfset local.Value = "'" />

                        <cfelseif (local.AsciiValue EQ 8230)>

                               <!--- Use several periods. --->
                               <cfset local.Value = "..." />

                        <cfelse>
                          <cfset local.Value =
                               "&###local.AsciiValue#;" />
                        </cfif>

                        <cfset local.Matcher.AppendReplacement(
                               local.Buffer,
                               JavaCast( "string", local.Value )
                               ) />

                </cfloop>

                <cfset local.Matcher.AppendTail(
                       local.Buffer
                       ) />


                <!--- Return the resultant string. --->
                <cfreturn local.Buffer.ToString() />
        </cffunction>
```

Making AJAX Data Requests from JQM

You can make AJAX requests to a server using the $.ajax() method illustrated below:

```
$.ajax({
    url: 'myservice.cfc?method=getdata',
    type: 'GET',
    dataType: 'json',
    error : function (){ alert('there was an error'); },
    success: function (data) {
       console.log(data);
       // debugger;
    }
});
```

Note the following:

- The callback to the success or error handler is executed asynchronously.
- The success handler receives the data as a javascript object. You do not need to use the eval() method or an equivalent to parse the JSON into a native JavaScript object.
- Use the console.log() method to output results to your debugger. Alternately, you can use the debugger; command to set a programmatic breakpoint.

Handling HTTP Request Payloads

Some JavaScript frameworks (Ext JS, Sencha Touch) post/put JSON data through the http request payload. In these cases, you'll need to access the data by converting the request data to a string and then deserialize it into a native ColdFusion datatype as illustrated below:

```
<cfcomponent>
 <cffunction
    name="saveData"
    access="remote"
    returntype="string"
    returnformat="plain">

   <cfset var requestData = deserializeJson(
              toString(getHttpRequestData().content)
   )>

   <!--- data now available as requestData.fieldName --->
   <cfreturn serializeJson({success=true})>
 </cffunction>
</cfcomponent>
```

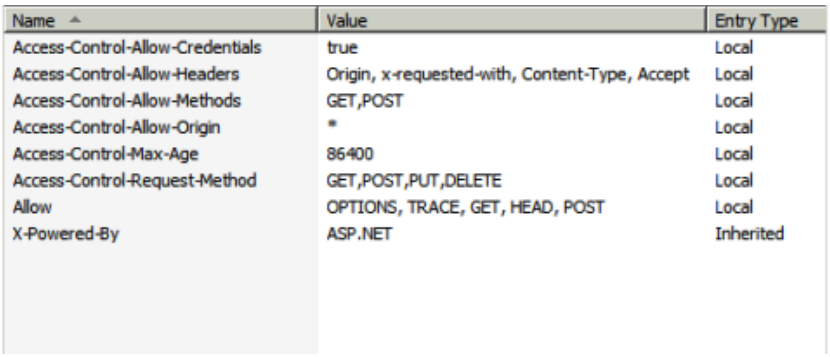
In some use- cases, nulls may be present in the requestData variable, leading you to use isDefined() or <cfparam> to normalize the dataset:

```
<cfcomponent>
 <cffunction
    name="saveData"
    access="remote"
    returntype="string"
    returnformat="plain">

   <cfset var requestData = deserializeJson(
              toString(getHttpRequestData().content)
   )>

   <!--- normalize foo if null --->
   <cfif not isdefined("requestdata.foo")>
     <cfset requestdata.foo = "">
   </cfif>

   <!--- data available as requestData.fieldName --->

   <!--- return success=true as json --->
   <cfreturn serializeJson({success=true})>

 </cffunction>
</cfcomponent>
```

Implementing Cross-Origin Resource Sharing

CORS (Cross-Origin Resource Sharing) enables you to run your app from a domain other than where your web services are being hosted. This has a number of advantages, the least of which is that you can get away from using the kludgey JSON-P proxy and rely on AJAX/REST for all of your get/post server transactions while maintaining the flexibility of launching your app from anywhere (including localhost).

Configuring CORS on IIS 7.5 required setting the following http headers:



Name	Value	Entry Type
Access-Control-Allow-Credentials	true	Local
Access-Control-Allow-Headers	Origin, x-requested-with, Content-Type, Accept	Local
Access-Control-Allow-Methods	GET,POST	Local
Access-Control-Allow-Origin	*	Local
Access-Control-Max-Age	86400	Local
Access-Control-Request-Method	GET,POST,PUT,DELETE	Local
Allow	OPTIONS, TRACE, GET, HEAD, POST	Local
X-Powered-By	ASP.NET	Inherited

Figure 2: Configuring IIS to support CORS

While CORS is generally supported by modern browsers, it is not supported by IE 6 or IE 7. See http://caniuse.com/cors to verify that your target browsers are supported.

Creating a JSONP Service

JSONP or "JSON with padding" is a complement to the base JSON data format. It provides a method to request data from a server in a different domain, something prohibited by typical web browsers due to security restrictions built into the XMLHttpRequest object.

Under the same origin policy, a web page served from server1.example.com cannot normally connect to or communicate with a server other than server1.example.com. Exceptions include the HTML <script> element, <link> element, and <img> elements. Exploiting the open policy for <script> elements, some pages use them to retrieve JavaScript code that operates on dynamically generated JSON-formatted data from other origins. This usage pattern is known as JSONP. Requests for JSONP retrieve not JSON, but arbitrary JavaScript code. They are evaluated by the JavaScript interpreter, not parsed by a JSON parser.

JQM supports making JSONP service requests. Note that in JSONP, the javascript framework typically transmits a dynamically constructed callback argument to the server-side function. The server must, in turn, wrap JSON with this function call.

In order to handle the request, you must modify your ColdFusion CFC remote method function to resemble code listing on the following page:

```
<cffunction name="getDataJSONP"
            access="remote"
            returntype="string"
            output="false"
            returnformat="plain">
  <cfargument name="callback" type="string" required="yes">

  <cfset local.q = "">
  <cfquery name="local.q">
       select firstname, lastname
       from people
  </cfquery>

  <cfreturn arguments.callback &
                          "(" & serializejson(q,true) & ")>
</cffunction>
```

Making JSONP Requests from JQM

You can make cross-domain JSON-P requests from jQuery using the $getJSON method illustrated below:

```
var url='http://someurl/somecfcomponent.cfc?';

$.getJSON(url + 'method=somemethod&callback=?',
          function(data) {
            console.log(data);
          }
);
```

Note that the ? Used in the callback url is replaced at runtime by a jQuery with a randomly generated set of numbers. JSONP GET requests therefore typically resemble the following:

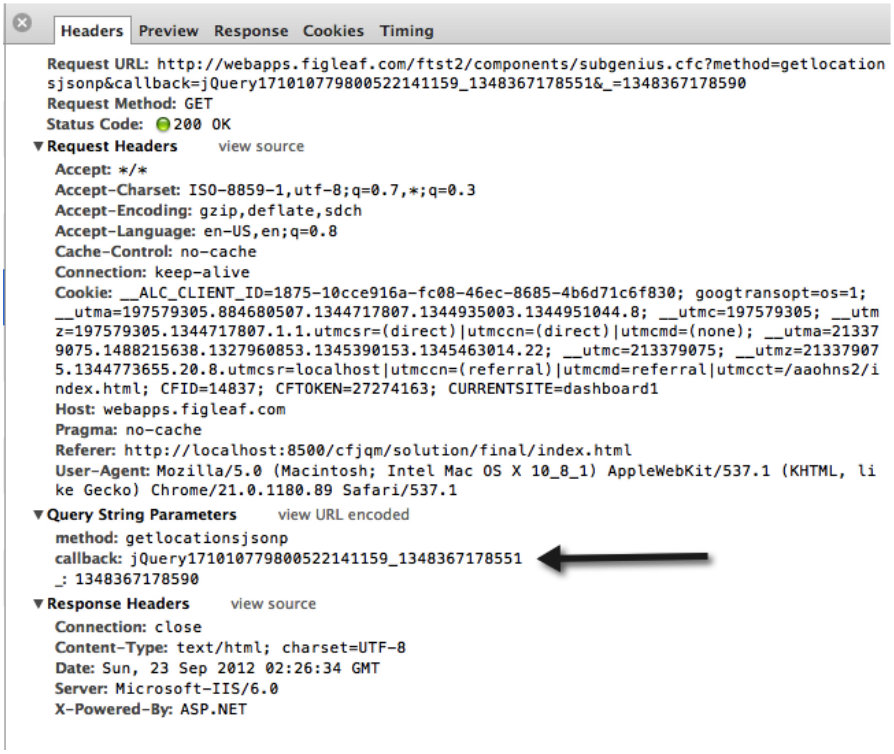


Figure 3: A typical JSONP Request.

Deferring Data Requests Until Page Activation

One of the challenges that you will have with building larger, more complicated JQM apps is minimizing the amount of memory that your app consumes. One strategy for managing memory is to not load data until it is absolutely needed. For instance, if a data request is required to populate a select list, you might want to defer making the request until such time as the page containing the select box becomes visible.

The typical pattern to trigger the execution of code on page visibility is the following:

```
$('#page2').live("pagecreate", function() {

  // execute code when page 2 gets instantiated

}
```

Walkthrough 9-1: Making Data Requests to CF

In this walkthrough, you will start developing the "Campus Map" feature for the fictional "SubGenius University"

- Create a ColdFusion CFC method that outputs location information as JSON.
- Make an AJAX request from JQM to your ColdFusion server.
- Dynamically populate a <select> box with data from the AJAX request.

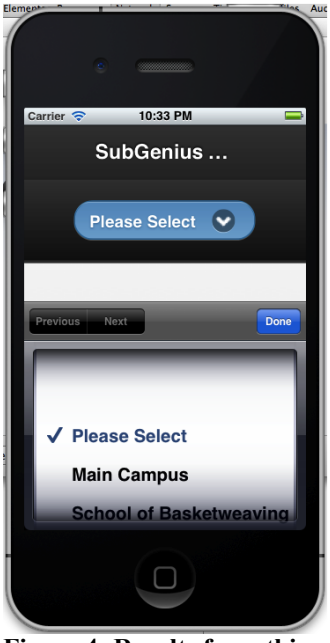


Figure 4: Results from this walkthrough

Steps

Review the Starter Code

1. Open **/walk/walk1/index.html** in your editor and review the code with your instructor.

Create a Component Method

2. Open /walk/walk1/data/subgenius.cfc.
3. Where indicated by the comment, define a new function that has the following attributes:
   - name: getLocations
   - access: remote
   - returntype: string
   - output: false
   - returnformat: plain
4. Inside the function, define a local variable named q and initialize it as an empty string.
5. After the code that you inserted in the prior step, define a <cfquery> with the following attribute:
   - name: local.q
6. Inside the <cfquery>, type the following sql:

   select * from location

7. After the </cfquery>, return the results from the query, serialized as a JSON object. Your code should appear as follows:

   <cfreturn serializeJson(local.q,true)>

8. Save the file.

9. Test the method by entering the following URL into your web browser:

```
http://localhost:8500/cfjqm/walk/walk1/data/subgenius.cfc?
method=getlocations
```

Make an AJAX Request

10. Return to **/walk1/index.html**

11. Where indicated by the comment, define an event listener that is triggered when a request is made to display #page2.

```
$('#page2').live("pageshow", function() {

}
```

12. Inside the function that you defined in the prior step, insert code to make an AJAX request to the CFC method that you defined earlier in this walkthrough. Your code should resemble the following:

```
$.ajax({
    url: ajaxUrl + "method=getlocations",
    type: 'GET',
    dataType: 'json',
    error : function (){
       alert('there was an error');
       console.log(arguments)
    },
    success: function (result) {

    }
});
```

13. Inside the success function, output the result variable to the browser's console.

14. Save the file and test. Note the contents of the result variable in the browser's console.

15. Return to your editor.

Add Options to a Select Box

16. Inside the success function of the ajax callback, define a local variable named q that points to the DATA property from the returned dataset.

```
var q = result.DATA;
```

17. Immediately after the code that you defined in the prior step, insert a for-loop that iterates from zero to the number of rows returned from the ColdFusion method call.

```
for (var i=0; i<result.ROWCOUNT; i++) {

}
```

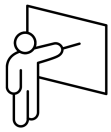

18. Inside the for-loop, add entries to the <select> box using the LAT/LNG values from the data request as the option values and the TITLE values as the option text values. Your code should resemble the following:

```
$('#campusLocationSelect')
        .append($("<option></option>")
        .attr("value",q.LAT[i] + ',' + q.LNG[i])
        .text(q.TITLE[i]));
```

19. Save the file and test.

– End of Walkthrough --

Creating and Invoking REST(ful) Web Services



REST, or Representational State Transfer, is an architectural style that specifies constraints and conventions to web services to induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST pattern, data and functionality are considered resources and are accessed using Web links.

The intent of REST services is to be simple, efficient, and fast. To achieve this, they stick to the following principles:

- **Resource Identification through the URI**
  The URL you are using to call the service will a very good indicator of the path, structure, and type of service you are calling.
- **Consistent Interface**
  Resources are manipulated through a fixed set of services: PUT, GET, POST, and DELETE operations.
- **Stateless transactions**
  Each transaction with a resource is stateless, allowing any individual system using the resource to implement to rely on it's own state management mechanism(s) for managing state between HTTP requests.

Defining REST Services

With ColdFusion 10, standard component definition practices apply, with ColdFusion handling much of the detail work in integrating the HTTP methods into the component architecture, data serialization, and URI construction.

- ColdFusion REST services follow the standard HTTP request-response model. Beyond having HTTP as a medium, the service lets you follow all HTTP norms. The components published as REST services can be consumed by HTTP/HTTPS requests.
- The REST methods of GET, POST, PUT, DELETE are defined as methods within your component.
- ColdFusion natively supports JSON and XML serialization/deserialization. Client applications can consume REST services via HTTP/HTTPS requests in XML or JSON format. ColdFusion will determine the format based on the extension of the resource in the URI.
- You can create and publish the same ColdFusion component as a REST service and WSDL service.

Analyzing a REST Service Call

REST service calls are made through a URL-like construct. These URLs are made up of several parts, which are comprised of fixed keywords, terms defined in the ColdFusion administrator, and terms setup in the definition of the CFC used to create the service.

The REST URL format usually resembles the following:

`http://myserver.com/rest/restServiceMapping/restPath`

OR (using a more realistic example)

`http://localhost/rest/crimeapp/crimes.json`

The parts of the URL broken down as follows:

- **`http://localhost`**
  Obviously the domain of the host. Both HTTP and HTTPS are supported for REST services.
- **`/rest/`**
  This is a (required) reserved word for ColdFusion to indicate that a RESTful service is being used.
- **`/crimeapp/`**
  This is the Service Mapping defined in the ColdFusion Administrator for this service.
- **`crimes.json`**
  This is the “crimes” name in the restPath parameter of the component definition. With the *.json extension, ColdFusion knows to serialize and return the data in JSON format. If you were, for example, to change the file name to crimes.xml, the data would be automatically serialized as XML.

Creating a Basic Component

When creating your CF component, there are some specifics that you need to keep in mind to make it available as a REST service under ColdFusion 10.

There are two **<cfcomponent>** parameters that must be specified in order to enable it as a service.

- **rest = 'true'**
  This will tell ColdFusion to treat this as a REST service when looking for those services to manage upon startup.
- **restPath = '*myrestpath*'**
  Where *myrestpath* is the service path that you must provide in the ColdFusion Administrator for this REST service. This is used to construct the URI to the resource for the web service call.

Within the **<cffunction>** for each method, a **httpmethod** parameter needs to be defined to tell ColdFusion which of the four types of REST operations to perform.

The basic component definition is as follows.

```
<cfcomponent rest="true"
             restpath="person"
             extends="base">

    <cffunction name="GetPeople"
                access="remote"
                returntype="array"
                httpmethod="GET">

          <cfquery name="local.q">
            select * from person
          </cfquery>

          <cfreturn query2array(local.q)>
    </cffunction>

</cfcomponent>
```

The above would be invoked to return the data as string data using the following URL:

http://servername/rest/MyApp/person.json

or

http://servername/rest/MyApp/person.xml

Note: This assumes that the base.cfc file contains the query2array() method described earlier in this unit.

Supporting Additional Methods (Resource Functions)

A more complete component will obviously contain methods for all four of the REST operations. If you do not include an `restPath` parameter for these, ColdFusion considers these Resource Functions.

Defining a GET method to return a single record

The GET method is typically used to retrieve records from a database and return them to the client. As illustrated by the following example, the **`restpath`** attribute is used to define the sequence in which argument values are expected to be encoded in to the calling URL.

```
<cffunction
 name="getPeson"
 access="remote"
 returntype="struct"
 httpMethod="get"
 restpath="{personId}">

  <cfargument name="personId"
              required="true"
              restargsource="Path"
              type="numeric" />

 <cfquery name="local.q">
   select *
   from person
   where personId = <cfqueryparam
                      cfsqltype="cf_sql_numeric"
                      value="#arguments.personId#">
 </cfquery>

 <cfset local.result = query2array(local.q)>

 <cfreturn local.result[1]>

</cffunction>
```

The URL to invoke the method would resemble the following:

`http://servername/rest/MyApp/1.json`

Where the “1” corresponds to the value of the argument **`personId`**.

Defining a POST (create) method

As illustrated by the following method, POST is typically used to persist data into a database. Note that the arguments all have a **restargsource** attribute which indicates that the data is going to be submitted to the service via an HTTP post operation.

```
<cffunction
   name="createPerson"
   access="remote"
   returntype="numeric"
   httpMethod="post">

 <cfargument name="firstname"
             required="true"
             type="string"
             restargsource="Form" />

 <cfargument name="lastname"
             required="true"
             type="string"
             restargsource="Form" />

 <cftransaction>
   <cfquery>
    insert into person (firstname, lastname)
    values (
     <cfqueryparam cfsqltype="cf_sql_varchar"
                   value="#arguments.firstname#">
     <cfqueryparam cfsqltype="cf_sql_varchar"
                   value="#arguments.lastname#">
    )
   </cfquery>
   <cfquery name="local.getlast">
    select LAST_INSERT_ID() as lastid
    from person
   </cfquery>
 </cftransaction>

 <cfreturn local.getlast.lastid>

</cffunction>
```

This method could be invoked through an HTTP form post operation using an html form as illustrated below:

```
   <form action="/rest/MyApp/Person.json" method="post">
     <input type="text" name="firstname"
                        placeholder="First Name″>
     <input type="text" name="lastname"
                        placeholder="Last Name″>
    <input type="submit">
   </form>
```

Defining a PUT method

The PUT method is typically used to update changes to an existing database record. The primary key for the record is typically passed on the URL, while other field-level information is transmitted in an HTTP post transaction.

```
<cffunction
     name="updatePerson" access="remote" returntype="struct"
     httpMethod="post"
     restpath = "{personId}">

    <cfargument
     name="personId"
     required="yes"
     restargsource="Path" />

    <cfargument
     name="firstname"
     required="true"
     type="string"
     restargsource="Form" />

   <cfargument
     name="lastname"
     required="true"
     type="string"
     restargsource="Form" />

   <cfquery>
     update person
     set firstname = <cfqueryparam
                       cfsqltype="cf_sql_varchar"
                       value="#arguments.firstname#">,
         lastname = <cfqueryparam
                       cfsqltype="cf_sql_varchar"
                       value="#arguments.lastname#">

     where personid = <cfqueryparam
                        cfsqltype="cf_sql_numeric"
                        value="#arguments.personid#">

   </cfquery>

   <cfreturn {personId = arguments.personId}>

   </cffunction>
```

This method could be invokes by a form that resembles the following. Note that the “3” corresponds to the primary key for the record:

```
   <form action="/rest/MyApp/person/3" method="post">
     <input type="text" name="firstname" value="Dave" />
     <input type="text" name="lastname" value="Watts"/>
     <input type="submit" value="Click me" />
   </form>
```


Defining a DELETE method

The DELETE method is typically used to remove a record from a database. Much like the PUT method, you will encode the primary key as part of the URL.

```
<cffunction
  name="deleteApplicant"
  access="remote"
  returntype="struct"
  httpMethod="delete"
  restpath="{personId}">

 <cfargument
  name="personid"
  required="true"
  restargsource="Path"
  type="numeric" />

 <cfquery>
  delete
  from person
  where personid = <cfqueryparam
                     cfsqltype="cf_sql_numeric"
                    value="#arguments.personId#">
 </cfquery>

 <cfreturn {
            applicantId = arguments.applicantId
            operation="delete"
           }
  >
</cffunction>
```

Note: The HTTP delete method is only supported in browsers by the XMLHttpRequest object.

Registering your component in CF Administrator

REST components must either be registered with the ColdFusion Administrator or registered programmatically using the restInitApplication() method.

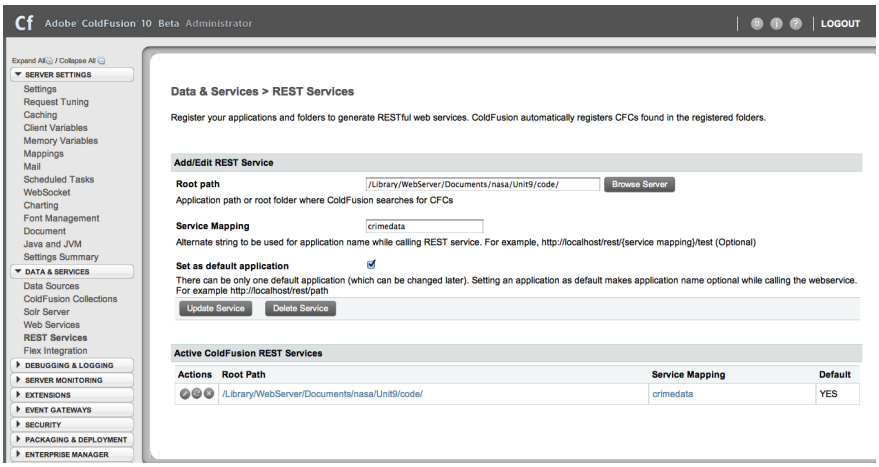


Figure 5: Registering a REST component

Initializing REST Services Programmatically

Whenever a change is made to a CFC registered within the ColdFusion Administrator REST Services, the service must be refreshed (reloaded) into the Administrator. This can be achieved programmatically via a call to the restInitApplication() method. The established best practice is to trigger the restInitApplication() method from inside of your Application.cfc's onRequestStart method as illustrated below. Use the application variable **`this.restsettings.cfcLocation`** in order to tell ColdFusion where your REST components are located.

```
<cfcomponent>
  <cfset this.name = "myapp">
  <cfset this.datasource="myappDSN">

  <cffunction name="onApplicationStart">
    <cfset restInitApplication(
           getDirectoryFromPath(getCurrentTemplatePath()),
           this.name
     )>
  </cffunction>

  <cffunction name="onRequestStart">
    <cfif isdefined("url.init")>
       <cfset onApplicationStart()>
    </cfif>
  </cffunction>
</cfcomponent>
```

Using JQM with REST

You can invoke ColdFusion REST services from JQM using either the $.ajax() method or via a JQM AJAX form post.

Retrieving a Dataset

Retrieving data via AJAX is typically done through the $.ajax() method when the page to display the information is requested from the user. Typically, JavaScript to fetch data from CF and populate a JQM list view will resemble the following:

```
var myApp = {}; // used to cache data results in global ns
var restUrl = "/rest/cfjqm_solution/";

$('#personView').live("pagecreate", function() {
 $.ajax({
   url: restUrl + "person.json",
   type: 'GET',
   dataType: 'json',
   error : function (){
      alert('there was an error');
      console.log(arguments)
   },
   success: function (result) {
      myApp.people = result;

      // delete list items
      $('#personList').empty();

      // loop over list items and add to list
      for (var i=0; i<result.length; i++) {
        var out = "<a href='#'>" + result[i].firstname + " "
                                 + result[i].lastname;
        $('#applicantList')
           .append($("<li></li>")
           .attr("data-value",result[i].personid)
           .html(out) );
      }
    }
 }); // ajax

 // refresh view
 $('#personList').listview('refresh');

 // bind a tap event listener
 $('#personList > li').bind('tap', function(e) {
       var targetValue = this.getAttribute('data-value');
       for (var i=0; i<myApp.people.length; i++) {
         if (myApp.applicants[i].applicantid == targetValue){
              // do something with the data
              // and change the page view - $mobile.changePage
              break;
         }
       }
 });
```

Adding New Data

Typically you will post data to your ColdFusion web service from a JQM form. JQM automatically transmits html form post operations through your browser's XMLHttpRequest object as illustrated below:

```
<div data-role="page" id="ContactForm">
   <div data-role="header"
        data-position="fixed"
         data-id="brandingbar">
    <h1>Add a Contact</h1>
   </div>
   <div data-role="content" id="ContactFormDetail">
    <form
        action="/rest/cfjqm_solution/applicant.json"
        method="post"
        data-ajax="true">

       <fieldset data-role="controlgroup">
        <div data-role="fieldcontain">
           <label for="firstNameField">First Name</label>
           <input type="text"
                  name="firstname"
                  id="firstNameField"
                 data-prevent-focus-zoom="true"
                 data-mini="true" />
        </div>
        <div data-role="fieldcontain">
           <label for="lastNameField">Last Name</label>
           <input type="text"
                  name="lastname"
                  id="lastNameField"
                  data-prevent-focus-zoom="true"
                  data-mini="true" />
         </div>
       </fieldset>

       <fieldset class="ui-grid-a">
        <div class="ui-block-b"
             style="float:none; text-align:center;
                    margin-left: auto; margin-right: auto">

           <button
             type="submit"
             data-theme="a">Submit</button>
        </div>
      </fieldset>
    </form>
  </div>
</div>
```

Updating the Data

Typically you will transmit record updates via an HTTP form post from a JQM form in a manner similar to the scenario described on the prior page. For update operations you will typically need to set the form action URL dynamically to include the primary key value for the record that you want to update. Your code may resemble the following:

```
<script type="text/javascript">

 fillForm = function(rec) {

  var restUrl = '/rest/cfjqm_solution/applicant';

  $('#ContactFormDetail > form')
   .attr('action',restUrl + '/' + rec.personid + '.json');
  }

  // fill rest of form fields
  $('#firstNameField')
       .val(rec.firstname)
       .textinput('refresh');

  $('#lastNameField')
       .val(rec.lastname)
       .textinput('refresh');

 }

</script>

 // code omitted for brevity
 <form
        action="/rest/cfjqm_solution/applicant.json"
        method="post"
        data-ajax="true">


 </form>
 // code omitted for brevity
```

Deleting Data

Use the $.ajax() method to transmit a delete action to your RESTful CFC as illustrated below. Note that the value for the type attribute is “delete” and that the primary key value is transmitted as part of the REST URL:

```
deleteRecord = function(id) {
  var restUrl = "/rest/cfjqm_solution/person/";

  $.ajax({
   url: restUrl + id + '.json',
   type: 'delete',
   dataType: 'json',
   error : function (){
     alert('there was an error');
     console.log(arguments)
   },
   success: function (result) {
     alert("Record Deleted");
   }
 }); // ajax
}
```

Note: You should always secure your REST services using ColdFusion's <cflogin> framework and the <cffunction> ROLES attribute.

Walkthrough 9-2:Using RESTful Services

In this walkthrough, you will create a REST CFC that enables full access to the Applicant table in the cfjqm datasource.

- Create a ColdFusion CFC method that supports CRUD operations against the Applicant table.
- Make a REST AJAX request from JQM to your ColdFusion server to populate a List View
- Make a REST AJAX request from JQM to post data from a form to the database table.

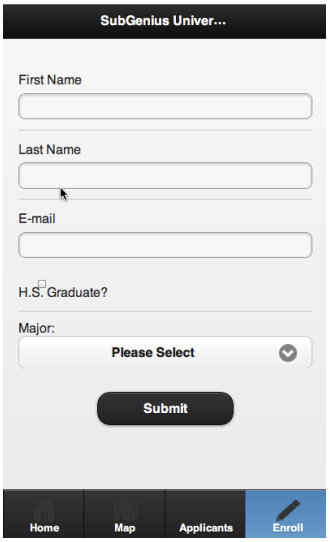


Figure 6: Become a student at SubGenius U!

Steps

Review the Starter Code

1. Open **/walk/walk2/index.html** in your editor and review the code with your instructor.
2. Open **/walk/walk2/data/Application.cfc** in your editor and review the code with your instructor. Note the call to **restInitApplication()**.

Create GET (Read) Methods

3. Open **/walk/walk2/data/applicant.cfc** in your editor and review the code structure. For time sake you have been provided the content of each CRUD method, but you will define each function.
4. Under the comment for *getApplicants*, define a new CFC function with the following parameters:

   - name: getApplicants
   - access: remote
   - returntype: array
   - httpMethod: get

   You code will look similar to the following:

```
<cffunction name="getApplicants"
                    access="remote"
                   returntype="array"
                    httpMethod="get">
```

5. Under the comment define a new CFC function with the following parameters:

   - name: getApplicant
   - access: remote
   - returntype: struct
   - httpMethod: get
   - restPath: {applicantId}

6. Define a `<cfargument>` for this function with the name “applicantId”, and has the following additional parameters:

   - required: true
   - type: numeric
   - restargsource: Path

   Your code will look like the following:

```
<cffunction name="getApplicant"
    access="remote"
    returntype="struct"
    httpMethod="get"
    restpath="{applicantId}"
    >

<cfargument
    name="applicantId"
    required="true"
    restargsource="Path"
    type="numeric" />
```

7. Save the file. You now have methods for returning one applicant, and all applicants.

DELETE Method

8. Under the *deleteApplicant* comment, create a CFC function definition with the following parameters:

   - name: deleteApplicant
   - access: remote
   - returnType: struct
   - httpMethod: delete
   - restpath: {applicantId}

9. Below this, create a `<cfargument>` for this function with the following parameters:
   - name: applicantId
   - required: true
   - restargsource: Path
   - type: numeric

10. Above the closing `</cffunction>` tag for your deleteApplicant method, you need to define data to return. Add the following just above the `</cffunction>` tag.

```
<cfreturn {applicantId = arguments.applicantId}>
```

11. Save the file

POST (Create) Method

12. Just below the *createApplicant* comment, create a CFC function definition with the following parameters:
    - name: createApplicant
    - access: remote
    - returnType: numeric
    - httpMethod: post

13. You now need to create an argument tag for each field you expect to receive from the form POST into the CFC method. The should appear as follows:

```
<cfargument
   name="firstname"
   required="true"
   type="string"
   restargsource="Form"/>

<cfargument
   name="lastname"
   required="true"
   type="string"
   restargsource="Form"/>

<cfargument
   name="email"
   required="true"
   type="string"
   restargsource="Form"/>

<cfargument
   name="hsgrad"
   required="false"
   type="boolean"
   default="0"
   restargsource="Form"/>

<cfargument
   name="hsgpa"
   required="true"
   type="numeric"
   restargsource="Form"/>

<cfargument
   name="major"
   required="true"
   type="string"
   restargsource="Form"/>
```

14. Save file

PUT (Update) Method

15. Just below the *updateApplicant* comment, create a CFC function definition with the following parameters:
    - name: updateApplicant
    - access: remote
    - returntype: struct
    - httpmethod: post
    - restpath: {applicantId}

16. Below the CFC function definition, define the fields that you expect to receive from the form post. Add in the argument tags as shown below. This is very similar to the POST arguments, so make note of the differences.

```
<cfargument
   name="applicantId"
   required="yes"
   restargsource="Path" />

<cfargument
   name="firstname"
   required="true"
   type="string"
    restargsource="Form"/>

<cfargument
   name="lastname"
   required="true"
   type="string"
   restargsource="Form"/>

<cfargument
   name="email"
   required="true"
   type="string"
   restargsource="Form"/>

<cfargument
   name="hsgrad"
   required="true"
   type="numeric"
   restargsource="Form"/>

<cfargument
   name="hsgpa"
   required="true"
   type="numeric"
   restargsource="Form"/>

<cfargument
   name="major"
   required="true"
   type="string"
   restargsource="Form"/>
```

17. Just above the closing `</cffunction>` tag, we need to return data. Add the following line to the function:

```
<cfreturn {applicantid = arguments.applicantid}>
```

18. Save the file

Register the REST Component

19. Open the ColdFusion Administrator and access the REST Services section (under Data & Services). You should see a screen similar to the following:

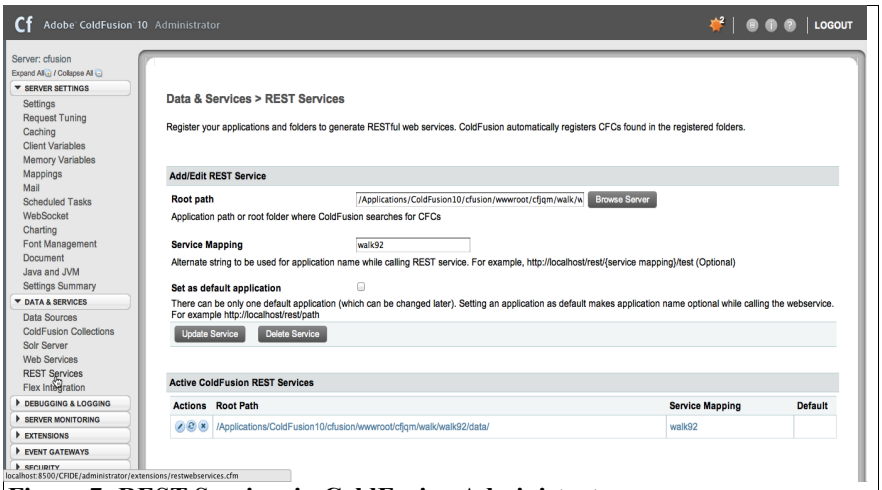


Figure 7: REST Services in ColdFusion Administrator

20. In Root Path field, click the Browse Server button. Using the pop up dialog, navigate to and select the **[webroot]/cfjqm/Unit9/walk/walk2/data** directory and press "ok".
21. In the Service Mapping field, enter **walk9_2**
22. Click the **Update Service** button
23. You should see a message at the top of the form that says "Server has been updated successfully."

Populate a Listview from $.ajax() Request

You will now move on to populating the "Students" tab of your SubGenius University mobile application. We will leverage the HTML structure of the page and add the JQM code to add data to it.

24. Open **/walk/walk2/index.html** in your editor.
25. Where indicated by the comment, define a "pagecreate" event listener tied to the applicationView ID. Type the following below the comment line to start the function:

```
$('#applicantView').live("pagecreate", function() {
```

26. After the curly brace, add a $.ajax() call to retrieve the data and build out the error and success functions.

```
$.ajax({
    url: restUrl + "applicant.json",
    type: 'GET',
    dataType: 'json',
```

27. After the dataType line, add a line to handle the Error condition if triggered:

```
error : function (){ alert('there was an error');
console.log(arguments)},
```

28. Next add a function to handle the Success condition:

```
success: function (result) {
   myApp.applicants = result;

    // delete list items
     $('#applicantList').empty();

    // loop over list items and add to list
    for (var i=0; i<result.length; i++) {
        var out = "<a href='#'>" + result[i].firstname + "
" + result[i].lastname
        $('#applicantList')
           .append($("<li></li>")
           .attr("data-value",result[i].applicantid)
           .html(out) );
}
```

This will clear the existing list, and repopulate it my looping over the results returned from the AJAX call to your CFC that was just created.

29. We next need to force the list to refresh within the DOM so it is visible to the user. Add the following below the Success function above.

```
// refresh view
$('#applicantList').listview('refresh');
```

30. Next, we complete the Success condition and close up the function by rebinding tap event listeners to the individual list items we have just added. This will allow us to add a simple detail page to display.

```
// bind a tap event listener
$('#applicantList > li').bind('tap', function(e) {
   var targetValue = this.getAttribute('data-value');
    for (var i=0; i<myApp.applicants.length; i++) {
       if (myApp.applicants[i].applicantid == targetValue)
           {
           // $("#alertbox").dialog('open');
           setApplicantDialog(myApp.applicants[i]);
           $.mobile.changePage("#applicantDetail");
           break;
           }
     }

    });
    }
});
});
```

Add List Detail

31. Now we can add the list detail. This will setup the detail page that is displayed when a user taps the applicant name from the list. ABOVE the STEP 31 comment line, add the following function:

```
setApplicantDialog = function(objConfig) {
   $("#applicantName").text(objConfig.firstname +
     " " + objConfig.lastname);

   var tpl = "<label>Name:</label>" +
               "${firstname} ${lastname}<br />";
    tpl += "<label>Major:</label>" + "${major}";

   $.template( "applicantTemplate", tpl );

   $( "#applicantDetailContent").empty();
   $.tmpl( "applicantTemplate",
objConfig).appendTo( "#applicantDetailContent" );
}
```

32. Save the file.
33. Open **/walk/walk2/index.html** and test the file with your instructor. Clicking the "Students" nav button should show a set of names in a JQM list view. Tapping a name should take you to a simple detail page.

Add Enrollment Form

34. Return to your editor.
35. Where indicated by the comment, add in the HTML/JQM form to POST data to our REST service. Start the form by adding the following:

```
<form     action="/rest/walk9_2/applicant.json"
          method="post"
          data-ajax="true">

<fieldset data-role="controlgroup">

<div data-role="fieldcontain">

   <label for="firstNameField">First Name</label>

   <input type="text"
     name="firstname"
     id="firstNameField"
     data-prevent-focus-zoom="true"
     data-mini="true" />

</div>
```

36. Repeat the same pattern for the Last Name and Email fields, and close up the fieldset.

```
<div data-role="fieldcontain">
 <label for="lastNameField">Last Name</label>
 <input type="text"
       name="lastname"
        id="lastNameField"
       data-prevent-focus-zoom="true"
       data-mini="true" />
</div>

<div data-role="fieldcontain">
 <label for="emailField">E-mail</label>
 <input type="email" name="email" id="emailField"
        data-prevent-focus-zoom="true" data-mini="true"/>
</div>

</fieldset>
```

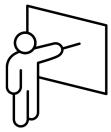
37. Create a new field set for the HSGradField and MajorField, then close up the fieldset and form.

```
   <fieldset data-role="controlgroup">
    <div data-role="fieldcontain">
     <input name="hsgrad" id="hsgradfield" type="checkbox">
     <label for="hsgradfield">
       High School Graduate
     </label>
    </div>
    <div data-role="fieldcontain" id="hsgpafieldcontainer">
      <label for="hsgpa">H.S. GPA</label>
      <input type="number" name="hsgpa" id="hsgpafield"
                 step="0.1" min="0" value="0.0" max="4.3" />
    </div>
    <div data-role="fieldcontain">
       <label for="major" class="select">Your Proposed
   Major:</label>
              <select name="major" id="majorField">
               <option value="">Please Select</option>
               <option
   value="Basketweaving">Basketweaving</option>
               <option value="Dodgeball Arts">Dodgeball
   Arts</option>
               <option value="Curling">Curling</option>
               <option
   value="Ghostbusting">Ghostbusting</option>
             </select>
            </div>
        </fieldset>
          <fieldset class="ui-grid-a">
           <div class="ui-block-b" style="float:none; text-
   align:center; margin-left: auto; margin-right: auto">

   <button type="submit" data-theme="a">Submit</button>

   </div>
   </fieldset>
</form>
```

38. Save the file.
39. To test your page, reload the index.html file, and click the “Enroll” nab button. Fill out the form, making sure to click the “High School Graduate” checkbox to expose the H.S. GPA field. After filling out the fields, you can click Submit. After successfully submitting, you can click the “Students” navigation button and the new name should appear in the list.

– End of Walkthrough --

Implementing Push with Web Sockets and ColdFusion 10



ColdFusion 10 fully supports HTML5 web sockets. Web sockets are an exciting new extension of the HTML standards that provide persistent connections between the browser and server, enabling real-time one-to-one and one-to-many push-based communication channels.

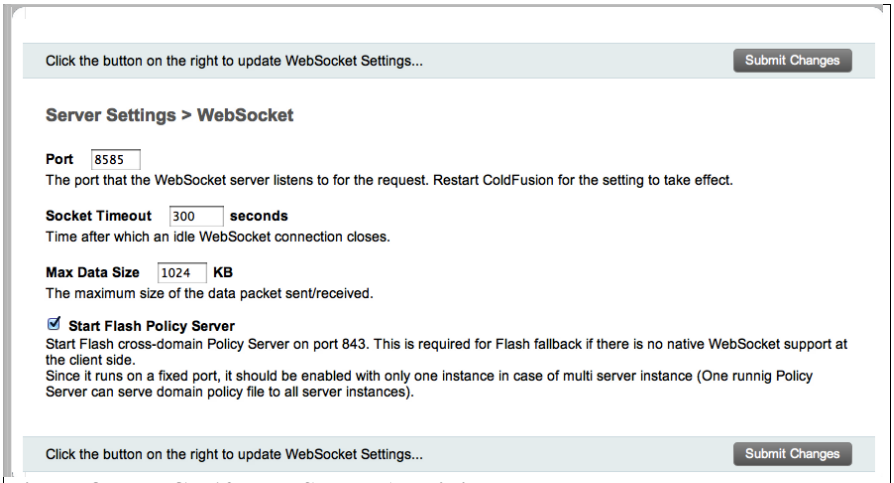


Figure 8: The CF 10 Web Socket Administrator

Note: ColdFusion HTML5 WebSockets are supported by both iOS and Android native browsers.

Understanding HTML5 Web Socket Features

HTML5 Web Sockets provides a full-duplex, bi-directional, single socket TCP channel for continuous communication between the client and server. This persistent connection provides several advantages:

- **Realtime connection**
  The data can be sent/received in realtime, without the lag time associated with traditional request/response polling inherent in HTTP protocol handling.
- **Single port**
  Send/receive continuously over a single assigned port
- **Lower network overhead**
  As only the data is being transmitted, there is not the overhead of full browser refreshes or large data packets.
- **Persistent connection**
  There is faster performance as there is no need to establish a new connection and then close it with each new data request.

Reviewing WebSocket Modes Supported by ColdFusion

ColdFusion 10 provides two modes of operation for web sockets.

- **Point to Point mode**

  In Point to Point (P2P) mode there is a single client and single server involved in the communications process. There is no need to deal with complex multi-client subscription models in this process.

- **Broadcast Mode**

  Broadcast mode follows three models, each intended to handle the one-to-many communication of messages/data. In each of these models there is a notion of a **subscriber** and **publisher**.

  A **publisher** is a source of content that a client will sign up to receive updates from. Data is sent from a publisher is transmitted to all "listening" subscribers

  A **subscriber** is a client that is signed up to be connected to the data stream of the publisher so they can receive the updates from that publisher.

  These two entities can be configured into the following models:

  - **Server relay**
    In this model the server will simply receive a message from a client, and relay the message to all listening (subscribing) clients as-is without any modification.

  - **Server process and relay**
    In this case a client will send a message to the publishing server first. The server will then perform some operation on the message before the message gets sent out to other subscribers. This is the most common model used, and can be implemented for everything from authentication to scanning content for objectionable content. Once validated, the content is then sent to all subscribing clients.

  - **Server push**
    In this situation, the publishing server itself initiates the message send to all subscribing clients. This most popular use case for this is the monitoring dashboard or alerting system.

Configuring the Web Socket Service

The ColdFusion Administrator provides some basic settings for configuring the use of Web Sockets.

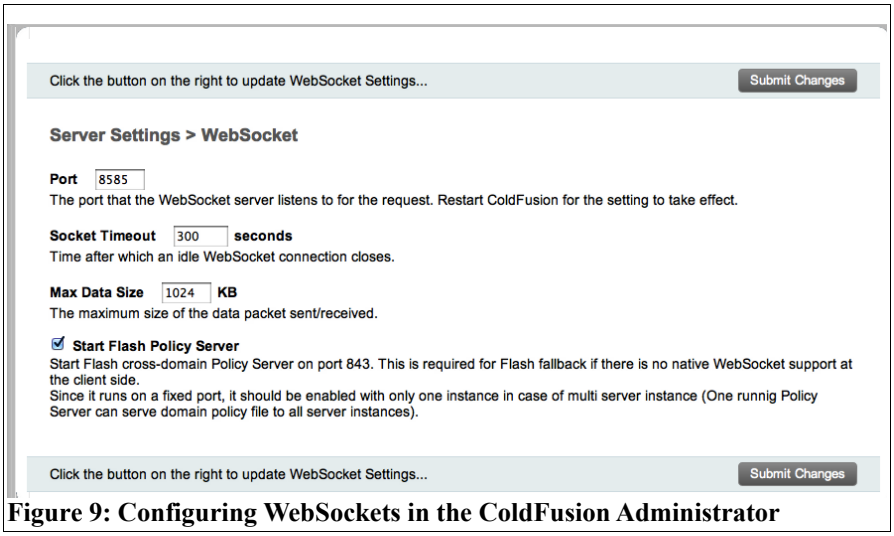


Figure 9: Configuring WebSockets in the ColdFusion Administrator

- **Port**
  This is the default port over which a web socket connection will be established if not defined upon creation.
- **Socket Timeout**
  The timeout for the connection if no activity is detected. This value will depend on the types of applications running on the server.
- **Max Data Size**
  This is the default size of the data packet sent over the persistent connection. Although adjustable, the default value should be adequate for most applications.
- **Start Flash Policy Server**
  Only needed if you are going to attempt support for client that do not support native HTML5 web socket services.

Note: A ColdFusion server restart is required for settings to be applied.

Using the Flash Policy Server

In order to gracefully handle situations where the browser does not have full HTML5 support, ColdFusion adds a safety net of sorts. It can drop down to providing communications via Flash rather than websockets if needed. Here is a breakdown of the fall back plan:

Situation	Actions
HTML5 native websocket support unavailable, but Flash IS installed	System falls back to Flash for communications.
HTML5 native websocket support unavailable, but Flash IS NOT installed	Sends a message indicating that the connection is not successful. To proceed, either move on to a compliant browser or install Flash.

In the event Flash is responsible for communications, please note the restrictions on this functionality in a multi-server environment as noted in the ColdFusion documentation.

Using Broadcast Mode

Broadcast mode is the predominant use for this protocol. In order to get started, you must first define websocket "channels."

Channels are defined in the Application.cfc file by configuring the "this.wschannels" variable as illustrated below:

```
<cfcomponent>
  <cfset this.name = "cfjqm_solution">
  <cfset this.datasource="cfjqm">
  <cfset this.wschannels = [
        {
         name : "chat",
         cfcListener : "ChatListener"
        }
  ]>
  ...
</cfcomponent>
```

The preceding example defines a single web socket broadcast channel named "chat." "Subchannels" can be created at runtime by adding to the structure, e.g. `chat.tech` or `chat.coldfusion`. The initial creation of the channel, however, needs to occur in the Application.cfc file.

Using the <cfwebsocket> tag

ColdFusion 10 abstracts the functionality of websockets into the `<cfwebsocket>` tag.

```
<cfwebsocket
  name="websocketName"
  onMessage="JavaScript function name"
  onOpen="JavaScript function name"
  onClose="JavaScript function name"
  onError="JavaScript function name"
  useCfAuth=true|false
  subscribeTo="channel_list">
```

Attribute	Description
`name`	(string, required) This is the name of the websocket object that will be created and used by all the local JavaScript functions.
`onMessage`	(string, required) The name of the JavaScript function that gets called when the websocket receives a messages from a channel it has subscribed to. There are two parameters passed into the called JavaScript function: • aEvent: The WebSocket event that is dispatched from the server. • aToken: Used to get data received from the server, e.g. aToken.data will retrieve the actual message.
`onOpen`	(string) The name of the JavaScript function that gets called when the websocket connection is initially opened.
`onClose`	(string) The name of the JavaScript function that gets called when the websocket connection is closed.
`onError`	(string) The name of the JavaScript function that gets called when the connection error is encountered.
`useCfAuth`	(boolean) default = **true**. If **true**, and the user has already logged into the names application, they do not need to authenticate to use the websocket. If **false**, the user will need to authenticate in order to use the web socket.

Attribute	Description
subscribeTo	(list) This is a comma-separated list of the channel names that you wish to use with this websocket object. The names in the list must be in the list defined in the Application.cfc file.

Assuming the Application.cfc we established earlier, a simple example to setup a subscriber to display an alert on a web page whenever a message is received on the "weather" channel, presumably for a weather alert, would resemble the following:

```
<cfwebsocket
    name = "weatherSocket"
    onOpen = "startWeatherAlerts"
    onMessage = "showWeatherAlert"
    onClose = "closeWeatherAlerts"
    subscribeTo = "weather" />

<script>
var startWeatherAlerts = function(){
   alert('Starting weather monitoring');
}

var closeWeatherAlerts = function(){
   alert('Ending weather monitoring');
}

// 'alert' is object passed in from from recevied message
var showWeatherAlert = function(alert){
   messageTxt = alert.data;
   if((messageTxt != '') && (alert.type == 'data')){
       alert('Weather alert: ' + messageTxt);
   }
}

</script>
```

Handling Message Responses

When developing applications that publish/receive messages, make extra use of the **`console.log()`** functionality of the browser to inspect the received responses in debugger.

When publishing a message on an established channel, you will get an initial "welcome" message when publish your first message. This is not the same as the **`onOpen`** event from the **`<cfwebsocket>`** tag.

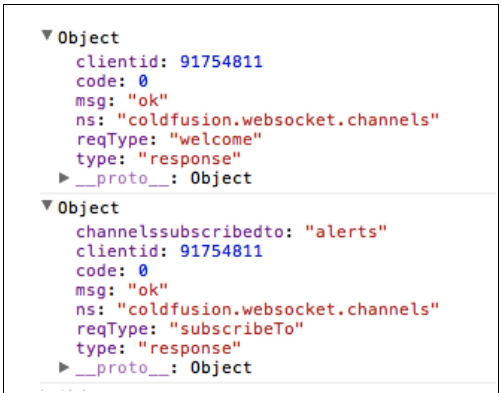


Figure 10: Recieved websocket messages

Further, as you are communicating over a single channel, you will receive an acknowledgment message from the server that your published message was received.

These messages are both received over the same channel that your application is "listening" on, so your code will need to filter out these utility messages from the true data messages you want to act upon.

All messages will trigger the **`onMessage`** event handler, so that code typically determines if the message received needs to be processed or just noted. For an example like the the one above, after the first message, when a message is published you will receive two responses. One is the publishing acknowledgment and the other is the message receipt, i.e. the message you were actually listening for as a client.

Figure 11: Data received from onMessage

Integrating CF 10 WebSockets with jQuery Mobile

Since the starting point for jQuery Mobile apps is an index.html file, you will not be able to use the <cfwebsocket> tag to do the “heavy lifting” of your code generation. You will need to hardcode the scripts and settings that would normally be output by ColdFusion.

Manually Loading ColdFusion's JavaScript WebSockets API

Your first step is to include the following code into the <head> section of your JQM application's index.html file. This content defines the url, websocket, and flash policy server ports that are set in the ColdFusion administrator:

```
<script type="text/javascript">
_cf_loadingtexthtml="<img alt=' '
src='/CFIDE/scripts/ajax/resources/cf/images/loading.gif'/>";

 _cf_contextpath="";

_cf_ajaxscriptsrc="http://localhost:8500/CFIDE/scripts/ajax";
_cf_jsonprefix='//';
_cf_websocket_port=8575;
_cf_flash_policy_port=1243;
</script>
```

Next, you will need to add <script> tags to load ColdFusion's undocumented websocket Javascript API from the following URLs:

```
http://localhost:8500/CFIDE/scripts/ajax/messages/cfmessage.js
http://localhost:8500/CFIDE/scripts/ajax/package/cfajax.js
http://localhost:8500/CFIDE/scripts/ajax/package/cfwebsocketCore.js
http://localhost:8500/CFIDE/scripts/ajax/package/cfwebsocketChannel.js
```

Defining your WebSocket Event Listeners

Your next step is to define a JavaScript object that contains all of your WebSocket event listeners, as well as a function that initializes the connection between the mobile browser and the ColdFusion 10 Websocket server as illustrated below:

```
var webSocket = {
 startChat : function() {
  alert('Starting chat');
 },

 closeChat : function(){
  alert('Ending Chat');
 },

 //'alert' is object passed in from from recevied message
 showChatMessage : function(alert){
  msgTxt = alert.data;
  if((msgTxt != '') && (alert.type == 'data')){
     var out = '<div class="msg">' + msgTxt + '</div>'
     $('#chatContent').append(out); // write to page
  }
 },

 // transmit msg to websocket server
 sendMessage : function(msg) {
   webSocket.chatter.publish("chat",msg)
 },

 // initialize!
 init : function() {
  var handle="chatter";
  var channel = "chat";
  var appName = 'cfjqm_solution';

  // generate pseudo-uuid for client identifier
  var uid = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'.replace(/[xy]/g,
  function(c) {
  var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
  return v.toString(16);
  });

  // initialize communications and event listeners
  webSocket.chatter = ColdFusion.WebSocket.init(
                    handle,
                    appName,
                    uid,
                    channel,
                    webSocket.showChatMessage,
                    webSocket.startChat,
                    webSocket.closeChat,
                    null,
                    '/cfjqm/walksolution/final/index.html'
              );
      }
 }
```

Initializing the WebSocket from a jQuery Page

The following markup presents the user with a simple chat UI. At the top of the page is a text field for typing in their message and a submit button for confirming its transmission. The content area of the page is reserved for displaying incoming messages from the WebSocket server.

```
<div data-role="page" id="chat">
 <div data-role="header" data-position="fixed"
                        data-id="brandingbar">
  <h1>SubGenius University</h1>
  <div data-role="header">
   <div class="ui-body ui-body-a">
    <div data-role="fieldcontain">
      <label for="basic" class="ui-hidden-accessible">
       Text Input:
      </label>
      <input type="text"
             id="chatInput"
             data-mini="true"
             data-mini="true"
             data-prevent-focus-zoom="true"
             placeholder="Your Comments"/>

       <button data-mini="true"
               id="btnChatSubmit"
               data-theme="a">Submit</button>
    </div>
   </div>
  </div>
  </div>
  <div data-role="content" id="chatContent"></div>
  <div data-role="footer"
       data-id="nav"
       data-position="fixed"
       class="nav-glyphish-example">
     <!-- content omitted for brevity -->
  </div>
</div>
```

You can attach an event listener to the submit button and initialize the WebSocket connection on page load as illustrated by the following:

```
$('#chat').live("pagecreate", function() {

 // handle button click
 $('#btnChatSubmit').bind('click', function(e) {
   webSocket.sendMessage($('#chatInput').val());
   $('#chatInput').val('');
 });
 // initialize web socket
 webSocket.init();
});
```

Walkthrough 9-3: Implementing Real-Time Chat

In this walkthrough, you will develop the realtime chat feature of the SubGenius University application.

- Define a CF websocket channel.
- Define and initialize a webSocket connection between your JQM app and ColdFusion 10.
- Send and Receive data from a WebSocket.

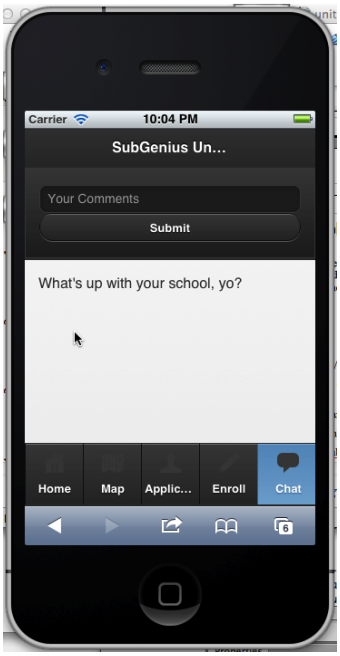


Figure 12: Real Time Chat with ColdFusion 10 WebSockets

Steps

Review the Starter Code

1. Open **/walk/walk3/index.html** in your editor and review the code with your instructor.

Modify Application.cfc

2. Open **/walk/walk3/data/Application.cfc**
3. Add the following line to the group of <cfset> tags at the top of the file to define the web socket channels.

```
<cfset this.wschannels = [
   {name : "chat",
    cfcListener : "ChatListener"
   }
]>
```

Create the ChatListener CFC

4. Open the file **/walk/walk3/data/ChatListener.cfc** in your editor.
5. Define a new component that extends **CFIDE.websocket.ChannelListener**
6. Create a new function named "**beforePublish**" with access of "**public**".
7. Add two arguments:
   - Name: message, type: any
   - Name: publisherInfo, type: struct

8. Add the following code to the body of the function.

```
<cfset local.time = DateFormat(now(),"long")>
<cfset local.message  = local.time & ": <b>" & message &
"</b>">

<cfreturn local.message>
```

9. The completed CFC should look like the following:

```
<cfcomponent extends="CFIDE.websocket.ChannelListener">
 <cffunction name="beforePublish" access="public">
   <cfargument name="message" type="any">
   <cfargument name="publisherInfo" type="struct">

    <cfset local.time = DateFormat(now(),"long")>
   <cfset local.message  = local.time & ": <b>" & message
& "</b>">

   <cfreturn local.message>
  </cffunction>
</cfcomponent>
```

10. Save the file.

Add CF 10 Web Socket Components

11. Open /walk3/index.html in your editor.
12. Where indicated by the comment, add references to the ColdFusion 10 Web Socket components as shown below:

```
<script type="text/javascript">
_cf_loadingtexthtml="<img alt=' '
src='/CFIDE/scripts/ajax/resources/cf/images/loading.gif'/
>";
_cf_contextpath="";
_cf_ajaxscriptsrc="http://localhost:8500/CFIDE/scripts/aja
x";
_cf_jsonprefix='//';
_cf_websocket_port=8575;
_cf_flash_policy_port=1243;
</script>
```

13. Add the script tags to load the CF 10 AJAX libraries:

```
<script type="text/javascript"
  src="/CFIDE/scripts/ajax/messages/cfmessage.js">
</script>
<script type="text/javascript"
   src="/CFIDE/scripts/ajax/package/cfajax.js"></script>
<script type="text/javascript"
   src="/CFIDE/scripts/ajax/package/cfwebsocketCore.js">
</script>
<script type="text/javascript"
   src="/CFIDE/scripts/ajax/package/cfwebsocketChannel.js">
</script>
```

14. After the code that you inserted in the previous step, create a script block to define the web socket connection and event handler functions that you will use later in the code.

```
<script type="text/javascript">
var webSocket = {
    startChat : function() {
           alert('Starting chat');
    },

    closeChat : function(){
           alert('Ending Chat');
    },

    // 'alert' is object passed in from recevied message
    showChatMessage : function(alert){
      messageTxt = alert.data;
      if((messageTxt != '') && (alert.type == 'data')){
        $('#chatContent').append(
          '<div class="message">' + messageTxt + '</div>'
        );
      }
},
```

15. Next you can add the sendMessage function.

```
sendMessage : function(msg) {
  webSocket.chatter.publish("chat",msg)
},
```

16. Finally the initialization of the web socket itself.

```
init : function() {

    var handle="chatter";
    var channel = "chat";
    var appName = 'walk92';
    var uid = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'.replace(/
[xy]/g, function(c) {
     var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|
0x8);
     return v.toString(16);
    });


    webSocket.chatter = ColdFusion.WebSocket.init(
           handle,
           appName,
           uid,
           channel,
           webSocket.showChatMessage,
           webSocket.startChat,
           webSocket.closeChat,
           null,
           '/cfjqm/walk/walk3/index.html'
    );
    }
  }
 </script>
```

17. Save the file.

Initialize Chat Form

18. Where indicated by the comment, add the following function to bind the sendMessage event to the form button client.

```
/* chat init */
$('#chat').live("pagecreate", function() {
   $('#btnChatSubmit').bind('click', function(e) {
     webSocket.sendMessage($('#chatInput').val());
     $('#chatInput').val('');
   });
   webSocket.init();
});
```

Add the Chat Form

19. Where indicated by the comment, add the following form code. This JQM form includes the DOM IDs that tie the form to the JavaScript code we added in previous steps.

```
<div data-role="fieldcontain">
  <label for="basic" class="ui-hidden-accessible">
    Text Input:
  </label>

  <input data-mini="true"
         type="text"
         name="name"
         id="chatInput"
         value=""
         data-mini="true"
         data-prevent-focus-zoom="true"
         placeholder="Your Comments"/>

  <button
         data-mini="true"
         id="btnChatSubmit"
         data-theme="a">Submit</button>

</div>
```

20. Save the file.

Test the application

21. Test the application by opening the index.html file in two browser windows.
22. In each window click on the “Chat” nav button.
23. Enter a message and click the Submit button. It will appear in both chat content windows.

– End of Walkthrough --

Unit Review

1. You can only refresh REST services via the ColdFusion Administrator. (True/False)?

2. What are the four HTTP methods supported by ColdFusion for REST services.

3. You must serialize/deserialize XML/JSON data for web services in ColdFusion 10. (True/False)?

4. ColdFusion 10 allows for nested REST services. (True/False)?

5. ColdFusion 10 supports HTML5 Web Socket communications between clients and servers. (true/false)

6. Describe how you would create and use a subresource in the context of REST(ful) services.