Creating a Web Service in ColdFusion MX

by Stacy Young

April 2002

Macromedia, Inc.
600 Townsend Street
San Francisco, CA 94103
415–252–2000

Contents

Maybe you've heard that web services will revolutionize the way you do business and develop applications on the web. You've heard they'll simplify corporate infrastructures and unite business partners all over the world and even make your coffee in the morning. Web services promise a lot, but is it fact or fiction?

The Next Big Thing

As a developer, I'm constantly presented with the supposed "next big thing." After enough disappointments, I stopped paying attention. When I first heard of web services I thought they were no exception. That was well over a year ago. Since that time, the momentum has only gotten stronger. This prompted me to stop and take a closer look. Happily, the release of Macromedia ColdFusion MX means getting your hands dirty with web services couldn't be easier.

As a developer at SureFire Commerce, I put together a simple example of how to process payments though a web service. The web service comes from SureFire Commerce, which enables any business to process electronic payments, such as charging a customer's credit card, e-mailing an invoice that can be paid online, or depositing money into the bank account of a supplier or employee. Included with its payment processing applications, SureFire offers merchant accounts, risk management, financial reconciliation and online reporting.

At SureFire, one of the biggest issues that I encounter as a developer is customer integration. As a payment processor serving customers who have specific and individual needs, it is cost-prohibitive for me to develop customized solutions for every platform, not to mention having to re-deploy all those solutions every time I want to make an enhancement. Additionally, traditional Internet communication using HTTP POST/GET is not fully compatible when implementing more complex service offerings. If you've ever had to use CFHTTP to parse data from another web page, this problem is clear to you.

Enter web services, a communication platform for applications. By using a combination of standardized protocols, it allows applications to communicate with each other regardless of their platform or programming language. By offering payment processing as a web service, I only need to define a single standard interface for everyone.

In this article, I'll explain how to create a web service using ColdFusion MX. Then I'll explore how you can invoke the new service over the Internet to process a credit card transaction.

Building the Back End

The first thing you need to do is learn how to construct a credit card authorization service and encapsulate that in a ColdFusion Component (CFC). Typically, a credit card transaction requires certain parameters to be processed, such as customer information, card number and expiration date. This CFC verifies that the correct parameters were supplied, queries your database for the available credit for the card requested and compares the requested authorization amount and the available credit. Lastly, it decides whether to authorize or decline the transaction, and returns the appropriate response.

The following is an example ColdFusion Component that performs a basic credit card authorization:

```
<cfcomponent>
   <cffunction access="remote" name="authorize" output="false"
    returntype="struct">

<!--- All parameters are mandatory! --->
<cfargument name="name"     type="string"       required="true">
<cfargument name="address"  type="string"       required="true">
<cfargument name="zip"      type="string"       required="true">
<cfargument name="state"    type="string"       required="true">
<cfargument name="country"  type="string"       required="true">
<cfargument name="cardtype"        type="string"
      required="true">
<cfargument name="cardnumber" type="string"     required="true">
<cfargument name="expiry"   type="string"       required="true">
<cfargument name="amount"   type="numeric" required="true">

<!--- Query database to find the available balance for the card --->
<cfinclude template="card.cfm">

<!--- Check for adequate credit, set transaction status accordingly -
    -->
<cfif card.availableCredit GTE arguments.amount>
   <cfinclude template="updateBalance.cfm">
   <cfset response.status  = "AUTHORIZED">
   <cfset response.message = "Your transaction was completed
    successfully, thank you.">
<cfelse>
   <cfset response.status  = "DECLINED">
   <cfset response.message = "We're sorry, your transaction failed,
    insufficient funds.">
</cfif>

<cfinclude template="insertTransaction.cfm">
<cfset temp = structAppend( response , arguments )>
<cfset response.reference = insertTransaction.sequence>

<cfreturn response>
   </cffunction>

</cfcomponent>
```

Dissecting the Component

The `cfcomponent` tag is a wrapper that defines the functions within it. Typically a component packages a set of related functions or "methods." For instance, a component dedicated to user management would likely contain functions that could perform the following:

- add a user
- update a user
- delete a user
- list users

You define each method with the `cffunction` tag. This article focuses on the "authorize" function, or method. The code for the "authorize" function is as follows:

```
<cffunction access="remote" name="authorize" output="false"
   returntype="struct">
```

To create a function that is a web service, specify the "access" value as "remote." This determines the level of exposure for the function. Specifying "`access=remote`" means that you can access the function "from abroad," whereas "`access=private`" means that you can only access the function from within the component.

Next, consider what type of response you'll send back to the calling application (such as string, numeric value, structure, array, query object and so forth). The `returntype` attribute specifies the data type of the response, in this case, you'll return a CFML **structure** (also called a **struct**). If you're unsure of the data type, perhaps due to conditional logic within your function, you can always specify a `returntype` of "any" and ColdFusion will adjust accordingly. However, for maximum compatibility with other web service clients, specifying a data type for the `returntype` attribute ensures a safer transaction.

```
<cfargument name="name" type="string" required="true" default="">
```

Use the `cfargument` tag to specify the required parameters for the function. The `default` attribute is optional, while the `type` and `required` attributes are mandatory.

To reference the arguments within your function, access them through the `arguments` scope (for example, `#arguments.myVariable#`):

```
<cfquery name="card" datasource="main" dbtype="ODBC">
 SELECT availableCredit
 FROM credit_cards
 WHERE cardnumber = #arguments.cardnumber#
</cfquery>
```

Our database query expects the `cardnumber` argument and returns a single field called `availableCredit`. (To keep the code simple, this example includes the database query with the `cfinclude` tag rather than putting all of the code in the CFC.)

```
<cfif card.availableCredit GTE arguments.amount>
```

Compare the amount of the purchase to the available credit and determine the transaction status: AUTHORIZED or DECLINED.

```
<cfinclude template = "insertTransaction.cfm">
<cfset temp = structAppend( response , arguments )>
<cfset response.reference = insertTransaction.sequence>
```

Next, insert a record of the transaction into your database. If the status is AUTHORIZED, make an adjustment on the available credit for the requesting cardholder to indicate that the funds have been spent. In either case, the database insert returns a reference number to pass back to the calling application for reconciliation:

```
<cfreturn response>
```

Specifying a `returntype` of "`struct`" enables you to return the response structure back to the calling application that carries all pertinent information. Here, use the `cfreturn` tag to return the entire structure "response" back to the calling application.

ColdFusion Component Browser

Upon typing the URL for your component, ColdFusion MX forwards you to a built-in server utility called the ColdFusion Component Browser, which auto-documents your components and benefits development teams.

Figure 1 shows the documentation for the *authorize* component built above.

sample.authorize

Component authorize

hierarchy:	WEB-INF.cftags.component sample.authorize
path:	C:\CFusionMX\wwwroot\sample\authorize.cfc
properties:	
methods:	authorize

* - private method

authorize

```
remote struct authorize ( required string name, required string address,
required string zip, required string state, required string country, required
string cardnumber, required date expiry, required numeric amount )
```

Output: supressed
Parameters:
name: string, required, name
address: string, required, address
zip: string, required, zip
state: string, required, state
country: string, required, country
cardnumber: string, required, cardnumber
expiry: date, required, expiry
amount: numeric, required, amount

Figure 1: *ColdFusion Component Browser*

Creating a WSDL File

When publishing a web service, make a description of it available to potential clients. This description enables your clients to interact with your web service and informs them of required arguments and data types they must pass to the web service. This description is called **Web Services Description Language**, or **WSDL**.

Once you save the component in your webroot, you can view the WSDL file (which ColdFusion creates for you) by appending "`?WSDL`" to the end of your component URL, as in the following example:

```
http://www.mydomain.com/mycomponent.cfc?WSDL
```

Browsing the URL above would produce the WSDL file shown in Figure 2.

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://sample"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:intf="http://sample"
    xmlns:impl="http://sample-impl" xmlns:tns2="http://xml.apache.org/xml-soap"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  - <types>
    - <schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xml.apache.org/xml-soap">
      - <complexType name="Map">
        - <sequence>
          - <element name="item" minOccurs="0" maxOccurs="unbounded">
            - <complexType>
              - <all>
                  <element name="key" type="xsd:anyType" />
                  <element name="value" type="xsd:anyType" />
                </all>
              </complexType>
            </element>
          </sequence>
        </complexType>
        <element name="Map" nillable="true" type="tns2:Map" />
      </schema>
    </types>
    <wsdl:message name="CFCInvocationException" />
  + <wsdl:message name="authorizeRequest">
  + <wsdl:message name="authorizeResponse">
  + <wsdl:portType name="authorize">
  + <wsdl:binding name="authorize.cfcSoapBinding" type="intf:authorize">
  + <wsdl:service name="authorizeService">
  </wsdl:definitions>
```

Figure 2: *Produced WSDL file*

Publishing a Web Service

To publish a web service with ColdFusion, make an entry in the Web Services control panel in the ColdFusion administrator (see Figure 3).

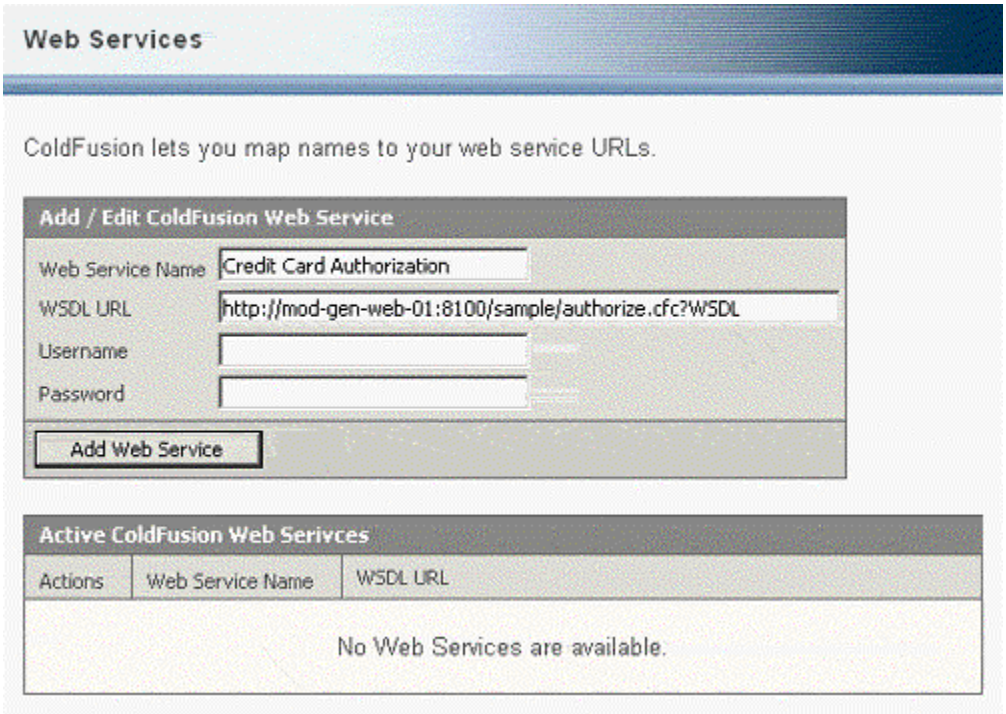

Figure 3: *ColdFusion Web Services entry form*

Enter a name and the URL of your web service (see Figure 4). Then click Add Web Service to activate it.

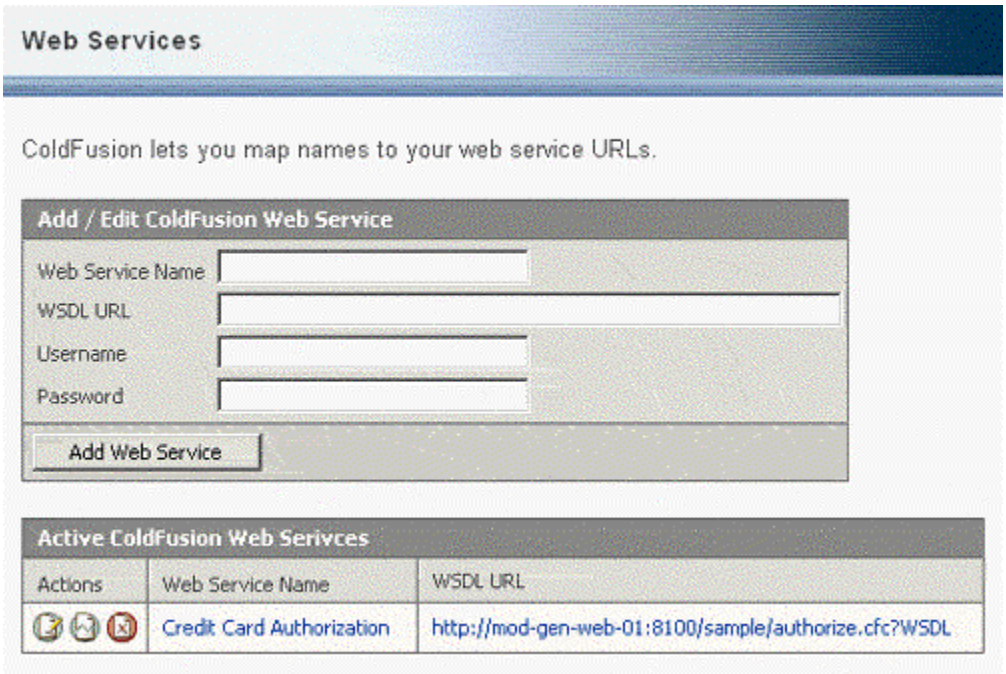

Figure 4: *Your entered web service in the ColdFusion administrator.*

Constructing a Client

Now that you have a working web service, you need to learn what to develop for the client side. An e-commerce website typically has a checkout page where the consumer enters his or her payment information. The customer submits the payment form and the form action sends the request to the payment processing service for authorization.

Here's an HTML form where your customer would enter payment information (step1.cfm):

```
<form action="step2.cfm" method="post">
   <table width="300" border="1">
 <tr>
    <td align="center" colspan="2">Joe's Credit Card Processing</td>
 </tr>
 <tr>
    <td>Name</td>
    <td><input name="name" type="text"></td>
 </tr>
 <tr>
    <td>Address</td>
    <td><input name="address" type="text"></td>
 </tr>
 <tr>
    <td>Zip</td>
    <td><input name="zip" type="text"></td>
 </tr>
 <tr>
    <td>State</td>
    <td><input name="state" type="text"></td>
 </tr>
 <tr>
    <td>Country</td>
    <td><input name="country" type="text"></td>
 </tr>
 <tr>
    <td>Card Type</td>
    <td>
<select name="cardtype">
      <option value="VI">Visa</option>
      <option value="MC">Mastercard</option>
   </select>
    </td>
 </tr>
 <tr>
    <td>Cardnumber</td>
    <td><input name="cardnumber" type="text"></td>
 </tr>
 <tr>
    <td>Expiry</td>
    <td><input name="expiry" type="text"></td>
 </tr>
 <tr>
    <td>Amount</td>
    <td><input name="amount" type="text"></td>
 </tr>
 <tr>
    <td align="right" colspan="2"><input type="submit"
   value="Process"></td>
```

```
 </tr>
   </table>
</form>
```

This code displays the screen shown in Figure 5 (without the information entered).

Joe's Credit Card Processing	
Name	Stacy Young
Address	123 My Place
Zip	H4B2Y1
State	Quebec
Country	Canada
Card Type	Mastercard
Cardnumber	5268010000000000
Expiry	05/04
Amount	100
	Process

Figure 5: *HTML entry form displayed in a browser*

Next, step2.cfm receives the POST from the HTML form, step1.cfm, and introduces two new tags: `cfinvoke` and `cfinvokeargument`:

```
<cfinvoke
 method="authorize"
 returnvariable="responseFromService"
 webservice="http://mod-gen-web-01:8100/sample/authorize.cfc?WSDL">
 <cfinvokeargument name="name"    value="#form.name#">
 <cfinvokeargument name="address" value="#form.address#">
 <cfinvokeargument name="zip"           value="#form.zip#">
 <cfinvokeargument name="state"   value="#form.state#">
 <cfinvokeargument name="country" value="#form.country#">
 <cfinvokeargument name="cardtype"      value="#form.cardtype#">
 <cfinvokeargument name="cardnumber" value="#form.cardnumber#">
 <cfinvokeargument name="expiry"  value="#form.expiry#">
 <cfinvokeargument name="amount"  value="#form.amount#">
</cfinvoke>
<cfdump var="#responseFromService#">
```

The `cfdump` tag displays the structure's data after the service has successfully processed the request (see Figure 6).

struct	
ADDRESS	123 My Place
AMOUNT	100
CARDNUMBER	5268010000000000
CARDTYPE	MC
COUNTRY	Canada
EXPIRY	05/04
NAME	Stacy Young
STATE	Quebec
ZIP	H4B2Y1
message	Your transaction was completed successfully, thank you.
reference	6853417
status	AUTHORIZED

Figure 6: *Data displayed in a structure from the cfdump tag.*

Web services really work. Using the new features in ColdFusion MX to create a standardized web service for our payment processing engine enables us at SureFire to drastically reduce the time and effort required in making our services available to customers. First, our customers benefit from SureFire's offering of standard protocols (converting our payment processing API to a web service). Second, Macromedia ColdFusion MX enables both us and our customers to connect to web services more easily by offering standard protocols to connect to SureFire's web service. In short, the customer benefits from shorter development times and lower support costs.

About the Author

Stacy Young is a web guru and technology evangelist at SureFire Commerce, located in Montreal, Canada. (www.surefirecommerce.com). With over seven years of experience in the IT world, Stacy has served time as a systems administrator, HTML junkie, Macromedia Flash enthusiast, ColdFusion fanatic, usability freak, bread maker, and aspiring author. He plays an active role in the FuseBox community (www.fusebox.org) and spends most of his free time replacing what his dog chews while he's at work. You can e-mail him directly at stacy.young@sfcommerce.com.