Building REST APIs with ColdFusion

ColdFusion makes writing REST API calls a simple task. With built in REST features you can create and publish a ColdFusion component or any functions in a component as a REST resource.

REST services can easily be generated by defining several attributes in the tags: cfcomponent, cffunction, and cfargument. With a few simple configurations, and a well-built ColdFusion component, you can publish your REST resource(s).

With RESTful Web Services, requests are made to a resource's URI. The response back from a REST service will be in a standard format of HTML, JSON, XML, and a few other potential options. HTTP is the most common protocol for interaction, and standard HTTP Methods like GET, POST, PUT, DELETE, HEAD, and OPTIONS are all likely method types that can be used while building services in ColdFusion. When using HTTP, you can access a REST endpoint via a browser, through code, or a client such as Postman.

When building ColdFusion REST Web Services, ColdFusion natively supports JSON and XML serialization/deserialization. This makes building and publishing Web Services a simple task, with less complexity and overhead in designing and building your services.

One additional advantage of building ColdFusion Web Services is the ability to publish the same ColdFusion component as a REST service and as a WSDL service, this allows for greater flexibility when publishing for potential subscribers.

In this tutorial you will have a firsthand look at registering, creating and consuming web services in ColdFusion. All examples will have sample code provided for your convenience. Let's first review some of the fundamentals with respect to webservices and ColdFusion

HTTP Responses

By default, ColdFusion provides default HTTP success and failure responses to the client as follows:

Success responses

Default HTTP Status	Description of HTTP Status Code
200 OK	Sent if the response has body.
204 No Content	Sent if the response does not have body.

Error Responses

Default HTTP Status	Description of HTTP Status Code
404 Not found	Request URL is not valid.
406 Not Acceptable	No function in the REST service can produce the MIME type requested by the client.
415 Unsupported Media Type Error	A resource is unable to consume the MIME type of client request.
405 Method not allowed	If the client invokes an HTTP method on a valid URI to which the request HTTP method is not bound.

Breaking Down the URL/Endpoint

The URL provided in the example can be interpreted as follows:

URL structure	Description
http://localhost:8500	Base URL which includes the IP address and port of the ColdFusion server.
rest	Implies that the request sent is a REST request. This default value can be renamed by revising the context path in web.xml available at cfusion/wwroot/WEB-INF and update the same mapping in uriworkermap.properties file found at config\wsconfig\1.
restTest	Application name or service mapping that you have used while registering the service in ColdFusion Administrator. If you do not specify a service mapping in the ColdFusion Administrator, then the application name is taken from Application.cfc.

restService	Rest path you defined in the service. That is, the value of the attribute restPath in the tag cfcomponent.

Specifying subresources

Functions in a REST service can either be a resource function, subresource function, or subresource locator.

Resource function

The functions for which you have not specified the RestPath at the function level in your Component but specified the httpMethod.

In this case, the resource function is invoked when the URL of the CFC is accessed. For example, if you do not provide the RestPath and you make a request from the browser, the code will map to a httpMethod of GET. This could be a problem if you have multiple GET’s. So, it is important to use the RestPath attribute at the function level of your Component if you believe your product will have multiple functions with the same method type.

Subresource function

The functions for which you have specified both resptPath and httpMethod.

Subresource functions are used to create subresources for the resource created in your component.

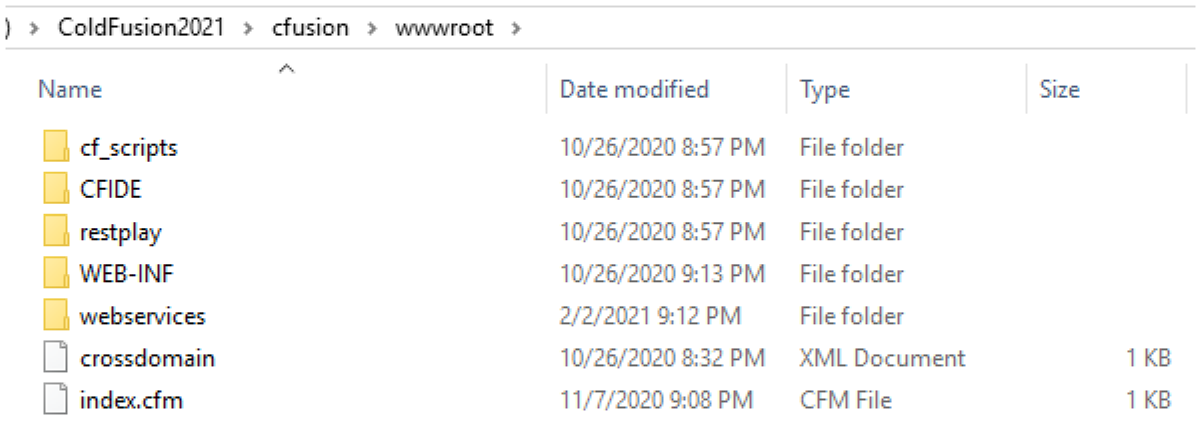
If the subresource has the httpMethod attribute in the cffunction you have declared, the function can directly handle HTTP requests as shown later in this tutorial. It is a safe bet to always use the httpMethod and specify the cffunction attributes for the restPath. This way you eliminate all ambiguity on a request.

Now that the housekeeping duties are tended to, let’s begin with some hands-on tutorials.

Creating REST Web Services with ColdFusion

You can create and publish a ColdFusion component (CFC) or any function in a component as a REST Resource.

1. First go to the root directory of your ColdFusion installation and create a directory for your service calls. The ColdFusion Components (CFCs) will live in this directory. For this example, you can create a directory called webservices.



) > ColdFusion2021 > cfusion > wwwroot >

Name	Date modified	Type	Size
cf_scripts	10/26/2020 8:57 PM	File folder	
CFIDE	10/26/2020 8:57 PM	File folder	
restplay	10/26/2020 8:57 PM	File folder	
WEB-INF	10/26/2020 9:13 PM	File folder	
webservices	2/2/2021 9:12 PM	File folder	
crossdomain	10/26/2020 8:32 PM	XML Document	1 KB
index.cfm	11/7/2020 9:08 PM	CFM File	1 KB

2. When creating a Component (CFC) specify the cfcomponent tag to one of the following: restPath or rest.

   In this example we will use restPath and set it equal to “customers”.

   We have set the rest attribute equal to “true”.

   And for good practice we have set the name attribute equal to “Customers”.

Tag Based Syntax:

```
<cfcomponent rest="true"  restpath="customers" Name="Customers">
```

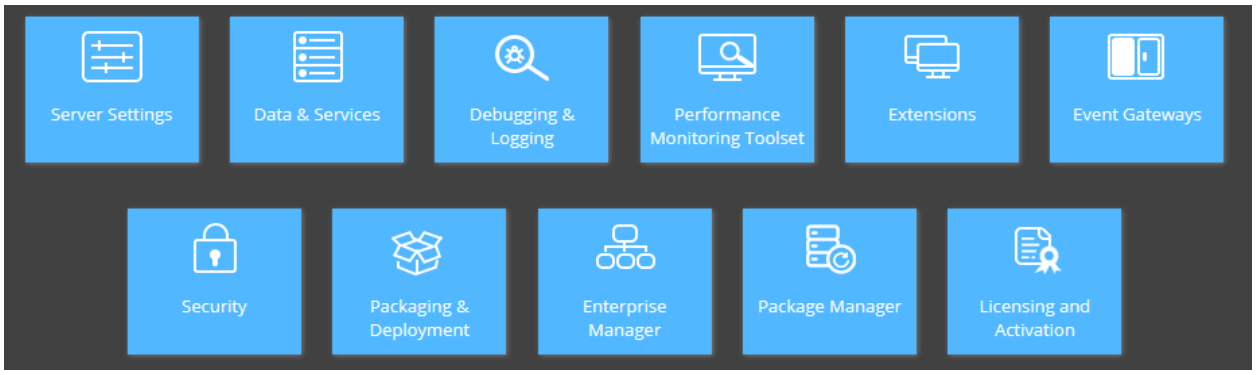
Script Based Syntax:

```
component rest="true" restpath="/customers" Name="Customers"{
```

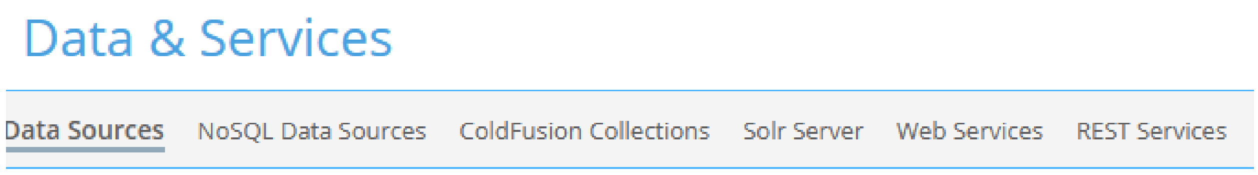
Next, you must register your REST API Services in ColdFusion. We will cover two options in how to complete this. We will assume you have access to the ColdFusion Administrator and cover that scenario first. The second option can be done directly through code in ColdFusion, in the event you do not have access to the administrator.

ColdFusion Administrator (REST Service Registration)

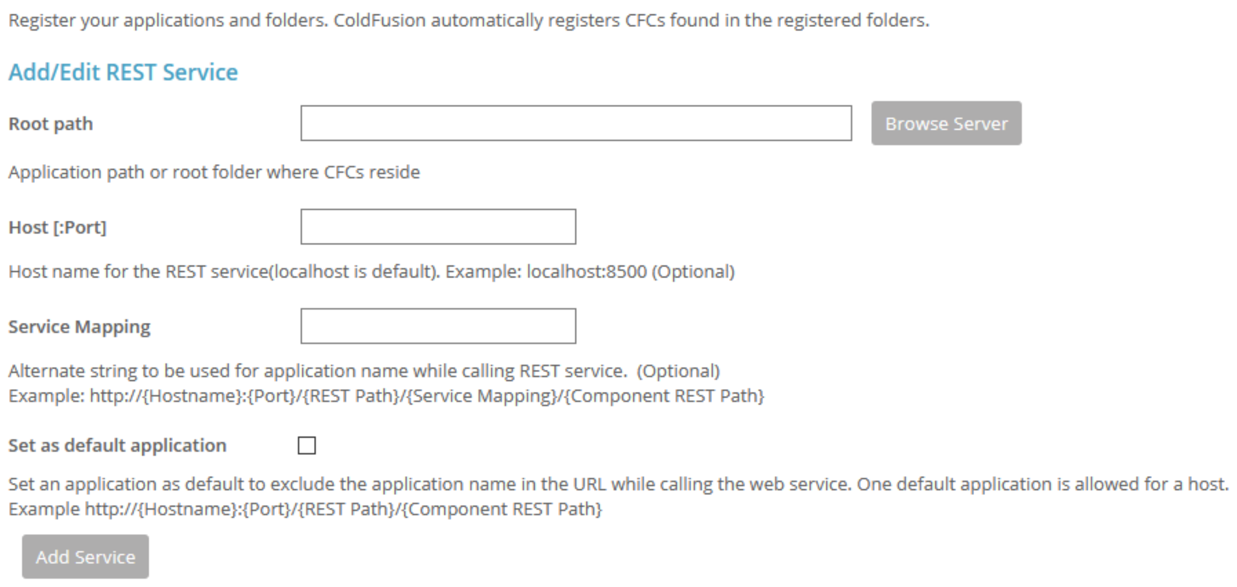
1. Log into the ColdFusion Administrator and click on the Data & Services. You should be able to access the administrator by visiting this link (http://localhost:8500/CFIDE/administrator/index.cfm).



2. Click on the Web Services option


Data & Services

3. Click on the Browse Server option and select the directory where your Rest Components will live. Earlier, we created the webservices directory in our ColdFusion root installation. Let’s continue to use this for demo purposes.


Register your applications and folders. ColdFusion automatically registers CFCs found in the registered folders.

Add/Edit REST Service

Root path

Application path or root folder where CFCs reside

Host [:Port]

Host name for the REST service(localhost is default). Example: localhost:8500 (Optional)

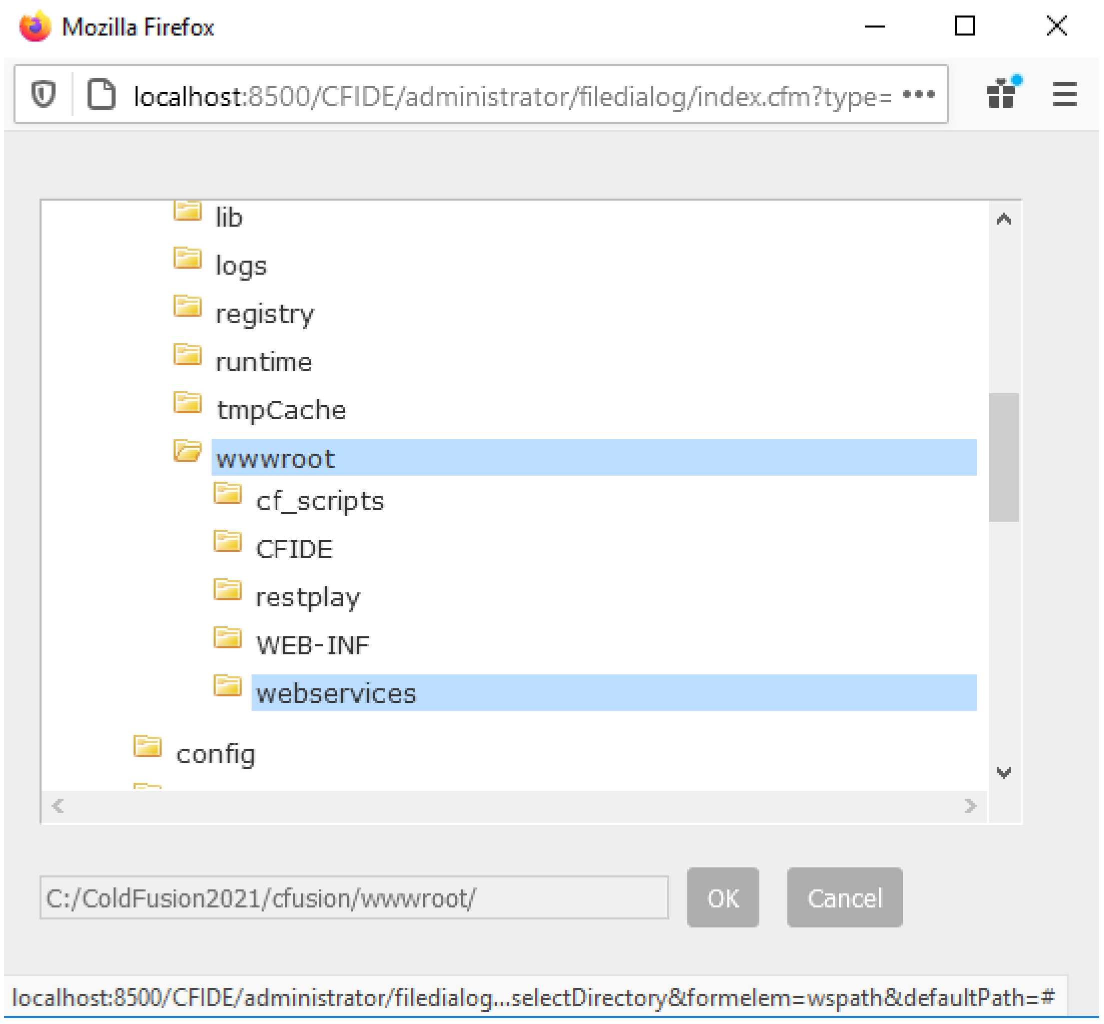
Service Mapping

Alternate string to be used for application name while calling REST service. (Optional)
Example: http://{Hostname}:{Port}/{REST Path}/{Service Mapping}/{Component REST Path}

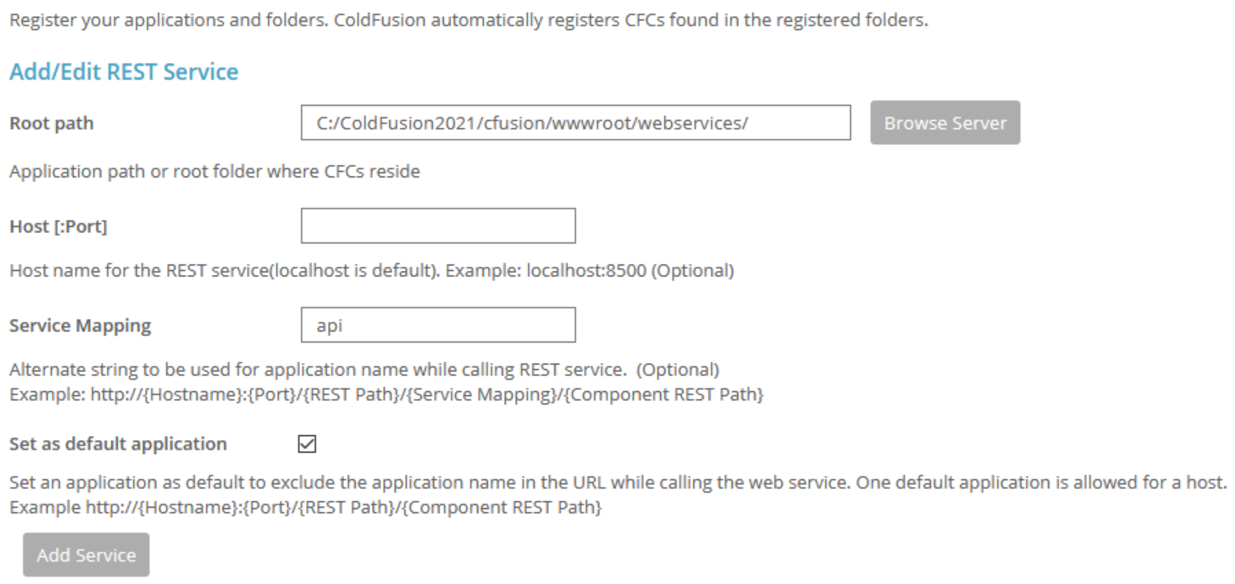
Set as default application ☐

Set an application as default to exclude the application name in the URL while calling the web service. One default application is allowed for a host.
Example http://{Hostname}:{Port}/{REST Path}/{Component REST Path}

4. Select the directory webservices and click on OK


Mozilla Firefox
localhost:8500/CFIDE/administrator/filedialog/index.cfm?type=
lib
logs
registry
runtime
tmpCache
wwwroot
cf_scripts
CFIDE
restplay
WEB-INF
webservices
config
C:/ColdFusion2021/cfusion/wwwroot/
OK
Cancel
localhost:8500/CFIDE/administrator/filedialog...selectDirectory&formelem=wspath&defaultPath=#

5. Select the service mapping name. In this tutorial we will call it "api", and we will set it as the "default application" and click on "Add Service"



Register your applications and folders. ColdFusion automatically registers CFCs found in the registered folders.

Add/Edit REST Service

Root path	C:/ColdFusion2021/cfusion/wwwroot/webservices/
Host [:Port]	
Service Mapping	api
Set as default application	✓

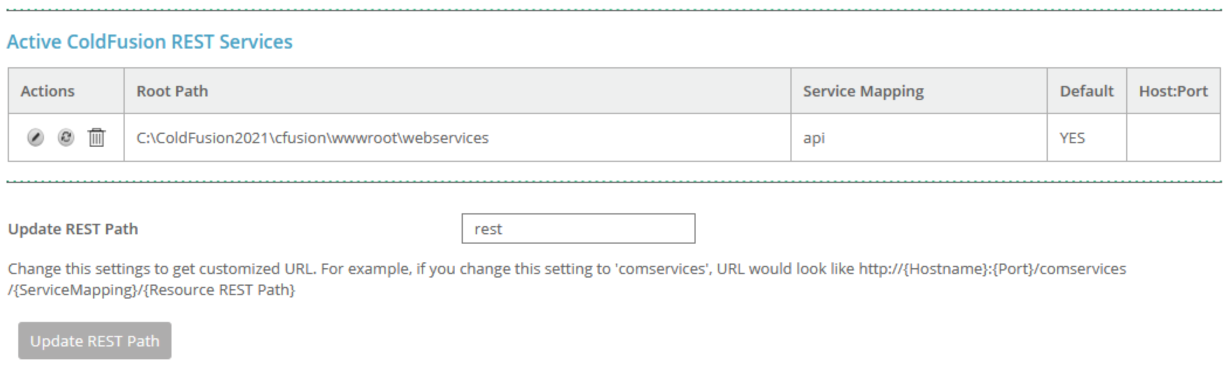
Application path or root folder where CFCs reside

Host name for the REST service(localhost is default). Example: localhost:8500 (Optional)

Alternate string to be used for application name while calling REST service. (Optional)
Example: http://{Hostname}:{Port}/{REST Path}/{Service Mapping}/{Component REST Path}

Set an application as default to exclude the application name in the URL while calling the web service. One default application is allowed for a host.
Example http://{Hostname}:{Port}/{REST Path}/{Component REST Path}

6. After the service is created, you should see the meta data associated with it listed in the "Active ColdFusion REST Services". This completes the Administration configuration.



Active ColdFusion REST Services

Actions	Root Path	Service Mapping	Default	Host:Port
	C:\ColdFusion2021\cfusion\wwwroot\webservices	api	YES	

Update REST Path	rest

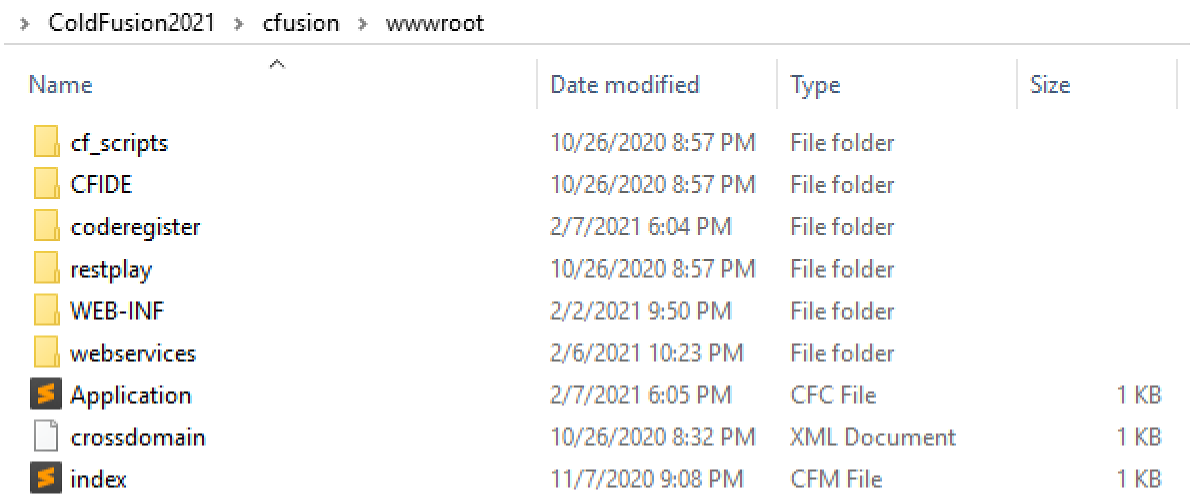
Change this settings to get customized URL. For example, if you change this setting to 'comservices', URL would look like http://{Hostname}:{Port}/comservices /{ServiceMapping}/{Resource REST Path}

7. The contextual endpoint for your services can now be invoked by calling http://localhost:8500/rest/api/{Mapping and Sub Resources}

We will now cover one additional option if your access to the ColdFusion administrator is not available.

ColdFusion Programmatic REST API Registration (Application.cfc)

1. First you will need to add a Application.cfc to the root directory of your installation.



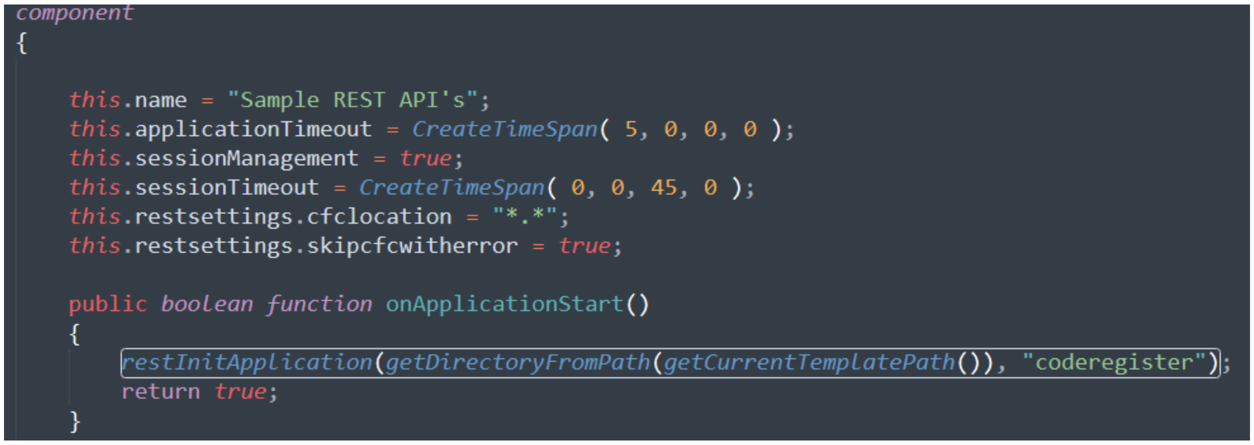
› ColdFusion2021 › cfusion › wwwroot

Name	Date modified	Type	Size
cf_scripts	10/26/2020 8:57 PM	File folder	
CFIDE	10/26/2020 8:57 PM	File folder	
coderegister	2/7/2021 6:04 PM	File folder	
restplay	10/26/2020 8:57 PM	File folder	
WEB-INF	2/2/2021 9:50 PM	File folder	
webservices	2/6/2021 10:23 PM	File folder	
Application	2/7/2021 6:05 PM	CFC File	1 KB
crossdomain	10/26/2020 8:32 PM	XML Document	1 KB
index	11/7/2020 9:08 PM	CFM File	1 KB

2. Add the "restInitApplication" function and provide the path to your directory where your service components are located.

   Utilizing the getDirectoryFromPath function and getCurrentTemplatePath function, you can dynamically set these values and then simply specify the "Service Mapping" name. For demonstration purposes I have named this sample "coderegister"so it will not cause conflict with the previous Administrator registration.

   In the earlier part of the tutorial we utilized the Administrator directly. In this case, the Application.cfc will register the services in the system, and they will appear in the Administrator automatically.

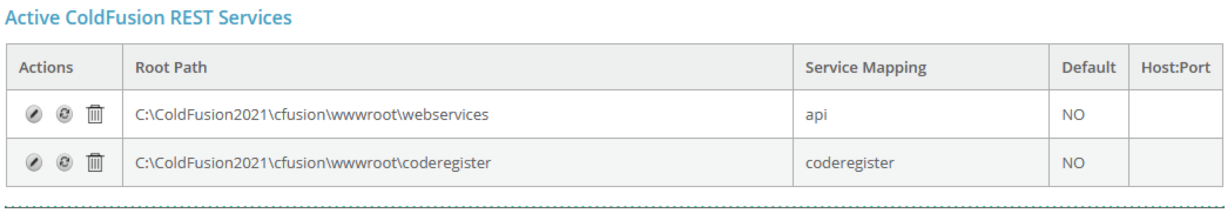


```
component
{

    this.name = "Sample REST API's";
    this.applicationTimeout = CreateTimeSpan( 5, 0, 0, 0 );
    this.sessionManagement = true;
    this.sessionTimeout = CreateTimeSpan( 0, 0, 45, 0 );
    this.restsettings.cfclocation = "*.*";
    this.restsettings.skipcfcwitherror = true;

    public boolean function onApplicationStart()
    {
        restInitApplication(getDirectoryFromPath(getCurrentTemplatePath()), "coderegister");
        return true;
    }
```

If you do have access to the ColdFusion Administrator, you can verify that the REST service has been registered and is active. You can see below the "coderegister" service group is now available and ready to take requests.



Active ColdFusion REST Services

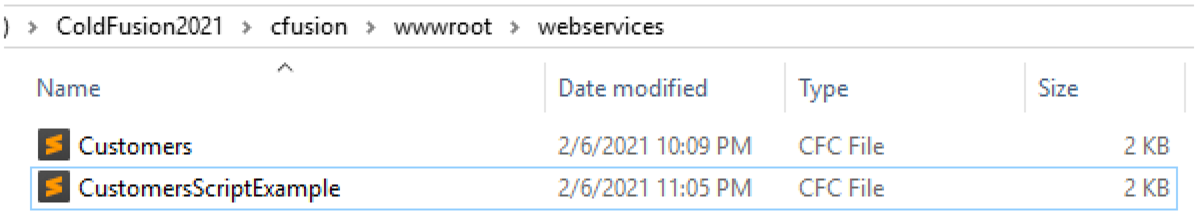
Actions	Root Path	Service Mapping	Default	Host:Port
	C:\ColdFusion2021\cfusion\wwwroot\webservices	api	NO	
	C:\ColdFusion2021\cfusion\wwwroot\coderegister	coderegister	NO	

3. The contextual endpoint for your services can now be invoked by calling http://localhost:8500/rest/coderegister/{Mappings and Sub Resources}

Sample Components and Functions to build your REST APIs

To illustrate some sample code, we will continue with the samples we have already provided. In the webservices directory that we mapped in ColdFusion earlier, we will create a Component (CFC) called Customers.

We will also create a CustomersScriptBased Component (CFC) to illustrate the same process but without tags. If you prefer to write only in script those samples will be provided as well.

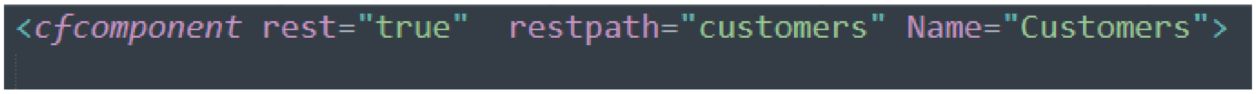


) › ColdFusion2021 › cfusion › wwwroot › webservices

Name	Date modified	Type	Size
Customers	2/6/2021 10:09 PM	CFC File	2 KB
CustomersScriptExample	2/6/2021 11:05 PM	CFC File	2 KB

Let’s take a closer look at the Customers.cfc. We will begin by writing a simple function called get, and in that an array to store all customers. Take note of some key attributes that we have specified in the CFC.

For the Customers.cfc we have used the syntax below (tag based):



```
<cfcomponent rest="true"  restpath="customers" Name="Customers">
```

The rest argument is equal to true and the restpath was set to customers. It is not required to match the name of the CFC. In this case, we will keep it simple and consistent.

Next, we look at the function tag. It is here we declare the name of our method and call it “get”. We set the access equal to “remote”. Even though this will return a JSON payload the data type is set to “array”, since ColdFusion handles the serialization of JSON automatically. We make sure to set the produces equal to “application/json” and lastly the httpmethod is set to “GET”.

One additional thing to understand, we did not set the restPath for the get. Since you will invoke this from the URL, and there is no other httpMethod of GET in the entire class, the routing will not be an issue.

ColdFusion will know to invoke this class and method and process the request. If we did have multiple GETs in the CFC this would be a problem. For good practice, it makes logical sense to provide the restPath at a function level.

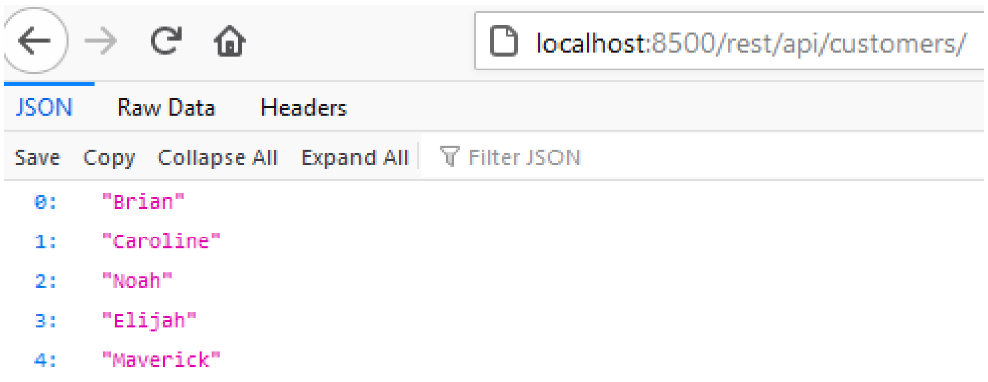
GET API Call Tag Based

```
 //GET Request to return customers
<cffunction name="get" access="remote" returnType="array" produces="application/json" httpmethod="GET">
    <cfset customers = arrayNew(1)>
    <cfset customers[1] = "Brian">
    <cfset customers[2] = "Caroline">
    <cfset customers[3] = "Noah">
    <cfset customers[4] = "Elijah">
    <cfset customers[5] = "Maverick">
    <cfreturn customers />
</cffunction>
```

GET API Call Script Based

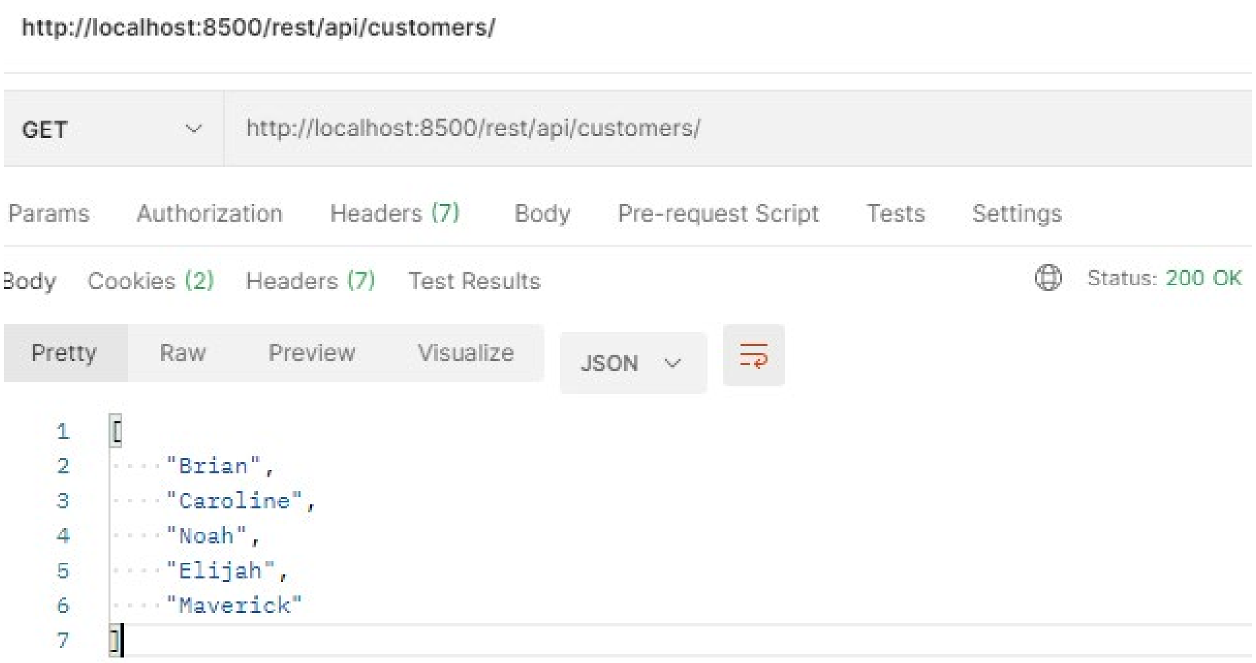
```
//GET Request to return customers
function get () access="remote" returntype="array" produces="application/json" httpmethod="GET" {
    var customers = [];
    customers[1] = "Brian";
    customers[2] = "Caroline";
    customers[3] = "Noah";
    customers[4] = "Elijah";
    customers[5] = "Maverick";
    return customers;
}
```

If you invoke from the browser you can see the data is returned in a JSON format.



```
0:  "Brian"
1:  "Caroline"
2:  "Noah"
3:  "Elijah"
4:  "Maverick"
```

Inokving it from Postman will return the same response, with a few more details.



```
[
    "Brian",
    "Caroline",
    "Noah",
    "Elijah",
    "Maverick"
]
```

POST API Calls Tag Based

In this example, we created a POST request. Normally this would be used to generate an entity in a database.

For example, you want to add a new customer into your data layer. We kept the verb naming convention for the method calling it "post". The access attribute stays the same with "remote". Since we are not returning a detailed payload and simply returning a success message, the returnType is set to a "string". The produces attribute will be "application/json" and lastly, the httpMethod should equal POST.

One noticeable thing with the function attributes, the restPath is missing, since we only have one POST method in the CFC this will still map correctly and execute.

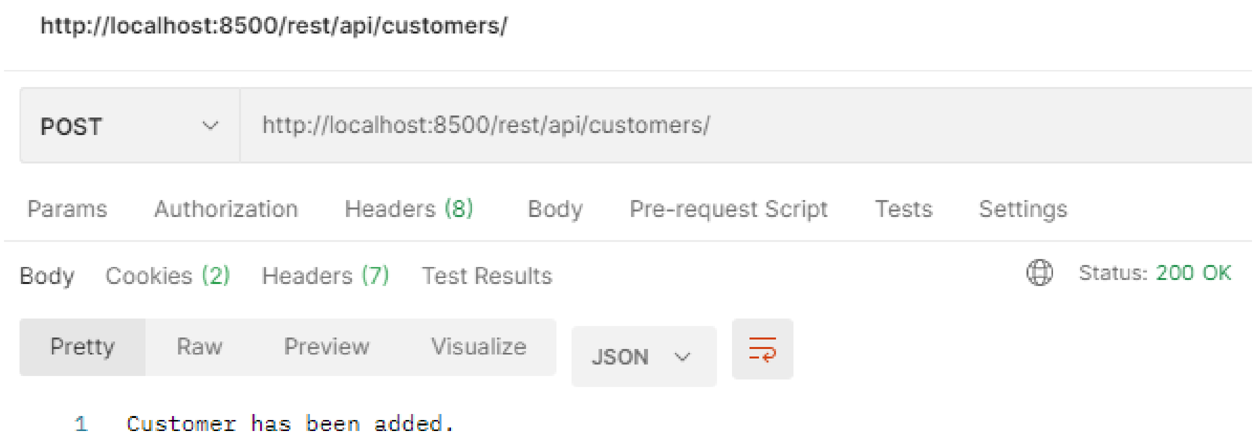
```
//POST Request to write a new customer to the database
<cffunction name="post" access="remote" returnType="string" produces="application/json" httpmethod="POST">
    //do database insert and validation logic
    //return a successful message in the response
    <cfset message = "Customer has been added.">
    <cfreturn message>
</cffunction>
```

POST API Call Script Based

```
//POST Request to write a new customer to the database
function post () access="remote" returntype="string" produces="application/json" httpmethod="POST" {
    //do database insert and validation logic
    //return a successful message in the response
    var message = "Customer has been added.";
    return message;
}
```

We will run this example through Postman instead of a web browser. Normally, you would have a JSON payload with information when calling a POST method. However, to keep things simple we have ommitted this from this scenairo.

Make sure when calling the endpoint you select POST in Postman, and provide the correct URL. In the example below, it executed successfully and returned the success message we coded in the sample method.



```
Customer has been added.
```

PUT API Call Tag Based

In this example, we created a PUT request. Normally this would be used to generate an entity or modify an entity in a database. The POST/PUT debate is always an area for strong opinions of which should be used for creates and updates, but again to keep things simple, a PUT in this example will be used for an update to an existing entity.

For example, you want to edit/update an existing customer in your data layer.

We kept the verb naming convention in place for this method, calling it "put". The access attribute stays the same with "remote". Since we are not returning a detailed payload and simply returning a success message, the returnType is set to a "string". The produces attribute will be "application/json" and lastly, the httpMethod should equal PUT.

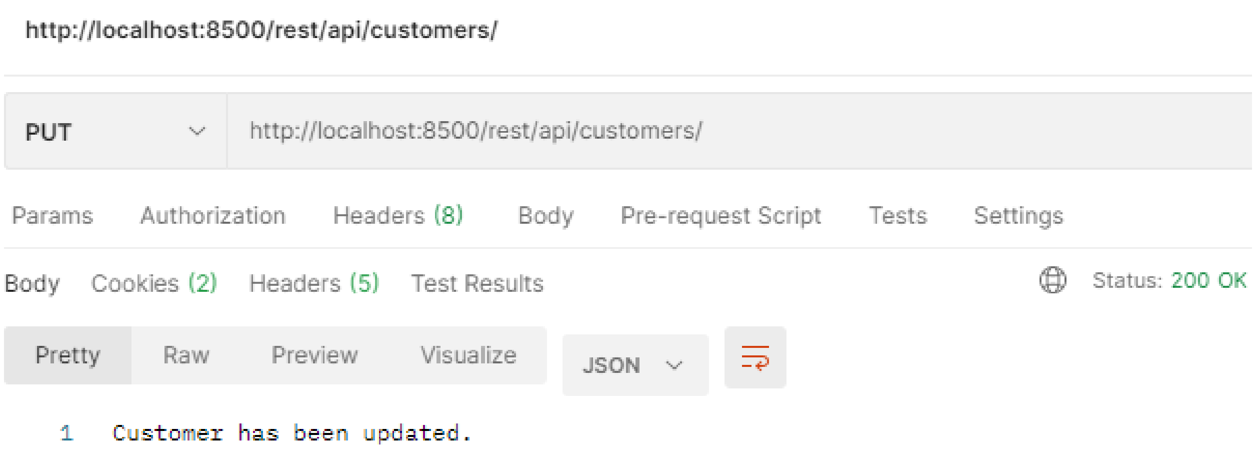
One thing to note, the restPath is missing, since we only have one PUT method in the CFC this will still map correctly and execute.

```
//PUT Request to edit a customer in the database
<cffunction name="put" access="remote" returnType="string" produces="application/json" httpmethod="PUT">
    //do database update and validation logic
    //return a successful message in the response
    <cfset message = "Customer has been updated.">
    <cfreturn message>
</cffunction>
```

PUT API Call Script Based

```
//PUT Request to edit a customer in the database
function put () access="remote" returntype="string" produces="application/json" httpmethod="PUT" {
    //do database update and validation logic
    //return a successful message in the response
    var message = "Customer has been updated.";
    return message;
}
```

To test the PUT in Postman simply change the method option to PUT, and execute. You can see the message has returned and the customer has been updated.


http://localhost:8500/rest/api/customers/

```
Customer has been updated.
```

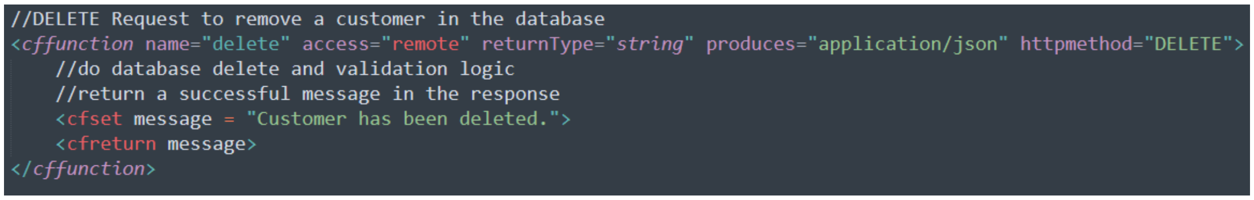
DELETE API Call Tag Based

In this example, we created a DELETE request. Normally this would be used to delete an entity in a database.

For example, you want to delete an existing customer in your data layer.

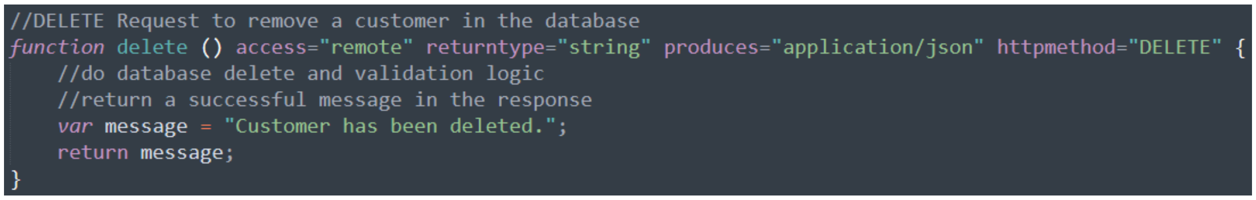
We kept the verb naming convention for this method calling it “delete”. The access attribute stays the same with “remote”. Since we are not returning a detailed payload and simply returning a success message, the returnType is set to a “string”. The produces attribute will be "application/json” and lastly, the httpMethod should equal DELETE.

One thing to note, the restPath is missing, since we only have one DELETE method in the CFC this will still map correctly and execute.



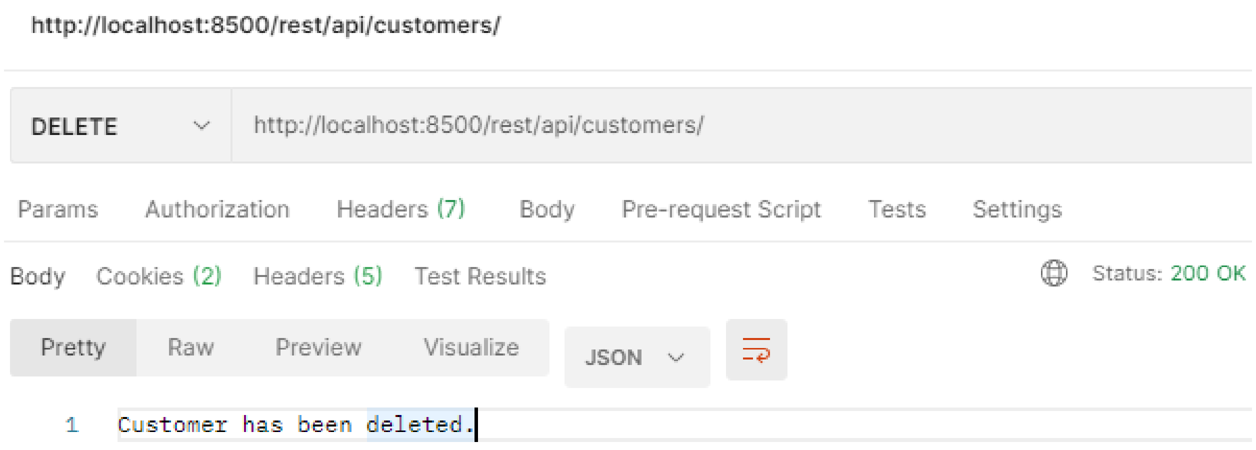
```
//DELETE Request to remove a customer in the database
<cffunction name="delete" access="remote" returnType="string" produces="application/json" httpmethod="DELETE">
    //do database delete and validation logic
    //return a successful message in the response
    <cfset message = "Customer has been deleted.">
    <cfreturn message>
</cffunction>
```

DELETE API Call Script Based



```
//DELETE Request to remove a customer in the database
function delete () access="remote" returntype="string" produces="application/json" httpmethod="DELETE" {
    //do database delete and validation logic
    //return a successful message in the response
    var message = "Customer has been deleted.";
    return message;
}
```

To test the DELETE in Postman simply change the method option to DELETE, and execute. You can see the message has returned and the customer has been deleted.



http://localhost:8500/rest/api/customers/

DELETE http://localhost:8500/rest/api/customers/

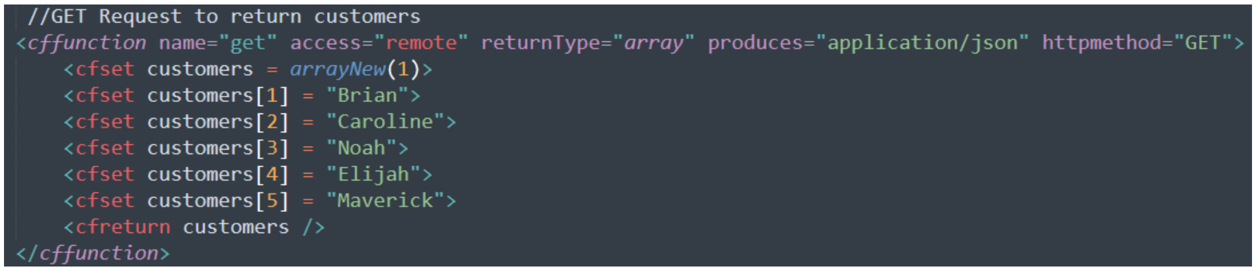
Status: 200 OK

```
Customer has been deleted.
```

Sub Resources and Multiple HTTP Methods

The examples provided earlier are great starting points. At some point your APIs will most certianly evovle in complexity and logic. Having multiple Methods that are the same with respect to the httpMethod will cause conflicts if you don’t properly identify which call routes to which function in your component.

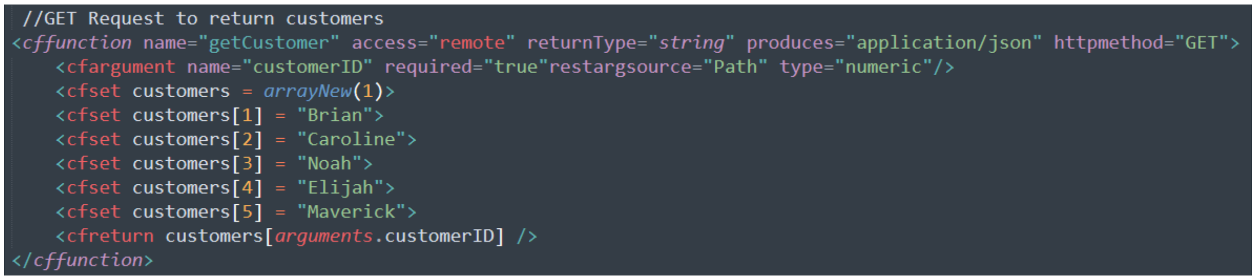
In our earlier exmaple we illustrated a simple get which returned all customers.



```
 //GET Request to return customers
<cffunction name="get" access="remote" returnType="array" produces="application/json" httpmethod="GET">
    <cfset customers = arrayNew(1)>
    <cfset customers[1] = "Brian">
    <cfset customers[2] = "Caroline">
    <cfset customers[3] = "Noah">
    <cfset customers[4] = "Elijah">
    <cfset customers[5] = "Maverick">
    <cfreturn customers />
</cffunction>
```

Suppose you wanted to have a second GET method. In this method you would like to accept function parameters to request a specific entity in the database. We created a second method to illustrate this.

We added a cfargument tag to account for a dynamic parameter. In this example, it is called customerID and we made it required by setting the argument to “true”. The restargsource is set to “Path” since it is a path param and not a query param or a body parameter coming in on the request. Lastly, the incoming parameter will be a “numeric” value.

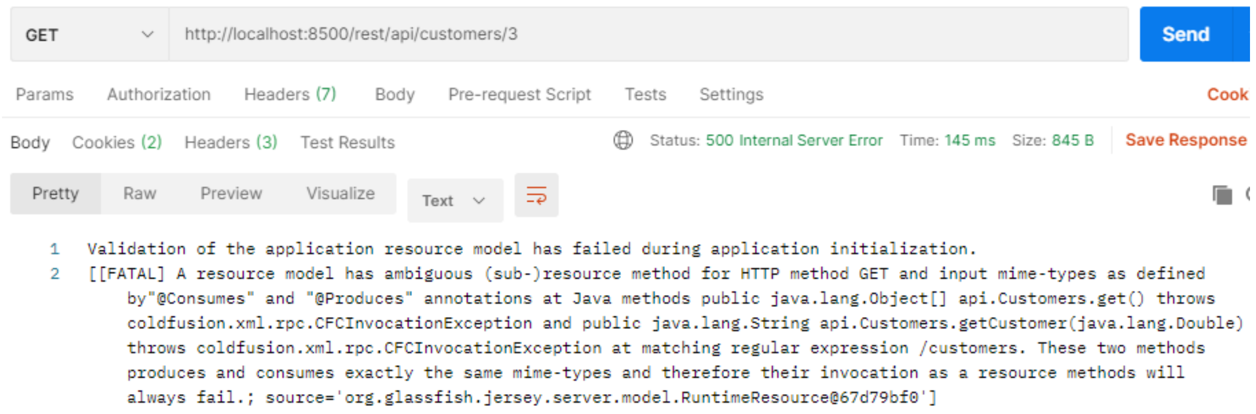


```
 //GET Request to return customers
<cffunction name="getCustomer" access="remote" returnType="string" produces="application/json" httpmethod="GET">
    <cfargument name="customerID" required="true"restargsource="Path" type="numeric"/>
    <cfset customers = arrayNew(1)>
    <cfset customers[1] = "Brian">
    <cfset customers[2] = "Caroline">
    <cfset customers[3] = "Noah">
    <cfset customers[4] = "Elijah">
    <cfset customers[5] = "Maverick">
    <cfreturn customers[arguments.customerID] />
</cffunction>
```

In order to test this we will modify our Postman call and execute. As you can see below the url is now http://localhost:8500/rest/api/customers/3

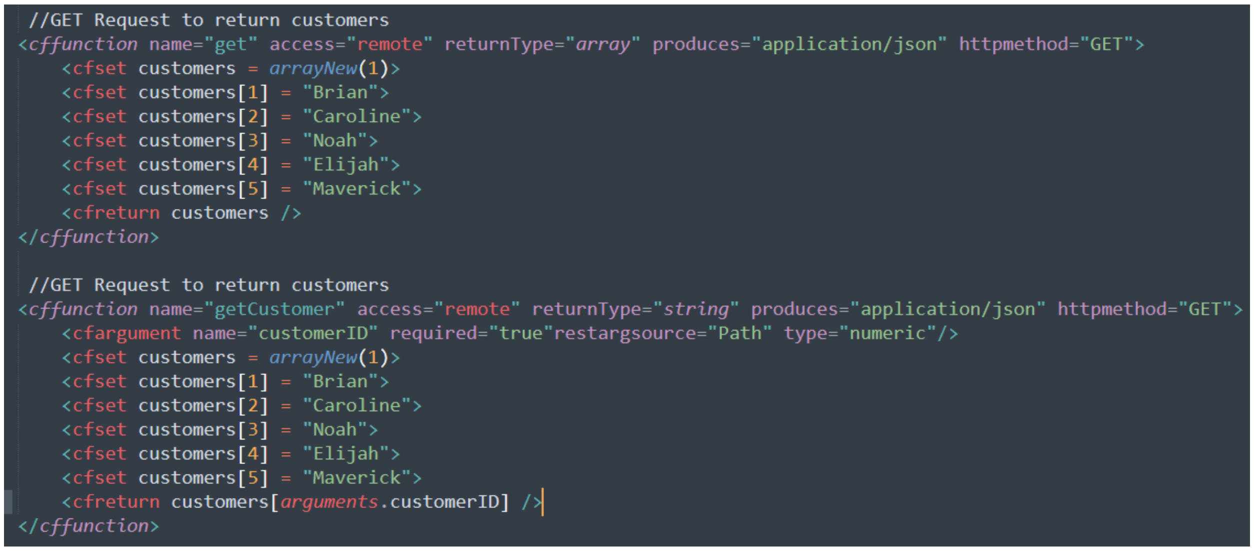
We added the /3 to return customer in position 3. This is a URL Path Parameter. It could also be passed in on the request through the query parameter options. An example would be http://localhost:8500/rest/api/customers?customerID=3

In the earlier examples we did not specify the restPath for each method. In doing so, we now have two GET methods in one CFC. This creates ambiguity and we now encounter an error. See the end result below. In order to avoid this we will create and identify the sub resource in the code.


GET
http://localhost:8500/rest/api/customers/3
Send
Params Authorization Headers (7) Body Pre-request Script Tests Settings Cook
Body Cookies (2) Headers (3) Test Results
Status: 500 Internal Server Error Time: 145 ms Size: 845 B Save Response
Pretty Raw Preview Visualize Text

```
Validation of the application resource model has failed during application initialization.
[[FATAL] A resource model has ambiguous (sub-)resource method for HTTP method GET and input mime-types as defined
    by"@Consumes" and "@Produces" annotations at Java methods public java.lang.Object[] api.Customers.get() throws
    coldfusion.xml.rpc.CFCInvocationException and public java.lang.String api.Customers.getCustomer(java.lang.Double)
    throws coldfusion.xml.rpc.CFCInvocationException at matching regular expression /customers. These two methods
    produces and consumes exactly the same mime-types and therefore their invocation as a resource methods will
    always fail.; source='org.glassfish.jersey.server.model.RuntimeResource@67d79bf0']
```

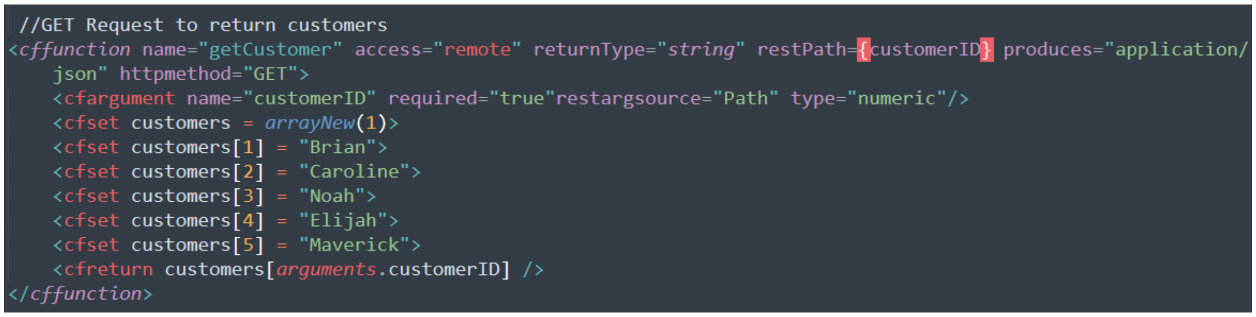
Here is the bad code, and how it is causing conflict on the request. As you can see, there are two GET methods in the Class (CFC) and no restPath attribute. We will add a restPath attribute to the second call so routing can be correct and data can be returned.



```
 //GET Request to return customers
<cffunction name="get" access="remote" returnType="array" produces="application/json" httpmethod="GET">
    <cfset customers = arrayNew(1)>
    <cfset customers[1] = "Brian">
    <cfset customers[2] = "Caroline">
    <cfset customers[3] = "Noah">
    <cfset customers[4] = "Elijah">
    <cfset customers[5] = "Maverick">
    <cfreturn customers />
</cffunction>

 //GET Request to return customers
<cffunction name="getCustomer" access="remote" returnType="string" produces="application/json" httpmethod="GET">
    <cfargument name="customerID" required="true"restargsource="Path" type="numeric"/>
    <cfset customers = arrayNew(1)>
    <cfset customers[1] = "Brian">
    <cfset customers[2] = "Caroline">
    <cfset customers[3] = "Noah">
    <cfset customers[4] = "Elijah">
    <cfset customers[5] = "Maverick">
    <cfreturn customers[arguments.customerID] />
</cffunction>
```

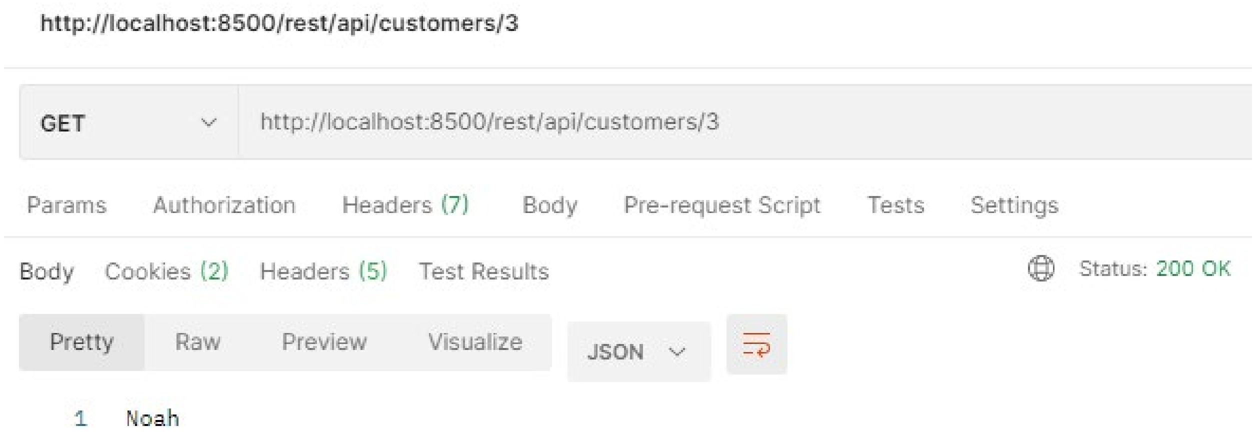
Correct the code with the restPath attribute specified in the function attributes. We will add it to the second GET as that one will receive the incoming Customer ID and match it correctly against the array of customers. Ideally, the request should only return one customer based on the ID of the customer that was passed in on the request through the path parameter. See the changes below.



```
 //GET Request to return customers
<cffunction name="getCustomer" access="remote" returnType="string" restPath={customerID} produces="application/
    json" httpmethod="GET">
    <cfargument name="customerID" required="true"restargsource="Path" type="numeric"/>
    <cfset customers = arrayNew(1)>
    <cfset customers[1] = "Brian">
    <cfset customers[2] = "Caroline">
    <cfset customers[3] = "Noah">
    <cfset customers[4] = "Elijah">
    <cfset customers[5] = "Maverick">
    <cfreturn customers[arguments.customerID] />
</cffunction>
```

The {customerID} is in {} indicating a dynamic path parameter. Let's update the endpoint in Postman and check the result.

As you can see, the customers/3 is indicating we would like to retrieve customer 3. With the code corrected and zero ambiguity in your ColdFusion application layer the api works and returns customer Noah.



http://localhost:8500/rest/api/customers/3

GET http://localhost:8500/rest/api/customers/3

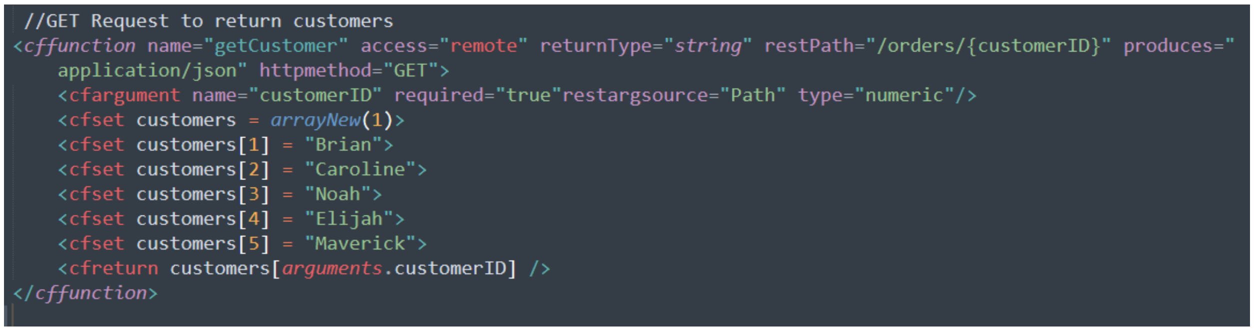
Status: 200 OK

```
1    Noah
```

Please keep in mind this value could be a string path with dynamic parameters as well. For example, "/orders/{customerID}" would now have a similar endpoint to:

http://localhost:8500/rest/api/customers/orders/3

As you can see in the code example below the restPath is now updated to "/orders/{customerID}" This allows for specifying the sub resource and allows ColdFusion to map the incoming request correctly.



```
 //GET Request to return customers
<cffunction name="getCustomer" access="remote" returnType="string" restPath="/orders/{customerID}" produces="
    application/json" httpmethod="GET">
    <cfargument name="customerID" required="true"restargsource="Path" type="numeric"/>
    <cfset customers = arrayNew(1)>
    <cfset customers[1] = "Brian">
    <cfset customers[2] = "Caroline">
    <cfset customers[3] = "Noah">
    <cfset customers[4] = "Elijah">
    <cfset customers[5] = "Maverick">
    <cfreturn customers[arguments.customerID] />
</cffunction>
```

Conclusion

ColdFusion does a fantastic job at helping to create and publish API's. This is just the tip of the iceberg and if you are comfortable with writing ColdFusion functions transitioning them to REST ready calls is not difficult.

The complexity of payloads, marshalling, routing, and orchestration is alleviated by taking advantage of the built in REST features. After all, if your future proofing and building things to scale why not create services and be ready for potential service-based opportunities that may enter your environment at some point.