ColdBox REST APIs

www.coldbox.org

Covers up to version 3.8.1

Contents

REST APIs are a popular and easy way to add HTTP endpoints to your web applications to act as web services for third parties or even other internal systems. REST is simpler and requires less verbosity and overhead than other protocols such as SOAP or XML-RPC. Creating a fully-featured REST API is easy with the ColdBox Platform. Everything you need for creating routes, massaging data, and enforcing security comes out of the box.

REST stands for **Representational State Transfer** and builds upon the basic idea that data is represented as **resources** and accessed via a **URI**, or unique address. An HTTP client (such as a browser, or the CFHTTP tag) can send requests to a URI to interact with it. The HTTP verb (GET, POST, etc) sent in the header of the request tells the server how the client want to interact with that resource.

As far as how your data is formatted or how you implement security is left up to you. REST is less prescriptive than other standards such as SOAP (which uses tons of heavy XML and strictly-typed parameters). This makes REST more lightweight, easier to understand, and simpler to test and debug.

DEFINING RESOURCES

Resources should usually be nouns as they represent the objects being accessed or modified by the API. A REST API can define its resources on its own domain (`http://api.example.com`), or after a static placeholder that differentiates it from the rest of the app (`http://www.example.com/api/`). We'll use the latter for these examples. Let's consider a resource we need to represent called "`user`". It is also important to note that REST is a style of URL architecture not a mandate, so it is an open avenue of sorts. However, you must stay true to its concepts of resources and usage of the HTTP verbs. Here are a few pointers when using the HTTP verbs:

```
http://www.example.com/api/user
```

- **GET /api/user** will return a list representation of all the users. It is permissible to use a query string to control pagination or filtering.
- **POST /api/user/** will create a new user
- **GET /api/user/53** will return a single representation of user 53
- **PUT /api/user/53** will update user 53
- **DELETE /api/user/53** will delete user 53

The GETs will return the results in the body of the HTTP response. The POST and PUT would expect to receive data in the body of the HTTP request. Whether or not the DELETE returns data (other than its status code) is up to you.

Note: GET, PUT, and DELETE methods should be idempotent which means repeated requests to the same URI don't do anything. Repeated POST calls however, would create multiple users.

In ColdBox, the easiest way to represent our "`/api/user`" resource is to create a handler called "`user.cfc`" in the `/handlers/api/` directory. In this instance, ColdBox will consider the "api" to be a handler package. Here in my handler, I have stubbed out actions for each of the operations I need to perform against my user resource.

/handlers/api/user.cfc

```
component {
    function index( event, rc, prc ) {
        // List all users
    }
    function view( event, rc, prc ) {
        // View a single user
    }
    function save( event, rc, prc ) {
        // Save a user
    }
    function remove( event, rc, prc ) {
        // Remove a user
    }
}
```

DEFINING URL ROUTES

The code above will create default routes based on ColdBox's conventions. You can also have full control of what the URL looks like though by mapping flexible URL patterns and HTTP verbs to the handler and action that will service each request. To do this, use the `/config/routes.cfm` file to declare URL routes we want the application to capture and define how to process them.

Let's add the following new routes to our `/config/routes.cfm` file BEFORE the default route.

```
// Map base route to list users
addRoute(
    pattern = 'api/user',
    handler = 'api.user',
    action = 'index'
);

// Map route to specific user.
// Different verbs call different actions!
addRoute(
    pattern = 'api/user/:userID',
    handler = 'api.user',
    action = {
        GET = 'view',
        POST = 'save',
        PUT = 'save',
        DELETE = 'remove'
});
```

You can see if that if action is a string, all HTTP verbs will be mapped there, however a struct can also be provided that maps different verbs to different actions. This gives you exact control over how the requests are routed.

The `:userID` part of the route pattern is a placeholder. It matches whatever text is in the URL in that position. The value of the text that is matched will be available to you in the request collection as `rc.userID`. You can get even more specific about what kind of text you want to match in your route pattern by specifying patterns that only match numeric/alpha characters, or take full control with regex patterns.

More on pattern matching can be found in the docs here:
http://wiki.coldbox.org/wiki/Building_Rest_APIs.cfm#Route_Placeholders

RETURNING DATA

REST does not dictate the format of data you use to represent your data. It can be JSON, XML, WDDX, plain text, or something else of your choosing. The most common way to return data from your handler's action is to use the event object's `renderData()` method. It takes complex data and turns it into a string representation. Here are some of the most common formats supported by `event.renderData():`

- XML
- JSON
- TEXT
- WDDX
- PDF
- HTML

Using ColdBox's data rendering in your REST handlers looks like this:

```
// xml marshalling
function getUsersXML( event, rc, prc ){
    var qUsers = getUserService().getUsers();
    event.renderData( type="XML", data=qUsers );
}
// json marshalling
function getUsersJSON( event, rc, prc ){
    var qUsers = getUserService().getUsers();
    event.renderData( type="json", data=qUsers );
}
```

Many APIs allow the user to choose the format they want back from the endpoint by appending a file "extension" to the end of the URL.

```
http://www.example.com/api/user.json
http://www.example.com/api/user.xml
http://www.example.com/api/user.text
```

ColdBox has built-in support for detecting an extension in the URL and will save it into the request collection in a variable called format. What's even better is that `renderData()` can find the the format variable and automatically render your data in the appropriate way. All you need to do is pass in a list of valid rendering formats and `renderData()` will do the rest.

```
function index( event, rc, prc ) {
    var qUsers = getUserService().getUsers();
    // Correct format auto-detected from the URL
    event.renderData(data=qUsers, formats="json,xml,text");
}
```

Of course, if you want more control you can generate the output of your REST calls to your choosing and simply return a string directly from your handler that will be the output the client receives.

```
function index( event, rc, prc ) {
    return 'This is the result of my REST call';
}
```

STATUS CODES

Status codes are a core concept in HTTP and REST APIs use them to send messages back to the client. Here are a few sample REST status codes and messages.

- **200 OK** - Everything is hunky-dory
- **202 Created** - The resource was created successfully
- **400 Bad Request** - The server couldn't figure out what the client was sending it
- **401 Unauthorized** - The client isn't authorized to access this resource
- **404 Not Found** - The resource was not found
- **500 Server Error** - Something bad happened on the server

You can easily set status codes as well as the status message with `renderData()`. HTTP status codes and messages are not part of the response body. They live in the HTTP header.

```
function view( event, rc, prc ) {
    var qUser = getUserService().getUser( rc.userID );
    if ( qUser.recordCount ) {
        event.renderData( type="JSON", data=qUser );
    } else {
        event.renderData( type="JSON",
                          data={},
                          statusCode=404,
                          statusMessage="User not found");
    }
}

function save( event, rc, prc ) {
    // Save the user represented in the request body
    var userIDNew = userService.saveUser(
                                event.getHTTPContent() );
    // Return back the new userID to the client
    event.renderData( type="JSON",
                      data={ userID = userIDNew },
                      statusCode=201,
                      statusMessage="User created");
}
```

Status codes can also be set manually by using the `event.setHTTPHeader()` method in your handler.

```
function worldPeace( event, rc, prc ){
    event.setHTTPHeader( statusCode=501,
                         statusText="Not Implemented" );
    return "Try back later.";
}
```

CACHING

One of the great benefits of building your REST API on the ColdBox platform is tapping into great features such as event caching. Event caching allows you to cache the entire response for a resource using the incoming FORM and URL variables as the cache key. To enable event caching, set the following flag to true in your ColdBox config.

/config/ColdBox.cfc

```
coldbox.eventCaching = true;
```

Next, simply add the `cache=true` annotation to any action you want to be cached. That's it! You can also get fancy, and specify an optional `cacheTimeout` and `cacheLastAccesstimeout` (in minutes) to control how long to cache the data.

```
// Cache for default timeout
function showEntry( event, rc, prc ) cache="true" {
    prc.qEntry = getData();
    event.renderData( type="JSON", data=prc.qEntry );
}

/* Cache for one hour, or 20 minutes
   from the last time accessed. */
function showEntry( event, rc, prc ) cache="true"
             cacheTimeout="60" cacheLastAccessTimeout="20" {
    prc.qEntry = getData();
    event.renderData( type="JSON", data=prc.qEntry );
}
```

Data is stored in CacheBox's template cache. You can configure this cache to store its contents anywhere including an external clustered location such as Couchbase!

SECURITY

Adding authentication to an API is a common task and while there is no standard for REST, ColdBox supports just about anything you might need.

One of the simplest and easiest forms of authentication is Basic HTTP Auth. Note, this is not the most robust or secure method of authentication and most major APIs such as Twitter and FaceBook have all moved away from it. In Basic HTTP Auth, the client sends a header called **Authorization** that contains a base 64 encoded concatenation of the username and password.

You can easily get the username and password using event. `getHTTPBasicCredentials()`. Here is an example `preHandler()` method that enforces Basic HTTP Auth for all the actions in the handler.

```
function preHandler( event, action, eventArguments ){
    var auth = event.getHTTPBasicCredentials();
    if ( !securityService.authenticate( auth.username,
                                        auth.password ) ) {
        event.renderData(
            type="JSON",
            data={
                message="Please check your credentials"
            },
            statusCode=401,
            statusMessage="You're not authorized
                           to do that");
    }
}
```

CUSTOM SECURITY

You can also implement more customized solutions by tapping into any of the ColdBox interception points such as `preProcess` which is announced at the start of every request. Remember interceptors can include an `eventPattern` annotation to limit what ColdBox events they apply to. Below, our example interceptor will only fire if the ColdBox event starts with "**api**.".

In addition to having access to the entire request collection, the event object also has handy methods such as `event.getHTTPHeader()` to pull specific headers from the HTTP request.

This example will see if an HTTP header called "APIUser" is equal to the value "Honest Abe". If not, it will return the results of an error event.

/interceptors/APISecurity.cfc

```
/**
* This interceptor secures all API requests
*/
component {
    // This will only run when the event starts with "api."
    function preProcess( event, interceptData, buffer )
                eventPattern = '^api\.' {
        var APIUser = event.getHTTPHeader(
                                'APIUser', 'default' );

        // Only Honest Abe can access our API
        if ( APIUser != 'Honest Abe' ) {
            /* Every one else will get the
               error response from this event */
            event.overrideEvent('api.general.authFailed');
        }
    }
}
```

To register this interceptor with ColdBox, just add a line of config.

/config/ColdBox.cfc

```
interceptors = [
    {class='coldbox.system.interceptors.SES'},
    {class='interceptors.APISecurity'}
];
```

RESTRICT HTTP VERBS

In our route configuration we mapped HTTP verbs to handlers and actions, but what if users try to access resources directly with an invalid HTTP verb? You can enforce valid verbs (methods) by adding `this.allowedMethods` at the top of your handler. In this handler the `list()` method can only be access via a GET, and the `remove()` method can only be access via POST and DELETE.

```
component {
    this.allowedMethods = {
        remove = 'POST,DELETE',
        list   = 'GET'
    };
    function list( event, rc, prc ) {}
    function remove( event, rc, prc ) {}
}
```

The keys in the `allowedMethods` struct are the names of the actions and the values are a list of allowed HTTP methods. If the action is not listed in the structure, then it means allow all. Illegal calls will throw a 405 exception. You can catch this scenario and still return a properly-formatted response to your clients by using the `onError()` convention in your handler or an exception handler which applies to the entire app.

ERROR HANDLING

ColdBox REST APIs can use all the same error faculties that an ordinary ColdBox application has. We'll cover two of the most common ways here.

Handler onError()

If you create a method called `onError()` in a handler, ColdBox will automatically call that method for runtime errors that occur while executing any of the actions in that handler. This allows for localized error handling that is customized to that resource.

```
// error uniformity for resources
function onError( event, rc, prc, faultaction, exception ) {
    var thisMessage = '#faultaction# failed on resource:
                            #event.getCurrentRoutedURL()#';

    // log exception
    log.error( thisMessage, getHTTPRequestData() );

    // Standard return for all actions in this handler
    event.renderData(
        data={
            status = 'error',
            detail = thisMessage
        },
        type = 'json',
        statusCode = '500',
        statusMessage = thisMessage );
}
```

Global Exception Handler

The global exception handler will get called for any runtime errors that happen anywhere in the typical flow of your application. This is like the `onError()` convention but covers the entire application. First, configure the event you want called in the `ColdBox.cfc` config file. The event must have the handler plus action that you want called.

/config/ColdBox.cfc

```
coldbox = {
    ...
    exceptionHandler = 'main.onException'
    ...
}
```

Then create that action and put your exception handling code inside. You can choose to do error logging, notifications, or custom output here. You can even run other events.

/handlers/main.cfc

```
component {
    function onException( event, rc, prc ) {
        // Standard site-wide error handling here
    }
}
```

You can read more about ColdBox error handling in our docs here:
http://wiki.coldbox.org/wiki/ExceptionHandling.cfm

COLDBOX RELAX

ColdBox Relax is a set of **ReSTful Tools For Lazy Experts**. We pride ourselves in helping you (the developer) work smarter and ColdBox Relax is a tool to help you complete your REST projects faster. ColdBox Relax provides you with the necessary tools to quickly model, document and test your ReSTful services. One can think of ColdBox Relax as a way to describe ReSTful web services, test ReSTful web services, monitor ReSTful web services and document ReSTful web services–all while you relax!

You can read more about Relax in the official docs:
http://wiki.coldbox.org/wiki/Projects:Relax.cfm

The ColdBox Framework is completely documented in our online wiki located at http://wiki.coldbox.org/

Read more about ColdBox REST support here:
http://wiki.coldbox.org/wiki/Building_Rest_APIs.cfm

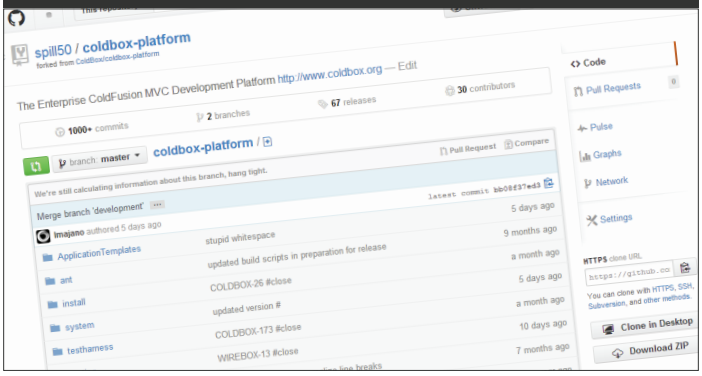
For API docs to see class definitions and method signatures, please visit the API docs located at http://www.coldbox.org/api

We have an active Google Group with hundreds of subscriber located at

http://groups.google.com/group/coldbox

Our official code repository is on GitHub. Please favorite us and feel free to fork and submit pull requests.

https://github.com/ColdBox/coldbox-platform



ColdBox REST APIs

Ortus Solutions
A Software Revolution

ColdBox is professional open source backed by Ortus Solutions, who provides training, support & mentoring, and custom development.

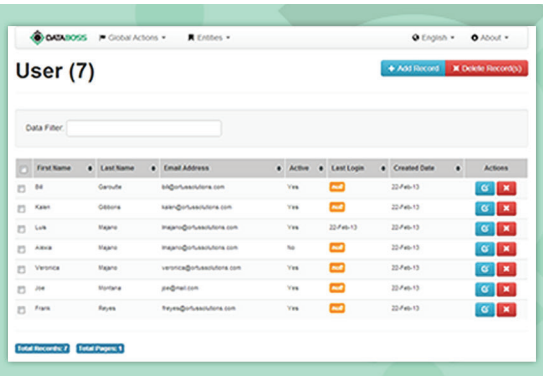
Support Program

The Ortus Support Program offers you a variety of Support Plans so that you can choose the one that best suit your needs. Subscribe to your plan now and join the Ortus Family!

With all of our plans you will profit not only from discounted rates but also receive an entire suite of support and development services. We can assist you with sanity checks, code analysis, architectural reviews, mentoring, professional support for all of our Ortus products, custom development and much more!

For more information visit
www.ortussolutions.com/services/support

Our Plans	M1	Entry	Standard	Premium	Enterprise
Price	**$199**	**$2099**	**$5699**	**$14099**	**$24599**
Support Hours	2 4 tickets	10 20 tickets	30 60 tickets	80 160 tickets	150 300 tickets
Discounted Rate	**$185/hr**	**$180/hr**	**$175/hr**	**$170/hr**	**$160/hr**
Renewal Price	**$199/month**	**$1800/year**	**$5250/year**	**$13600/year**	**$2400/year**
Phone/Online Appointments	✔	✔	✔	✔	✔
Web Ticketing System	✔	✔	✔	✔	✔
Architectural Reviews	✔	✔	✔	✔	✔
Hour Rollover		✔	✔	✔	✔
Custom Development		✔	✔	✔	✔
Custom Builds & Patches			✔	✔	✔
Priority Training Registration			✔	✔	✔
Development & Ticket Priority			✔	✔	✔
Response Times	1-5 B.D.	1-3 B.D.	1-2 B.D.	< 24 hr	< 12 hr
Books, Trainings, Product Discounts	0%	5%	10%	15%	20%
Free Books	0	0	1	3	5



"Build dynamic ORM administrators in seconds"

DATABOSS
Dynamic Administrator

www.data-boss.com

www.coldbox.org | www.ortussolutions.com

First Edition - April 2014