Steps to go from ColdBox Hero to SUPERHERO

(All commands assume we are in the `box` shell unless stated otherwise.)

Create App Folder

Create a new application folder where we will build our app on, call it `hcms` and place the given docker compose file in the root and the workbench folder as well.

```
hcms
 + docker-compose.yaml
 + workbench
```

Starting the Database

```
docker-compose up
```

Now open up your favorite SQL tool and make sure you can connect with the following credentials:

```
server: 127.0.0.1
port: 3307
database: cms
user: cms
password: coldbox
```

Your database should be online and populated with mock data.

> Tip: You can find the database export file here `/workbench/db/cms.sql`. You will also find the migrations we used and the seeder we used for populating the database under `/workbench/resources`

Global Dependencies

Before we start let's make sure we have our global CommandBox dependencies that we will use for environment control, cfconfig for CFML portability (cfconfig - https://cfconfig.ortusbooks.com/):

```
install commandbox-dotenv,commandbox-cfconfig,commandbox-cfformat
```

Creating our REST HMVC App

We will now begin creating our application using CommandBox. This will scaffold out a REST application using our `rest-hmvc` template. It will create a modular approach to our API based on ColdBox 6

The following dependencies will be installed for you:

- `coldbox` - Super HMVC Framework
- `testbox` - BDD testing library (`development` dependency)
- `modules/cbsecurity` - For securing your API
- `modules/cbvalidation` - For validating your API
- `modules/mementifier` - For marshalling objects to data
- `modules/relax` - Module for documenting, exploring and testing our API (`development` dependency)
    - `modules/cbSwagger` - Open API support for documenting our API
- `modules/route-visualizer` - For visualizing our routes

```
mkdir hcms --cd
coldbox create app name=cms skeleton=rest-hmvc
```

Let's go over what is in this template.

Updating our `.env` file

Update the environment file with the following information:

```
# ColdBox Environment
APPNAME=ColdBox
ENVIRONMENT=development

# Database Information
DB_CONNECTIONSTRING=jdbc:mysql://127.0.0.1:3307/cms?
useSSL=false&useUnicode=true&characterEncoding=UTF-
8&serverTimezone=UTC&useLegacyDatetimeCode=true
DB_CLASS=com.mysql.jdbc.Driver
DB_DRIVER=MySQL
DB_HOST=127.0.0.1
DB_PORT=3307
DB_DATABASE=cms
DB_USER=cms
DB_PASSWORD=coldbox

# JWT Information
JWT_SECRET=

# S3 Information
S3_ACCESS_KEY=
S3_SECRET_KEY=
S3_REGION=us-east-1
S3_DOMAIN=amazonaws.com
```

This will allow for CommandBox and the servers we run have access to these environment settings. Once you do changes on the `.env` file you will need to reload the shell or even the server for changes to take effect: `reload or server restart`

> We will fill out the JWT Secret later on.

CFConfig

Let's review our `.cfconfig.json` file.

```
{
    "requestTimeoutEnabled":true,
    "whitespaceManagement":"white-space-pref",
    "requestTimeout":"0,0,5,0",
    "cacheDefaultObject":"coldbox",
    "caches":{
        "coldbox":{
            "storage":"true",
            "type":"RAM",
            "custom":{
                "timeToIdleSeconds":"1800",
                "timeToLiveSeconds":"3600"
            },
            "class":"lucee.runtime.cache.ram.RamCache",
            "readOnly":"false"
        }
    },
    "datasources" : {
         "${DB_DATABASE}":{
             "host":"${DB_HOST}",
             "dbdriver":"${DB_DRIVER}",
             "database":"${DB_DATABASE}",
             "dsn":"jdbc:mysql://{host}:{port}/{database}",
             "custom":"useUnicode=true&characterEncoding=UTF-
8&useLegacyDatetimeCode=true&autoReconnect=true",
             "port":"${DB_PORT}",
             "class":"${DB_CLASS}",
             "username":"${DB_USER}",
             "password":"${DB_PASSWORD}",
             "connectionLimit":"100",
             "connectionTimeout":"1"
         }
    }
}
```

Automatic Formatting Goodness

Let's add some automatic formatting goodness. Open up your `box.json` and throw in this script:

```
"scripts":{
    "format:watch" : "cfformat watch
config,models,modules_app/api,tests/resources,tests/specs,*.cfc
./.cfformat.json"
},
```

Now you can run `run-script format:watch` and your code will format as you type.

Start it up!

Let's start our server up but let's configure it to use the `42518` port so we can all share URL's

```
server start port=42518
```

Boom! Our REST API is now online

> Tip: `server log --follow` to see the console logs of your server, try it out. All logging messages will also appear here as well. Great to have in a separate window.

BDD Setup and Test Harness

Let's also discover the automated tests that were created for us. Navigate to the following URL: http://localhost:42518/tests/runner.cfm and you will see the test results. Check out the specs in `tests/specs` as well.

Now let's try it via CommandBox CLI

```
testbox run "http://localhost:42518/tests/runner.cfm"
```

Now let's tell CommandBox about our runner.

```
package set testbox.runner="http://localhost:42518/tests/runner.cfm"
testbox run
```

Cool. Now we can test our stuff without leaving the editor. Let's make this even more cooler.

```
testbox watch
```

This will start a watcher so you can do your code changes and tests will be executed automatically for you. Go break a test and check it out.

> Hint: Type `<ctrl>+<c>` to exit out of watch mode

Application Modules

We will install several modules to assist us with the development of our API

- `qb` - Fluent query builder for fancy queries (https://forgebox.io/view/qb)
- `bcrypt` - To enable encrypting of passwords (https://www.forgebox.io/view/BCrypt)

```
install qb,bcrypt
coldbox reinit
```

Datasource Configuration

Open `Application.cfc` so we can add the global datasource we registered with the CFML Engine via `.cfconfig.json`

```
// App datasource
this.datasource = "cms";
```

Let's do the same with the tets application: `/tests/Application.cfc`

```
// App datasource
this.datasource = "cms";
```

Try running the app again. If it runs, it works. Or just issue a `testbox run`

Base Integration Spec Class

Let's create a base spec class all of our integration tests will inherit from. Place it under `tests/resources/BaseIntegrationSpec.cfc`. This resource will give us a few little reusable tidbits for testing:

- Only unload ColdBox once ALL tests has finalized
- Custom matcher for status codes (`toHaveStatus()`)
- Custom matcher for cbvalidation data (`toHaveInvalidData()`)
- A nice test reset method (`reset()`)

```
component extends="coldbox.system.testing.BaseTestCase" appMapping="/root"
{

    // Load on first test
    this.loadColdBox    = true;
    // Never unload until the request dies
    this.unloadColdBox  = false;

    /**
```

```
     * ------------------------------------------------------------------------
-------
     * Fixture Data
     * ------------------------------------------------------------------------
-------
     * Here is the global fixture data you can use
     */

    /*********************************** LIFE CYCLE Methods
***********************************/

    /**
     * Run Before all tests
     */
    function beforeAll() {
        super.beforeAll();
        // Wire up the test object with dependencies
        application.wirebox.autowire( this );

        // Add Custom Matchers
        addMatchers( {
            toHaveStatus : ( expectation, args = {} ) => {
                // handle both positional and named arguments
                param args.statusCode = "";
                if ( structKeyExists( args, 1 ) ) {
                    args.statusCode = args[ 1 ];
                }
                param args.message = "";
                if ( structKeyExists( args, 2 ) ) {
                    args.message = args[ 2 ];
                }
                if ( !len( args.statusCode ) ) {
                    expectation.message = "No status code provided.";
                    return false;
                }
                var statusCode = expectation.actual.getStatusCode();
                if ( statusCode != args.statusCode ) {
                    expectation.message = "#args.message#. Received
incorrect status code. Expected [#args.statusCode#]. Received
[#statusCode#].";
                    debug( expectation.actual.getMemento() );
                    return false;
                }
                return true;
            },
            // Verifies invalid cbValidation data
            toHaveInvalidData : ( expectation, args = {} ) => {
                param args.field = "";
                if ( structKeyExists( args, 1 ) ) {
                    args.field = args[ 1 ];
                }
                param args.error = "";
                if ( structKeyExists( args, 2 ) ) {
                    args.error = args[ 2 ];
```

```
                }
                param args.message = "";
                if ( structKeyExists( args, 3 ) ) {
                    args.message = args[ 3 ];
                }

                // If !400 then there is no invalid data
                if ( expectation.actual.getStatusCode() != 400 ) {
                    expectation.message = "#args.message#. Received
incorrect status code. Expected [400]. Received
[#expectation.actual.getStatusCode()#].";
                    debug( expectation.actual.getMemento() );
                    return false;
                }
                // If no field passed, we just check that invalid data was
found
                if ( !len( args.field ) ) {
                    if ( expectation.actual.getData().isEmpty() ) {
                        expectation.message = "#args.message#. Received
incorrect status code. Expected [400]. Received
[#expectation.actual.getStatusCode()#].";
                        debug( expectation.actual.getMemento() );
                        return false;
                    }
                    return true;
                }
                // We have a field to check and it has data
                if (
                    !structKeyExists(
                        expectation.actual.getData(),
                        args.field
                    ) || expectation.actual.getData()[ args.field
].isEmpty()
                ) {
                    expectation.message = "#args.message#. The requested
field [#args.field#] does not have any invalid data.";
                    debug( expectation.actual.getMemento() );
                    return false;
                }
                // Do we have any error messages to check?
                if ( len( args.error ) ) {
                    try {
                        expect(
                            expectation.actual.getData()[ args.field ]
                                .map( ( item ) => item.message )
                                .toList()
                        ).toInclude( args.error );
                    } catch ( any e ) {
                        debug( expectation.actual.getMemento() );
                        rethrow;
                    }
                }

                // We checked and it's all good!
```

```
                return true;
            }
        } );
    }

    function afterAll(){
        super.afterAll();
    }

    function reset(){
        application.cbController.getLoaderService().processShutdown();
        structDelete( application, "wirebox" );
        structDelete( application, "cbController" );
    }

    /**
     * This function is tagged as an around each handler.  All the
integration tests we build
     * will be automatically rolled backed. No database corruption
     *
     * @aroundEach
     */
    function wrapInTransaction( spec ) {
        transaction action="begin" {
            try {
                arguments.spec.body();
            } catch ( any e ){
                rethrow;
            } finally {
                transaction action="rollback";
            }
        }
    }

}
```

Update the `tests/specs/integration/EchoTests.cfc` to inherit from this spec and verify it works. It should look like this:

```
/**********************************************************************
******
*   Integration Test as BDD (CF10+ or Railo 4.1 Plus)
*
*   Extends the integration class: coldbox.system.testing.BaseTestCase
*
*   so you can test your ColdBox application headlessly. The 'appMapping'
points by default to
*   the '/root' mapping created in the test folder Application.cfc.
Please note that this
*   Application.cfc must mimic the real one in your root, including ORM
settings if needed.
```

```
*
*   The 'execute()' method is used to execute a ColdBox event, with the
following arguments
*   * event : the name of the event
*   * private : if the event is private or not
*   * prePostExempt : if the event needs to be exempt of pre post
interceptors
*   * eventArguments : The struct of args to pass to the event
*   * renderResults : Render back the results of the event
**************************************************************************
*****/
component extends="tests.resources.BaseIntegrationSpec"{

/*********************************** BDD SUITES
***********************************/

    function run(){

        describe( "My RESTFUl Service", function(){

            beforeEach(function( currentSpec ){
                // Setup as a new ColdBox request, VERY IMPORTANT. ELSE
EVERYTHING LOOKS LIKE THE SAME REQUEST.
                setup();
            });

            it( "can handle invalid HTTP Calls", function(){
                var event = execute( event="v1:echo.onInvalidHTTPMethod",
renderResults = true );
                var response = event.getPrivateValue( "response" );
                expect( response.getError() ).toBeTrue();
                expect( response.getStatusCode() ).toBe( 405 );
            });

            it( "can handle global exceptions", function(){
                var event = execute(
                    event           = "v1:echo.onError",
                    renderResults   = true,
                    eventArguments  = { exception={ message="unit test",
detail="unit test", stacktrace="" } }
                );

                var response = event.getPrivateValue( "response" );
                expect( response.getError() ).toBeTrue();
                expect( response.getStatusCode() ).toBe( 500 );
            });

            it( "can handle an echo", function(){
                var event       = this.request( "/api/v1/echo/index" );
                var response    = event.getPrivateValue( "response" );
                expect( response.getError() ).toBeFalse();
                expect( response.getData() ).toBe( "Welcome to my ColdBox
RESTFul Service" );
            });
```

```
            it( "can handle missing actions", function(){
                var event       = this.request( "/api/v1/echo/bogus" );
                var response    = event.getPrivateValue( "response" );
                expect( response.getError() ).tobeTrue();
                expect( response.getStatusCode() ).toBe( 405 );
            });


        });

    }

}
```

Security Configuration

We will start by configuring `cbsecurity` so we can secure our app and be able to provide Json Web Tokens (JWT) for securing our app. Once the configuration is done, we will move on to start the user registration process.

- Go over cbSecurity
- Go over cbAuth
- Go over JWT

Open the `config/ColdBox.cfc` and locate the `moduleSettings`, we will be adding the following configuration for cbauth, cbsecurity and jwt. Please go over each configured setting below:

```
moduleSettings = {

    cbauth = {
        // Which class will provide user information for authentication
        userServiceClass : "UserService"
    },

    cbsecurity = {
        // The global invalid authentication event or URI or URL to go if
an invalid authentication occurs
        "invalidAuthenticationEvent"    :
"v1:Echo.onAuthenticationFailure",
        // Default Auhtentication Action: override or redirect when a user
has not logged in
        "defaultAuthenticationAction"   : "override",
        // The global invalid authorization event or URI or URL to go if
an invalid authorization occurs
        "invalidAuthorizationEvent"     :
"v1:Echo.onAuthorizationFailure",
        // Default Authorization Action: override or redirect when a user
does not have enough permissions to access something
        "defaultAuthorizationAction"    : "override",
        // You can define your security rules here or externally via a
```

```
source
        "rules"                         : [
            // We will add them later
        ],
        // The validator is an object that will validate rules and
annotations and provide feedback on either authentication or authorization
issues.
        "validator"                     : "JWTService@cbsecurity",
        // WireBox ID of the user service to use
        "userService"                   : "UserService",
        // Activate security rule visualizer, defaults to false by default
        "enableSecurityVisualizer"      : true,
        // JWT Settings
        "jwt"                           : {
            // The jwt secret encoding key to use
            "secretKey"                 : getSystemSetting( "JWT_SECRET", ""
),
            // by default it uses the authorization bearer header, but you
can also pass a custom one as well or as an rc variable.
            "customAuthHeader"          : "x-auth-token",
            // The expiration in minutes for the jwt tokens
            "expiration"                : 60,
            // encryption algorithm to use, valid algorithms are: HS256,
HS384, and HS512
            "algorithm"                 : "HS512",
            // Which claims neds to be present on the jwt token or
`TokenInvalidException` upon verification and decoding
            "requiredClaims"            : [] ,
            // The token storage settings
            "tokenStorage"              : {
                // enable or not, default is true
                "enabled"       : true,
                // A cache key prefix to use when storing the tokens
                "keyPrefix"     : "cbjwt_",
                // The driver to use: db, cachebox or a WireBox ID
                "driver"        : "db",
                // Driver specific properties
                "properties"    : {
                    "table" : "cbjwt"
                }
            }
        }
    }
}
```

JWT Secret Key

Let's add the JWT Secret now, let's begin by generating a secret key. Go to CommandBox and let's generate one:

```
#generateSecretKey blowfish 256

>/TvWsL6k2Ap2/wbDroYmM9WT5JF/PndOdlzpJQEqUuI=
```

Please note that this is not necessary in the latest version of cbSecurity. cbSecurity will generate a new jwt secret each time the application expires and reloads. Which can also be nice to have. You decide the approach!

Copy the output of the key and paste it into the `.env` setting called `JWT_SECRET`

```
JWT_SECRET=/TvWsL6k2Ap2/wbDroYmM9WT5JF/PndOdlzpJQEqUuI=
```

Now we need to reinit our server since we added a new secret. Also, CommandBox CLI doesn't know about it, since you just added it, so let's reload the shell as well. Go to CommandBox:

```
// Reload the shell
reload

// restar the server
restart
```

Securfity Visualizer

Go to http://127.0.0.1:42518/cbsecurity and you can see the security visualizer. Super useful!

That's it! Make sure your tests work: `testbox run`

Registration

Let's focus now on the user registration requirement. This is the BDD story to complete:

```
story( "I want to be able to register users in my system and assign them a
JWT token upon registration" );
```

Let's start by modeling our user object, which our rest template has already pre-built for us: `models/Users.cfc`

`User.cfc`

Here are the properties we want to model, so make the appropriate changes in the pre-built user object.

- `id`
- `name`
- `email`
- `username`

- password
- createdDate:date
- modifiedDate:date

We will create the model object and a basic compile unit test:

```
coldbox create model name="User"
properties="id,name,email,username,password,createdDate:date,modifiedDate:
date"
```

Let's open the file and add some initialization procedures and a utility method to know if the User object has been populated from the database or not: `isLoaded()`. We will also add instructions for the `mementifier` module to know how to convert the object to JSON, and our validation constraints.

```
component accessors="true" {

    // DI
    property name="qb" inject="provider:QueryBuilder@qb";

    /**
     * ---------------------------------------------------------------------
-------
     * Properties
     * ---------------------------------------------------------------------
-------
     */

    // Properties
    property name="id"           type="string";
    property name="name"         type="string";
    property name="email"        type="string";
    property name="username"     type="string";
    property name="password"     type="string";
    property name="createdDate"  type="date";
    property name="modifiedDate" type="date";
    property name="permissions"  type="array";

    /**
     * ---------------------------------------------------------------------
-------
     * Mementifier
     * ---------------------------------------------------------------------
-------
     */
    this.memento = {
        defaultIncludes : [ "*" ],
        defaultExcludes : [],
        neverInclude    : [ "password" ]
    };
```

```
    /**
     * ------------------------------------------------------------------------
-------
     * Validation
     * ------------------------------------------------------------------------
-------
     */
    this.constraints = {
        name    : { required : true },
        email    : { required : true, type : "email" },
        username    : { required : true, udf : ( value, target ) => {
            if( isNull( arguments.value ) ) return false;
            return qb.from( "users" ).where( "username", arguments.value
).count() == 0;
        }},
        password    : { required : true }
    };

    /**
     * Constructor
     */
    function init() {
        variables.permissions = [];
        variables.createdDate = now();
        variables.modifiedDate = now();

        return this;
    }

    /**
     * ------------------------------------------------------------------------
-------
     * Authentication/Authorization Methods
     * ------------------------------------------------------------------------
-------
     */

    /**
     * Check if a user is loaded from the db or not
     */
    boolean function isLoaded() {
        return ( !isNull( variables.id ) && len( variables.id ) );
    }

    /**
     * Verify if the user has one or more of the passed in permissions
     *
     * @permission One or a list of permissions to check for access
     *
     */
    boolean function hasPermission( required permission ) {
        // If no permissions, then it a default value of true comes in
        if ( isBoolean( arguments.permission ) && arguments.permission ) {
            return true;
```

```
        }

        if ( isSimpleValue( arguments.permission ) ) {
            arguments.permission = listToArray( arguments.permission );
        }

        return arguments.permission
            .filter( function( item ) {
                return ( variables.permissions.findNoCase( item ) );
            } )
            .len();
    }

    /**
     * --------------------------------------------------------------------
-------
     * IJwtSubject Methods
     * --------------------------------------------------------------------
-------
     */

    /**
     * A struct of custom claims to add to the JWT token
     */
    struct function getJwtCustomClaims() {
        return {};
    }

    /**
     * This function returns an array of all the scopes that should be
attached to the JWT token that will be used for authorization.
     */
    array function getJwtScopes() {
        return [];
    }

}
```

Now open the unit test: `/tests/specs/unit/UserTest.cfc` and update it to this:

```
describe( "A User", function(){

    it( "can be created", function(){
        expect( model ).toBeComponent();
    });

});
```

Run your tests! We have broken a few things, but we will get them sorted out soon.

Authentication and JWTSubject

Since we will be using cbsecurity's authentication service (**cbauth**) and it's jwt services, let's make sure our User object adhers to those requirements by implementing the methods it asks us to do. We won't test them, since those will be tested by the integration portions.

`IAuthUser` - https://coldbox-security.ortusbooks.com/usage/authentication-services#user-interface

We will skip adding the `getId()` function since that is added already by the accessor. We don't have any permissions yet in the system, so we will always return true, and for `isLoggedIn()` we will delegate to cbauth (`authenticationService@cbauth`).

```
interface{

    /**
     * Return the unique identifier for the user
     */
    function getId();

    /**
     * Verify if the user has one or more of the passed in permissions
     *
     * @permission One or a list of permissions to check for access
     *
     */
    boolean function hasPermission( required permission );

    /**
     * Shortcut to verify it the user is logged in or not.
     */
    boolean function isLoggedIn();

}
```

`IJwtSubject` - https://coldbox-security.ortusbooks.com/jwt/jwt-services#jwt-subject-interface

For now we won't have any custom claims or custom security scopes. Maybe later on in our training we will add them.

```
interface{

    /**
     * A struct of custom claims to add to the JWT token
     */
    struct function getJwtCustomClaims();

    /**
     * This function returns an array of all the scopes that should be
attached to the JWT token that will be used for authorization.
     */
```

```
    array function getJwtScopes();

}
```

So our `User.cfc` will end up like this:

```
component accessors="true" {

    // DI
    property name="qb" inject="provider:QueryBuilder@qb";

    /**
     * ---------------------------------------------------------------------
-------
     * Properties
     * ---------------------------------------------------------------------
-------
     */

    // Properties
    property name="id"           type="string";
    property name="name"         type="string";
    property name="email"        type="string";
    property name="username"     type="string";
    property name="password"     type="string";
    property name="createdDate"  type="date";
    property name="modifiedDate" type="date";
    property name="permissions"  type="array";

    /**
     * ---------------------------------------------------------------------
-------
     * Mementifier
     * ---------------------------------------------------------------------
-------
     */
    this.memento = {
        defaultIncludes : [ "*" ],
        defaultExcludes : [],
        neverInclude    : [ "password" ]
    };

    /**
     * ---------------------------------------------------------------------
-------
     * Validation
     * ---------------------------------------------------------------------
-------
     */
    this.constraints = {
        name    : { required : true },
        email    : { required : true, type : "email" },
```

```
        username    : { required : true, udf : ( value, target ) => {
            if( isNull( arguments.value ) ) return false;
            return qb.from( "users" ).where( "username", arguments.value
).count() == 0;
        }},
        password    : { required : true }
    };

    /**
     * Constructor
     */
    function init() {
        variables.permissions = [];
        variables.createdDate = now();
        variables.modifiedDate = now();

        return this;
    }

    /**
     * --------------------------------------------------------------------
-------
     * Authentication/Authorization Methods
     * --------------------------------------------------------------------
-------
     */

    /**
     * Check if a user is loaded from the db or not
     */
    boolean function isLoaded() {
        return ( !isNull( variables.id ) && len( variables.id ) );
    }

    /**
     * Verify if the user has one or more of the passed in permissions
     *
     * @permission One or a list of permissions to check for access
     *
     */
    boolean function hasPermission( required permission ) {
        // If no permissions, then it a default value of true comes in
        if ( isBoolean( arguments.permission ) && arguments.permission ) {
            return true;
        }

        if ( isSimpleValue( arguments.permission ) ) {
            arguments.permission = listToArray( arguments.permission );
        }

        return arguments.permission
            .filter( function( item ) {
                return ( variables.permissions.findNoCase( item ) );
            } )
```

```
            .len();
    }

    /**
     * --------------------------------------------------------------------
-------
     * IJwtSubject Methods
     * --------------------------------------------------------------------
-------
     */

    /**
     * A struct of custom claims to add to the JWT token
     */
    struct function getJwtCustomClaims() {
        return {};
    }

    /**
     * This function returns an array of all the scopes that should be
attached to the JWT token that will be used for authorization.
     */
    array function getJwtScopes() {
        return [];
    }

}
```

BDD Integration

Now that our model is complete and satisfies the **cbsecurity** requirements for authentication and jwt services let's focus on the actual registration. We will create our BDD spec first to write down our requirements. We will then proceed to create the implementation.

Our template comes with a handler for authentication and an integration test as well:

- `modules_app/api/modules_app/v1/handlers/Auth.cfc`
- `tests/specs/integration/AuthTests.cfc`

Let's open our test and modify it a bit by making sure it inherits from our base class, remove some boilerplate and also update it for our new model. Also, let's comment out the stories pre-generated for us so we can focus only on the registration story (Use the `x` prefix)

It should end up like this:

```
component extends="tests.resources.BaseIntegrationSpec"{

    property name="jwtService" inject="provider:JwtService@cbsecurity";
    property name="cbauth"
inject="provider:authenticationService@cbauth";
```

```
    /*********************************** BDD SUITES
***********************************/

    function run() {
        describe( "RESTFul Authentication", function() {
            beforeEach( function( currentSpec ) {
                // Setup as a new ColdBox request, VERY IMPORTANT. ELSE
EVERYTHING LOOKS LIKE THE SAME REQUEST.
                setup();

                // Make sure nothing is logged in to start our calls
                cbauth.logout();
                jwtService.getTokenStorage().clearAll();
            } );

            xstory( "I want to authenticate a user and receive a JWT
token", function() {
                given( "a valid email and password", function() {
                    then( "I will be authenticated and will receive the
JWT token", function() {
                        // Use a user in the seeded db
                        var event = this.post(
                            route  = "/api/v1/login",
                            params = {
                                email    : "admin@coldbox.org",
                                password : "admin"
                            }
                        );
                        var response = event.getPrivateValue( "Response"
);
                        expect( response.getError() ).toBeFalse(
response.getMessages().toString() );
                        expect( response.getData() ).toBeString();

                        debug( response.getData() );

                        var decoded = jwtService.decode(
response.getData() );
                        expect( decoded.sub ).toBe( 1 );
                        expect( decoded.exp ).toBeGTE( dateAdd( "h", 1,
decoded.iat ) );
                    } );
                } );
                given( "invalid email and password", function() {
                    then( "I will receive a 401 exception ", function() {
                        var event = this.post(
                            route  = "/api/v1/login",
                            params = {
                                email    : "invalid",
                                password : "invalid"
                            }
                        );
                        var response = event.getPrivateValue( "Response"
);
```

```
                        expect( response.getError() ).toBeTrue();
                        expect( response.getStatusCode() ).toBe( 401 );
                    } );
                } );
            } );

            story( "I want to register into the system", function() {
                given( "valid registration details", function() {
                    then( "I should register, log in and get a token",
function() {
                        // Use a user in the seeded db
                        var event = this.post(
                            route  = "/api/v1/register",
                            params = {
                                name    : "luis Majano",
                                email    : "lmajano@coldbox.org",
                                username : "lmajano",
                                password : "lmajano"
                            }
                        );
                        var response = event.getPrivateValue( "Response"
);

                        debug( response.getData() );

                        expect( response ).toHaveStatus( 200 );
                        expect( response.getData() ).toHaveKey(
"token,user" );

                        var decoded = jwtService.decode(
response.getData().token );
                        expect( decoded.sub ).toBe(
response.getData().user.id );
                        expect( decoded.exp ).toBeGTE( dateAdd( "h", 1,
decoded.iat ) );
                    } );
                } );
                given( "invalid registration details", function() {
                    then( "I should get an error message", function() {
                        var event = this.post(
                            route  = "/api/v1/register",
                            params = {
                                email    : "invalid",
                                password : "invalid"
                            }
                        );
                        var response = event.getPrivateValue( "Response"
);
                        expect( event.getResponse() ).toHaveStatus( 400 );
                    } );
                } );
                xgiven( "valid registration data but with a non-unique
username", function(){
                    then( "a validation message should be sent to the user
```

```
with an error message", function(){

                    });
                } );
            } );

            xstory( "I want to be able to logout from the system using my
JWT token", function() {
                given( "a valid incoming jwt token", function() {
                    then( "my token should become invalidated and I will
be logged out", function() {
                        // Log in first to get a valid token to logout
with
                        var token   = jwtService.attempt(
"admin@coldbox.org", "admin" );
                        var payload = jwtService.decode( token );
                        expect( cbauth.isLoggedIn() ).toBeTrue();

                        // Now Logout
                        var event = this.post( route = "/api/v1/logout",
params = { "x-auth-token" : token } );

                        var response = event.getPrivateValue( "Response"
);
                        expect( response.getError() ).toBeFalse(
response.getMessages().toString() );
                        expect( response.getStatusCode() ).toBe( 200 );
                        expect( cbauth.isLoggedIn() ).toBeFalse();
                    } );
                } );
                given( "an invalid incoming jwt token", function() {
                    then( "I should see an error message", function() {
                        // Now Logout
                        var event = this.post( route = "/api/v1/logout",
params = { "x-auth-token" : "123" } );

                        var response = event.getPrivateValue( "Response"
);
                        expect( response.getError() ).toBeTrue(
response.getMessages().toString() );
                        debug( response.getStatusCode( 500 ) );
                    } );
                } );
            } );
        } );
    }

}
```

Run it, it will run but some things are failing now, since we where basically mocking things before. So we will be fixing each story one by one until our entire auth procedures are done with.

Routing

Open the `v1` module's router: `modules_app/api/modules_app/v1/config/Router.cfc` and add our routing or verify that we have all of our necessary routes:

```
component{

    function configure(){

        // API Echo
        get( "/", "Echo.index" );

        // API Authentication Routes
        post( "/login", "Auth.login" );
        post( "/logout", "Auth.logout" );
        post( "/register", "Auth.register" );

        // API Secured Routes
        get( "/whoami", "Echo.whoami" );

        route( "/:handler/:action" ).end();
    }

}
```

Issue a `coldbox reinit` and check out the Route Visualizer: http://127.0.0.1:42518/route-visualizer

Event Handler

Now that we have our route, let's fill out our event handler for registration only. Open the `auth.cfc` in our `v1` module and let's code it out. We will need to populate a new user object from incoming `rc` data, validate it, create it and then tell the cbsecurity's jwt services to create a token for the user. Also remember that our handler's MUST Inherit from our `coldbox.system.RestHandler`, which is new in ColdBox 6.

- Investigate RestHandler and what you get.

```
/**
 * Register a new user in the system
 *
 * @x-route (POST) /api/v1/register
 * @requestBody ~api/v1/auth/register/requestBody.json
 * @response-default ~api/v1/auth/register/responses.json##200
 * @response-400 ~api/v1/auth/register/responses.json##400
 */
function register( event, rc, prc ) {
    param rc.fname    = "";
    param rc.lname    = "";
    param rc.email    = "";
    param rc.password = "";
```

```
    // Populate, Validate, Create a new user
    prc.oUser = userService.create( validateOrFail( populateModel( "User"
) ) );

    // Log them in if it was created!
    event
        .getResponse()
        .setData( {
            "token" : jwtAuth().fromuser( prc.oUser ),
            "user"  : prc.oUser.getMemento()
        } )
        .addMessage(
            "User registered correctly and Bearer token created and it
expires in #jwtAuth().getSettings().jwt.expiration# minutes"
        );
}
```

Once we write up this code, two new scenarios should pop up in your head:

```
given( "invalid registration details", function() {
    then( "I should get an error message", function() {

    });
} );
given( "valid registration data but with a non-unique username",
function(){
    then( "a validation message should be sent to the user with an error
message", function(){

    });
} );
```

You will have to fill these out on your own! Let's do it!

User Services

Since we must use a `UserService` in our handler, then I guess we need to build it. How? Well, we know we need to add a `create()` method, but our template already generated one for us. It is using mocks, so let's update it so we can actually use the database.

> Also note that we have a unit test for the UserService: `tests/specs/unit/UserServiceTest.cfc`

Here is the unit test pre-generated for us, which is enough for us to continue upon. We are not building any hard-core algorithms or anything to test in isolation.

```
component extends="coldbox.system.testing.BaseTestCase" {

    /*********************************** LIFE CYCLE Methods
***********************************/
```

```
    function beforeAll() {
        structDelete( application, "cbController" );
        structDelete( application, "wirebox" );
        super.beforeAll();
    }

    function afterAll() {
        super.afterAll();
    }

    /*********************************** BDD SUITES
***********************************/

    function run() {
        describe( "UserService", function() {
            beforeEach( function( currentSpec ) {
                setup();
                model = getInstance( "UserService" );
            } );

            it( "can be created", function() {
                expect( model ).toBeComponent();
            } );

        } );
    }

}
```

Here is the code for the `UserService` using our new model approach and the methods we need to update for now:

- `init()`
- `create()`

```
/**
* User Services
*/
component accessors="true" singleton {

    /**
     * ---------------------------------------------------------------------
-------
     * DI
     * ---------------------------------------------------------------------
-------
     */
    property name="populator"   inject="wirebox:populator";
    property name="bcrypt"      inject="@BCrypt";
    property name="qb"          inject="provider:QueryBuilder@qb";
```

```
    /**
     * Constructor
     */
    function init() {
        return this;
    }

    /**
     * Construct a new user object via WireBox
     */
    User function new() provider="User";

    /**
     * Create a new user in the system
     *
     * @user The user to create
     *
     * @return The created user
     */
    User function create( required user ) {
        var qResults = qb.from( "users" )
            .insert( values = {
                "name"      = arguments.user.getName(),
                "email"     = arguments.user.getEmail(),
                "username"  = arguments.user.getUsername(),
                "password"  = bcrypt.hashPassword(
arguments.user.getPassword() )
            } );

        // populate the id
        arguments.user.setId( qResults.result.generatedKey );

        return arguments.user;
    }
}
```

Now let's verify our tests and adjust as necessary, but we should have a working registration now and jwt token creations. Also, check out your database, a new table `cbjwt` has been created for you!

Question, why doesn't the table have any data on it?

> Go into the base integration test and remove the `aroundEach` annotation from the `wrapInTransaction()` method, run the tests, and check out the database now. Run again, the data persists. So be careful, decide and move on. Clear the db data if you like.

Authentication

Now that we have registration complete, let's focus on authentication for our API. Here are two stories to start with which can be found already in our auth spec.

```
story( "I want to be able to authenticate with a username/password and
receive a JWT token", function(){

} );

story( "I want to be able to logout from the system using my JWT token",
function(){

} );
```

BDD

Let's start with our BDD specs before we move to the implementation phase. Let's open the `AuthTests` spec and revise our login/logout tests so they use our new model and matchers:

```
component extends="tests.resources.BaseIntegrationSpec" {

    property name="jwtService" inject="provider:JwtService@cbsecurity";
    property name="cbauth"
inject="provider:authenticationService@cbauth";

    /*********************************** BDD SUITES
***********************************/

    function run() {
        describe( "RESTFul Authentication", function() {
            beforeEach( function( currentSpec ) {
                // Setup as a new ColdBox request, VERY IMPORTANT. ELSE
EVERYTHING LOOKS LIKE THE SAME REQUEST.
                setup();

                // Make sure nothing is logged in to start our calls
                cbauth.logout();
                jwtService.getTokenStorage().clearAll();
            } );

            story( "I want to authenticate a user and receive a JWT
token", function() {
                given( "a valid username and password", function() {
                    then( "I will be authenticated and will receive the
JWT token", function() {
                        // Use a user in the seeded db
                        var event = this.post(
                            route  = "/api/v1/login",
                            params = {
                                username : "Milkshake10",
                                password : "test"
                            }
                        );
                        var response = event.getPrivateValue( "Response"
```

```
);
                        debug( response.getData() );

                        expect( response ).toHaveStatus( 200 );
                        expect( response.getData() ).toHaveKey(
"token,user" );

                        var decoded = jwtService.decode(
response.getData().token );
                        expect( decoded.sub ).toBe( 10 );
                        expect( decoded.exp ).toBeGTE( dateAdd( "h", 1,
decoded.iat ) );
                        expect( decoded.sub ).toBe(
response.getData().user.id );
                    } );
                } );
                given( "invalid email and password", function() {
                    then( "I will receive a 401 exception ", function() {
                        var event = this.post(
                            route  = "/api/v1/login",
                            params = {
                                email    : "invalid",
                                password : "invalid"
                            }
                        );
                        var response = event.getPrivateValue( "Response"
);
                        expect( response ).toHaveStatus( 401 );
                    } );
                } );
            } );

            story( "I want to register into the system", function() {
                given( "valid registration details", function() {
                    then( "I should register, log in and get a token",
function() {
                        // Use a user in the seeded db
                        var event = this.post(
                            route  = "/api/v1/register",
                            params = {
                                name     : "luis Majano",
                                email    : "lmajano@coldbox.org",
                                username : "lmajano",
                                password : "lmajano"
                            }
                        );
                        var response = event.getPrivateValue( "Response"
);

                        debug( response.getData() );

                        expect( response ).toHaveStatus( 200 );
                        expect( response.getData() ).toHaveKey(
"token,user" );
```

```
                        var decoded = jwtService.decode(
response.getData().token );
                        expect( decoded.sub ).toBe(
response.getData().user.id );
                        expect( decoded.exp ).toBeGTE( dateAdd( "h", 1,
decoded.iat ) );
                    } );
                } );
                given( "invalid registration details", function() {
                    then( "I should get an error message", function() {
                        var event = this.post(
                            route  = "/api/v1/register",
                            params = {
                                email    : "invalid",
                                password : "invalid"
                            }
                        );
                        var response = event.getPrivateValue( "Response"
);
                        expect( event.getResponse() ).toHaveStatus( 400 );
                    } );
                } );
                xgiven( "valid registration data but with a non-unique
username", function() {
                    then( "a validation message should be sent to the user
with an error message", function() {
                    } );
                } );
            } );

            story( "I want to be able to logout from the system using my
JWT token", function() {
                given( "a valid incoming jwt token", function() {
                    then( "my token should become invalidated and I will
be logged out", function() {
                        // Log in first to get a valid token to logout
with
                        var token   = jwtService.attempt( "Milkshake10",
"test" );
                        var payload = jwtService.decode( token );
                        expect( cbauth.isLoggedIn() ).toBeTrue();

                        // Now Logout
                        var event = this.post( route = "/api/v1/logout",
params = { "x-auth-token" : token } );

                        var response = event.getPrivateValue( "Response"
);
                        expect( response ).toHaveStatus( 200 );
                        expect( cbauth.isLoggedIn() ).toBeFalse();
                    } );
                } );
                given( "an invalid incoming jwt token", function() {
```

```
                    then( "I should see an error message", function() {
                        // Now Logout
                        var event = this.post( route = "/api/v1/logout",
params = { "x-auth-token" : "123" } );

                        var response = event.getPrivateValue( "Response"
);
                        expect( response.getError() ).toBeTrue(
response.getMessages().toString() );
                        expect( response ).toHaveStatus( 500 );
                    } );
                } );
            } );
        } );
    }

}
```

Routing

Open the `v1` module's router: `modules_app/api/modules_app/v1/config/Router.cfc` and make sure our login/logout routes are there:

```
component {

    function configure() {
        // API Echo
        get( "/", "Echo.index" );

        // API Authentication Routes
        post( "/login", "Auth.login" );
        post( "/logout", "Auth.logout" );
        post( "/register", "Auth.register" );

        // API Secured Routes
        get( "/whoami", "Echo.whoami" );


        route( "/:handler/:action" ).end();
    }

}
```

Issue a `coldbox reinit` and check out the Route Visualizer: http://127.0.0.1:42518/route-visualizer

Event Handler

Let's go back to our `auth` handler and finalize the login/logout actions.

```
/**
 * Authentication Handler
 */
component extends="coldbox.system.RestHandler" {

    // Injection
    property name="userService" inject="UserService";

    /**
     * Login a user into the application
     *
     * @x-route (POST) /api/v1/login
     * @requestBody ~api/v1/auth/login/requestBody.json
     * @response-default ~api/v1/auth/login/responses.json##200
     * @response-401 ~api/v1/auth/login/responses.json##401
     */
    function login( event, rc, prc ) {
        param rc.username = "";
        param rc.password = "";

        prc.token = jwtAuth().attempt( rc.username, rc.password );

        prc.response
            .setData( {
                "token" : prc.token,
                "user"  : cbSecure()
                    .getAuthService()
                    .getUser()
                    .getMemento()
            } )
            .addMessage( "Bearer token created and it expires in
#jwtAuth().getSettings().jwt.expiration# minutes" );
    }

    /**
     * Register a new user in the system
     *
     * @x-route (POST) /api/v1/register
     * @requestBody ~api/v1/auth/register/requestBody.json
     * @response-default ~api/v1/auth/register/responses.json##200
     * @response-400 ~api/v1/auth/register/responses.json##400
     */
    function register( event, rc, prc ) {
        param rc.fname    = "";
        param rc.lname    = "";
        param rc.email    = "";
        param rc.password = "";

        // Populate, Validate, Create a new user
        prc.oUser = userService.create( validateOrFail( populateModel(
"User" ) ) );

        // Log them in if it was created!
```

```
        event
            .getResponse()
            .setData( {
                "token" : jwtAuth().fromuser( prc.oUser ),
                "user"  : prc.oUser.getMemento()
            } )
            .addMessage(
                "User registered correctly and Bearer token created and it
expires in #jwtAuth().getSettings().jwt.expiration# minutes"
            );
    }

    /**
     * Logout a user
     *
     * @x-route (POST) /api/v1/logout
     * @security bearerAuth,ApiKeyAuth
     * @response-default ~api/v1/auth/logout/responses.json##200
     * @response-500 ~api/v1/auth/logout/responses.json##500
     */
    function logout( event, rc, prc ) {
        jwtAuth().logout();
        event.getResponse().addMessage( "Successfully logged out" )
    }

}
```

Wow, our handlers look so nice and tidy and with strange documentation! However, we still need to build out our User Service that will power all this goodness.

Please check out all of the jwt service methods, there are tons of them and really helpful!

https://coldbox-security.ortusbooks.com/jwt/jwt-services

UserService

In order for the jwt services and cbauth can authenticate and create tokens for us, we must adhere to the following interface (https://coldbox-security.ortusbooks.com/usage/authentication-services#user-services). This is needed so the calls in our handlers can work correctly as the cbauth and jwt services will be calling our user services and leveraging our User object.

```
interface{

    /**
     * Verify if the incoming username/password are valid credentials.
     *
     * @username The username
     * @password The password
     */
    boolean function isValidCredentials( required username, required
password );
```

```
    /**
     * Retrieve a user by username
     *
     * @return User that implements JWTSubject and/or IAuthUser
     */
    function retrieveUserByUsername( required username );

    /**
     * Retrieve a user by unique identifier
     *
     * @id The unique identifier
     *
     * @return User that implements JWTSubject and/or IAuthUser
     */
    function retrieveUserById( required id );
}
```

That's it. The jwt services and cbauth will know how to put everything together for you. So let's build the service out.

```
/**
 * User Services
 */
component accessors="true" singleton {

    /**
     * ---------------------------------------------------------------------
-------
     * DI
     * ---------------------------------------------------------------------
-------
     */
    property name="populator" inject="wirebox:populator";
    property name="bcrypt"    inject="@BCrypt";
    property name="qb"        inject="provider:QueryBuilder@qb";

    /**
     * Constructor
     */
    function init() {
        return this;
    }

    /**
     * Construct a new user object via WireBox
     */
    User function new() provider="User";

    /**
     * Create a new user in the system
```

```
     *
     * @user The user to create
     *
     * @return The created user
     */
    User function create( required user ) {
        var qResults = qb
            .from( "users" )
            .insert(
                values = {
                    "name"     : arguments.user.getName(),
                    "email"    : arguments.user.getEmail(),
                    "username" : arguments.user.getUsername(),
                    "password" : bcrypt.hashPassword(
arguments.user.getPassword() )
                }
            );

        // populate the id
        arguments.user.setId( qResults.result.generatedKey );

        return arguments.user;
    }

    /**
     * Verify if the incoming username/password are valid credentials.
     *
     * @username The username
     * @password The password
     */
    boolean function isValidCredentials( required username, required
password ) {
        var oTarget = retrieveUserByUsername( arguments.username );
        if ( !oTarget.isLoaded() ) {
            return false;
        }

        // Check Password Here: Remember to use bcrypt
        try {
            return variables.bcrypt.checkPassword( arguments.password,
oTarget.getPassword() );
        } catch ( any e ) {
            return false;
        }
    }

    /**
     * Retrieve a user by username
     *
     * @return User that implements JWTSubject and/or IAuthUser
     */
    function retrieveUserByUsername( required username ) {
        return populator.populateFromStruct(
            new (),
```

```
            qb.from( "users" )
                .where( "username", arguments.username )
                .first()
        );
    }

    /**
     * Retrieve a user by unique identifier
     *
     * @id The unique identifier
     *
     * @return User that implements JWTSubject and/or IAuthUser
     */
    User function retrieveUserById( required id ) {
        return populator.populateFromStruct(
            new (),
            qb.from( "users" )
                .where( "id", arguments.id )
                .first()
        );
    }

}
```

That's it! Go run your tests and make sure all the tests pass! What have you learned?

Invalid Routes

Ok, so what would happen if we try to execute `/api/v1/bogus` in the browser? Go and try it!

You will see that the browser blows up with a nasty invalid event. actually, ANY route we try to execute in the `v1` api will fail like this and this is not nice. We want uniformity, so let's add a catch all route that issues the Rest Handler's `onInvalidRoute()` method.

Open the `v1` router and add the invalid routes catch all before the default route and actually remove the default route as we won't be using it.

```
// Invalid Routes
route( "/:anything", "echo.onInvalidRoute" );

//route( "/:handler/:action" ).end();
```

Issue a nice `coldbox reinit` and hit the route again or any invalid route and you should see a nice API return 404 message. Now, this is great, but we lost something? Anybody can guess?

> We lost all of our convention based routing which was below it. This means, that we must register all the routes we want.

Swagger

Ok, before we go any further building stuff out, let's go over the weird documentation in our handlers. Go open one and try to decipher it? We are using cbswagger directives so we can document our API. Go to http://127.0.0.1:42518/cbswagger and see the json created for you. Copy it and go to https://editor.swagger.io/ and paste it in.

WOW! Our API is fully documented! What magic unicorn is this!

- https://www.forgebox.io/view/cbswagger
- https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md

Customize It

Open your `config/ColdBox.cfc` and under the `moduleSettings` there is a `cbSwagger` section. Let's update it a bit to match what we are building:

```
cbswagger : {
    // The route prefix to search.  Routes beginning with this prefix will
be determined to be api routes
    "routes"        : [ "api" ],
    // Any routes to exclude
    "excludeRoutes" : [ "api/v1/:anything/" ],
    // The default output format: json or yml
    "defaultFormat" : "json",
    // A convention route, relative to your app root, where
request/response samples are stored ( e.g.
resources/apidocs/responses/[module].[handler].[action].[HTTP Status
Code].json )
    "samplesPath"   : "resources/apidocs",
    // Information about your API
    "info"          : {
        // A title for your API
        "title"       : "Hero to SuperHero Headless CMS",
        // A description of your API
        "description" : "A nice hmvc headless CMS",
        // The contact email address
        "contact"     : {
            "name"  : "API Support",
            "url"   : "https://www.swagger.io/support",
            "email" : "info@ortussolutions.com"
        },
        // A url to the License of your API
        "license" : {
            "name" : "Apache 2.0",
            "url"  : "https://www.apache.org/licenses/LICENSE-2.0.html"
        },
        // The version of your API
        "version" : "1.0.0"
    },
    // Tags
    "tags"         : [ ],
    // https://swagger.io/specification/#externalDocumentationObject
    "externalDocs" : {
```

```
        "description" : "Find more info here",
        "url"         : "https://blog.readme.io/an-example-filled-guide-
to-swagger-3-2/"
    },
    // https://swagger.io/specification/#serverObject
    "servers" : [
        {
            "url"         : "https://mysite.com/v1",
            "description" : "The main production server"
        },
        {
            "url"         : "http://127.0.0.1:42518",
            "description" : "The dev server"
        }
    ],
    // An element to hold various schemas for the specification.
    // https://github.com/OAI/OpenAPI-
Specification/blob/master/versions/3.0.0.md#componentsObject
    "components" : {
        // Define your security schemes here
        // https://github.com/OAI/OpenAPI-
Specification/blob/master/versions/3.0.0.md#securitySchemeObject
        "securitySchemes" : {
            "ApiKeyAuth" : {
                "type"        : "apiKey",
                "description" : "User your JWT as an Api Key for
security",
                "name"        : "x-api-key",
                "in"          : "header"
            },
            "bearerAuth" : {
                "type"         : "http",
                "scheme"       : "bearer",
                "bearerFormat" : "JWT"
            }
        }
    }

    // A default declaration of Security Requirement Objects to be used
across the API.
    // https://github.com/OAI/OpenAPI-
Specification/blob/master/versions/3.0.0.md#securityRequirementObject
    // Only one of these requirements needs to be satisfied to authorize a
request.
    // Individual operations may set their own requirements with
`@security`
    // "security" : [
    //  { "APIKey" : [] },
    //  { "UserSecurity" : [] }
    // ]
},
```

Resources

You can also find all the resources we generated previously from our app template in the `resources/apidocs` folder. You will find here global schemas, and our routing by convention. Let's explore them and update them accordingly since our model changed.

- Update all resources to match our new API structure.

Fix our Error on SwaggerHub

You might see an error on top specifying the `{anything}` route we added. In all reality, this is an internal route, so we don't need it documented, so open your `config/Coldbox.cfc` and look for the `excludedRoutes` in the `cbSwagger` module settings:

```
// Any routes to exclude
"excludeRoutes" : [ "api/v1/:anything/"],
```

Regenerate the swagger doc and we have our internal routing removed! Voila!

PostMan Automation

Let's do one more mystical trick. Copy the swagger json and open Postman. Look for the import and import our cbwagger.

WOW! We know have imported all of our API into Postman for testing and even more sweet documentation!

Listing Content

Ok, we have all the building blocks for now focusing on our first content stories:

```
story( "In order to interact with content in the CMS you must be
authenticated" );
story( "I want to see content with different filtering options" )
story( "I want to see a single content object via a nice slug" )
```

Ok, let's start by modeling our content object

Content.cfc

We will be creating a `Content.cfc` that will store our headless content:

- `id`
- `slug`
- `title`
- `body`
- `isPublished:boolean`
- `publishedDate:date`

- `createdDate:date`
- `modifiedDate:date`
- `user (many to one)`

Ok, let's creat it via CommandBox

```
coldbox create model name="Content"
properties="id,slug,title,body,isPublished:boolean,publishedDate:date,crea
tedDate:date,modifiedDate:date,FK_userID,user"
```

Open up the object and the companion unit test. Remember in our unit test, we just want quick verifications.

- Initialize the content object
- Add an `isLoaded()` to verify persistence
- Add a `getUser()` to retrieve the relationship, so we will need to inject the `UserService`
- Add the mementifier instructions
- Add the validation constraints

```
/**
* I am a new Model Object
*/
component accessors="true"{

    // inject the user service
    property name="userService" inject="UserService";

    // Properties
    property name="id"            type="string";
    property name="slug"          type="string";
    property name="title"         type="string";
    property name="body"          type="string";
    property name="isPublished"   type="boolean";
    property name="publishedDate" type="date";
    property name="createdDate"   type="date";
    property name="modifiedDate"  type="date";
    property name="FK_userID"     type="string" default="";

    this.constraints = {
        slug        : { required : true, udf : ( value, target ) => {
            if( isNull( arguments.value ) ) return false;
            return qb.from( "content" ).where( "slug", arguments.value
).count() == 0;
        }},
        title       : { required : true },
        body        : { required : true },
        FK_userID   : { required : true }
    };

    this.memento = {
        defaultIncludes = [
```

```
            "slug",
            "title",
            "body",
            "isPublished",
            "publishedDate",
            "createdDate",
            "modifiedDate",
            "user.name",
            "user.email"
        ],
        defaultExcludes = [
            "FK_userID",
            "user.id",
            "user.username",
            "user.modifiedDate",
            "user.createdDate"
        ]
    };

    /**
     * Constructor
     */
    Content function init(){
        variables.createdDate   = now();
        variables.modifiedDate  = now();
        variables.isPublished   = false;
        variables.FK_userID     = "";
        return this;
    }

    boolean function isLoaded(){
        return ( !isNull( variables.id ) && len( variables.id ) );
    }

    User function getUser(){
        return variables.userService.retrieveUserById( variables.FK_userId
);
    }

    Content function setUser( required user ){
        if( user.isLoaded() ){
            variables.FK_userId = arguments.user.getId();
        }
        return this;
    }

}
```

Update your tests:

```
describe( "Content Object", function(){
```

```
        it( "can be created", function(){
            expect( model ).toBeComponent();
        });

    });
```

Verify we can compile by running your tests!

BDD

Now that we have our model let's start with the stories and integration. We can create a nice ColdBox resource for our content: `resources( "content" )` and it will provide us with the following:

- `GET:content.index` : Display all content objects
- `POST:content.create` : Create a content object
- `GET:content.show` : Display a single content
- `PUT/PATCH:content.update` : Update a content object
- `DELETE:content.delete` : Remove a content object

```
coldbox create handler name="content"
actions="index,create,show,update,delete"
directory=modules_app/api/modules_app/v1/handlers
```

Let's open up the specs and start building it out:

```
describe( "Content Services: In order to interact with content in the CMS
you must be authenticated", function(){

    beforeEach(function( currentSpec ){
        // Setup as a new ColdBox request for this suite, VERY IMPORTANT.
ELSE EVERYTHING LOOKS LIKE THE SAME REQUEST.
        setup();
        // Need to login
        jwt = jwtService.attempt( "Milkshake10", "test" );
        getRequestContext().setValue( "x-auth-token", jwt );
    });

    story( "I want to be able to see content with different options",
function(){
        it( "should display all content using the default options",
function(){
            var event = get( route = "/api/v1/content" );
            var response = event.getPrivateValue( "Response" );
            expect( response ).toHaveStatus( 200 );
            expect( response.getData() ).toBeArray();
        });
    });

    story( "I want to see a single content object via a nice slug",
```

```
function(){
        given( "a valid slug", function(){
            then( "I should be able to display the content object",
function(){
                var testSlug = "Spoon-School-Staircase";
                var event = get( route = "/api/v1/content/#testSlug#" );
                var response = event.getPrivateValue( "Response" );

                debug( response.getMessages() );

                expect( response ).toHaveStatus( 200 );
                expect( response.getData() ).toBeStruct();
                expect( response.getData().slug ).toBe( testSlug );
            });
        });

        given( "an invalid slug", function(){
            then( "then we should get a 404", function(){
                var testSlug = "invalid-bogus-object";
                var event = get( route = "/api/v1/content/#testSlug#" );
                var response = event.getPrivateValue( "Response" );

                debug( response.getMessages() );

                expect( response ).toHaveStatus( 404 );
            });
        });
    });

} );
```

Security

Now that we have our handler generated, we will secure it using a rule. Open the `config/ColdBox.cfc` and add the following rule to the `cbsecurity.rules` array:

```
{
    secureList  : "v1:content"
}
```

That's it! Now any requests made to that secure pattern will be inspected by the JWT Validator and a bearer token must be valid to access it! BOOM!

You can also secure using annotations, we can get rid of the rule and then in our handler we can add the `secured` annotation to the `component` definition. Same Approach.

Routing

Let's add our routing as we explained above in our `v1` router.

```
resources( resource="content", parameterName="slug" );
```

Please note that we change the parameter name to `slug` since we will use those unique slugs for operation and not the Id.

Event Handler

Now let's build out the handler that can satisfy our previous stories

```
/**
* I am a new handler
*/
component extends="coldbox.system.RestHandler" {

    property name="contentService" inject="ContentService";

    /**
     * index
     */
    function index( event, rc, prc ) {
        prc.response.setData(
            contentService
                .list()
                .map( ( item ) => {
                    return item.getMemento();
                } )
        );
    }

    /**
    * create
    */
    function create( event, rc, prc ){
        event.setView( "content/create" );
    }

    /**
     * show
     */
    function show( event, rc, prc ) {
        param rc.slug = "";

        prc.oContent = contentService.findBySlug( rc.slug );

        if ( !prc.oContent.isLoaded() ) {
            prc.response
                .setError( true )
                .setStatusCode( event.STATUS.NOT_FOUND )
                .setStatusText( "Not Found" )
                .addMessage( "The requested content object (#rc.slug#)
```

```
could not be found" );
            return;
        }

        prc.response.setData( prc.oContent.getMemento() );
    }

    /**
    * update
    */
    function update( event, rc, prc ){
        event.setView( "content/update" );
    }

    /**
    * delete
    */
    function delete( event, rc, prc ){
        event.setView( "content/delete" );
    }

}
```

Content Services

Ok, now we need to focus on our Content Services that will power the handler since we already created the Content object, so we need to implement the `findBySlug()` and the `list()` methods:

Let's generate what we need:

```
coldbox create model name="ContentService" persistence="singleton"
methods="list,get,findBySlug"
```

Also open the unit test and do a quick compile test:

```
describe( "ContentService Suite", function(){

    it( "can be created", function(){
        expect( model ).toBeComponent();
    });

});
```

Now let's build it out:

```
/**
* I am a new Model Object
```

```
*/
component singleton accessors="true"{

    // Properties
    property name="populator"   inject="wirebox:populator";
    property name="qb"          inject="provider:QueryBuilder@qb";


    /**
     * Constructor
     */
    ContentService function init(){
        return this;
    }

    Content function new() provider="Content";

    /**
    * list
    */
    array function list( orderBy="publishedDate", orderType="asc" ){
        return qb
            .from( "content" )
            .orderBy( arguments.orderBy, arguments.orderType )
            .get()
            .map( ( content ) => {
                return populator.populateFromStruct(
                    new(),
                    content
                );
            } );
    }

    /**
    * get
    */
    function get( required id ){
        return populator.populateFromStruct(
            new(),
            qb.from( "content" ).where( "id" , arguments.id ).first()
        );
    }

    /**
    * Find by slug
    */
    function findBySlug( required slug ){
        return populator.populateFromStruct(
            new(),
            qb.from( "content" ).where( "slug" , arguments.slug ).first()
        );
    }
```

```
}
```

Ok, it seems we are done, let's run our tests and make sure we are listing all content and getting a single content.

Extra Credit: Leverage postman to test these endpoints. Remember you must get a jwt token first!

Creating Content

Ok, we can list all and one piece of content, let's try creating one now.

```
story( "I want to be able to create content objects" )
```

I don't have to do any more setup for security, resources or even handlers. We have our resourceful handler already. So let's delve into the BDD first.

BDD

Update the spec with a new story and scenarios:`

```
story( "I want to be able to create new content objects", function(){
    given( "valid incoming data", function(){
        then( "it should create a new content object", function(){
            var event = post(
                route = "/api/v1/content",
                params = {
                    slug          : "my-new-test-#createUUID()#",
                    title         : "I love BDD",
                    body          : "I love BDD soooooooooo much!",
                    isPublished   : true,
                    publishedDate : now()
                }
            )

            // expectations go here.
            var response = event.getPrivateValue( "Response" );

            debug( response.getData() );

            expect( response ).toHaveStatus( 200 );
            expect( response.getData().title ).toBe( "I love BDD" );
            expect( response.getData().id ).notToBeEmpty();
        });
    });

    given( "invalid data", function(){
        then( "it should throw a validation error", function(){
            var event = post(
```

```
                route = "/api/v1/content",
                params = {
                    body          : "I love BDD soooooooooo much!",
                    isPublished   : true,
                    publishedDate : now()
                }
            )

            // expectations go here.
            var response = event.getPrivateValue( "Response" );

            expect( response ).toHaveStatus( 400 );
            expect( response ).toHaveInvalidData( "slug", "is required" );
        });
    });
});
```

Ok, now let's put it together!

Create Action

```
/**
* create
*/
function create( event, rc, prc ){

    // populate, validate and create
    prc.oContent = contentService.create(
        validateOrFail( populateModel( "Content" ).setUser(
jwtAuth().getUser() ) )
    );

    prc.response.setData( prc.oContent.getMemento() );
}
```

We have to also get the authenticated user to add it into the content.

Create Services

Now to the ugly (funky) SQL

```
/**
* create
*/
function create( required content ){
    var qResults = qb.from( "content" )
        .insert( values = {
            "slug"          = arguments.content.getSlug(),
            "title"         = arguments.content.getTitle(),
```

```
            "body"          = arguments.content.getBody(),
            "isPublished"   = { value :
arguments.content.getIsPublished(), cfsqltype : "tinyint" },
            "publishedDate" = { value :
arguments.content.getPublishedDate(), cfsqltype : "timestamp" },
            "createdDate"   = { value : now(), cfsqltype : "timestamp" },
            "modifiedDate"  = { value : now(), cfsqltype : "timestamp" },
            "FK_userId"     = arguments.content.getUser().getId()
        } );

    // populate the id
    arguments.content.setId( qResults.result.generatedKey );

    return arguments.content;
}
```

Run your tests, validate and BOOM! Creation done! Next!

Updating Content

```
story( "I want to be able to update content objects" )
```

I don't have to do any more setup for security, resources or even handlers. We have our resourceful handler already. So let's delve into the BDD first.

BDD

Update the spec with a new story and scenarios:`

```
story( "I want to be able to update content objects", function(){
    given( "valid incoming data", function(){
        then( "it should update the content object", function(){
            var event = put(
                route = "/api/v1/content/Record-Slave-Crystal",
                params = {
                    title         : "I just changed you!",
                    body          : "I love BDD soooooooooo much!",
                    isPublished   : false
                }
            )

            // expectations go here.
            var response = event.getPrivateValue( "Response" );

            debug( response.getData() );

            expect( response ).toHaveStatus( 200 );
            expect( response.getData().title ).toBe( "I just changed you!"
);
```

```
            expect( response.getData().id ).notToBeEmpty();
        });
    });

    given( "an invalid slug", function(){
        then( "it should throw a validation error", function(){
            var event = put(
                route = "/api/v1/content/bogus",
                params = {
                    body          : "I love BDD sooooooooooo much!",
                    isPublished   : true,
                    publishedDate : now()
                }
            )

            // expectations go here.
            var response = event.getPrivateValue( "Response" );

            expect( response ).toHaveStatus( 404 );
        });
    });
});
```

Ok, now let's put it together!

Update Action

```
/**
 * update
 */
function update( event, rc, prc ) {
    param rc.slug = "";

    prc.oContent = contentService.findBySlug( rc.slug );

    if ( !prc.oContent.isLoaded() ) {
        prc.response
            .setError( true )
            .setStatusCode( event.STATUS.NOT_FOUND )
            .setStatusText( "Not Found" )
            .addMessage( "The requested content object (#rc.slug#) could
not be found" );
        return;
    }

    // populate, validate and create
    prc.oContent = contentService.update(
        validateOrFail( populateModel( prc.oContent ).setUser(
jwtAuth().getUser() ) )
    );
```

```
    prc.response.setData( prc.oContent.getMemento() );
}
```

We have to also get the authenticated user to add it into the content.

Update Services

Now to the ugly (funky) SQL

```
/**
* update
*/
function update( required content ){
    var qResults = qb.from( "content" )
        .whereId( arguments.content.getId() )
        .update( {
            "slug"          = arguments.content.getSlug(),
            "title"         = arguments.content.getTitle(),
            "body"          = arguments.content.getBody(),
            "isPublished"   = { value :
arguments.content.getIsPublished(), cfsqltype : "tinyint" },
            "publishedDate" = { value :
arguments.content.getPublishedDate(), cfsqltype : "timestamp" },
            "modifiedDate"  = { value : now(), cfsqltype : "timestamp" },
            "FK_userId"     = arguments.content.getUser().getId()
        } );

    return arguments.content;
}
```

Run your tests, validate and BOOM, validation errors!!! WHATTTTTT. What could be wrong?

Updating The Unique Validator

It seems our validator is in need of some updating, since if we do an update it will claim that the slug is already there, but I want an update not a creation. So let's udpate it.

```
slug : { required : true, udf : ( value, target ) => {
    if( isNull( arguments.value ) ) return false;
    return qb.from( "content" )
        .where( "slug", arguments.value )
        .when( this.isLoaded(), ( q ) => {
            arguments.q.whereNotIn( "id", this.getId() );
        } )
        .count() == 0;
}},
```

Check out the cool `when()` function. It allows us to switch up the SQL if the actual object has been persisted already. Now run your tests and we should be good now!

Removing Content

```
story( "I want to be able to remove content objects" )
```

BDD

Update the spec with a new story and scenarios:`

```
story( "I want to be able to remove content objects", function(){
    given( "a valid incoming slug", function(){
        then( "it should remove content object", function(){
            var event = DELETE(
                route = "/api/v1/content/Record-Slave-Crystal"
            );

            // expectations go here.
            var response = event.getPrivateValue( "Response" );

            debug( response.getData() );

            expect( response ).toHaveStatus( 200 );
            expect( response.getMessages().toString() ).toInclude(
"Content deleted" );
        });
    });

    given( "an invalid slug", function(){
        then( "it should throw a validation error", function(){
            var event = delete(
                route = "/api/v1/content/bogus"
            );

            // expectations go here.
            var response = event.getPrivateValue( "Response" );

            expect( response ).toHaveStatus( 404 );
        });
    });
});
```

Ok, now let's put it together!

Delete Action

```
/**
* delete
*/
function delete( event, rc, prc ){
    param rc.slug = "";

    prc.oContent = contentService.findBySlug( rc.slug );

    if ( !prc.oContent.isLoaded() ) {
        prc.response
            .setError( true )
            .setStatusCode( event.STATUS.NOT_FOUND )
            .setStatusText( "Not Found" )
            .addMessage( "The requested content object (#rc.slug#) could
not be found" );
        return;
    }

    // populate, validate and create
    contentService.delete( prc.oContent );

    prc.response.addMessage( "Content deleted!" );
}
```

Delete Services

Now to the ugly (funky) SQL

```
/**
 * delete
 */
function delete( required content ){
    var qResults = qb.from( "content" )
        .whereId( arguments.content.getId() )
        .delete();

    arguments.content.setId( "" );

    return arguments.content;
}
```

Run your tests!

Where Do We Go From Here

We should be excited, exhausted, and amazed that we have started to build a headless CMS! This is just the start, what else can we do? Here are some more ideas?

- Content Versioning

- Content Drafts
- Content Categories
- Move to an ORM (Hibernate or Quick)
- Allow creator and editor in content objects
- Pagination
- Search
- The list goes on!!!

Happy Coding!