# Secure your Spring RESTful APIs with JWT — A real world example

Tharsan Sivakumar Follow

May 22, 2018 · 6 min read

In this article we will see how to integrate a REST API authorization using JSON Web Token (JWT) standard and Spring Security into Spring Boot application. A private project has been created by this author and will be used as basis for this article and the code for same can be found on GitHub. Entire code hasn't been presented here so you may need to check the project repository to understand the context. However this will include as much code snippet as possible while keeping the article clean.

## Brief introduction

It's good to get very short technical background about some of the technologies that we are going to touch in this article, before we get into practical code. Hence some of the terms will be briefly explained below.

### Spring boot

Over the past few years, Spring Boot has greatly simplified the configuration of a Spring Framework application. This approach has enabled the developers to enjoy many other powerful features such as

"Basic" security for an application by simply having "Spring Security" dependency on the class path.

## OAuth2

This protocol allows third-party applications to grant limited access to an HTTP service, either on behalf of a resource owner or by allowing the third-party application to obtain access on its own behalf. Access is requested by a client, it can be a website or a mobile application for example.

OAuth defines four roles:

1) Resource Server—Hosts the protected resources

2) Authorization Server—the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization

3) Resource Owner– *user* who authorizes an *application* to access their account. The application's access to the user's account is limited to the "scope" of the authorization granted (e.g. read or write access).

4) Client—is the *application* that wants to access the *user*'s account.

## JWT

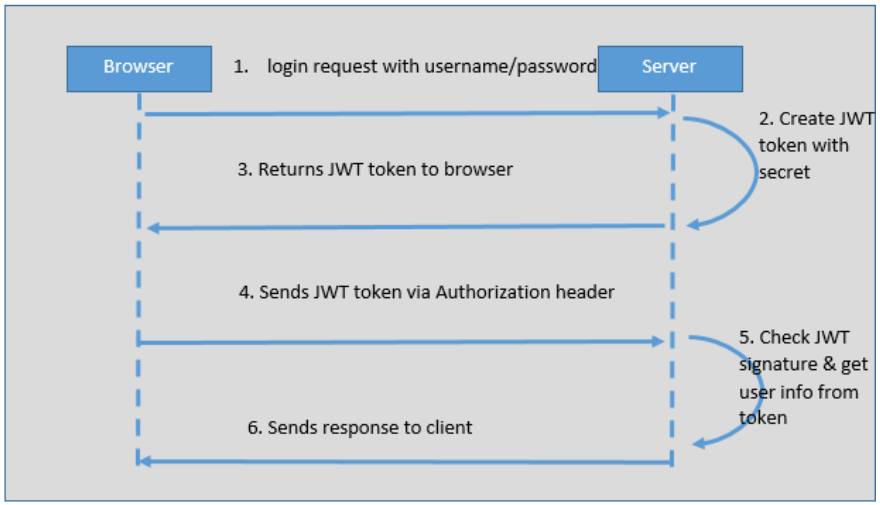
*JSON web tokens*, commonly known as JWTs, are tokens that are used to authenticate users on applications. This technology has gained popularity over the past few years because it enables back end to accept requests simply by validating the contents of these JWTs. Choosing JWT to secure your API endpoints is a great choice because it ensures a

stateless exchange of tokens between the client and the server and it's compact and URL-safe. Applications that use JWTs no longer have to store access tokens in a database or to hold cookies or other session data about their users. This characteristic facilitates scalability while keeping applications secure.

During the authentication process, when a user successfully logs in using his/her credentials, a JSON web token is returned and must be saved locally (typically in local storage). Whenever the user wants to access a protected route or resource (an endpoint), the user agent must send the JWT, usually in the Authorization header along with the request.

When a back end server receives a request with a JWT, the first thing it would do is, to validate the token. This consists of a series of steps such as whether JWT is well formed, verify the signature, validate the claims and ratify client's claim and if any of these fails then, the request will be rejected.

Following are key steps that will be performed while implementing JWT into Spring Framework.




## The Sample Spring boot API Overview

The RESTful Spring Boot API that we are going to secure in this article is a country code manager, which basically keeps country code and some other related information. To clone and run this application, let's issue the following commands. (Hopefully you have configured development environment with Java, maven and git)

### $ git clone https://github.com/tharsans/spring-boot-security-with-jwt.git

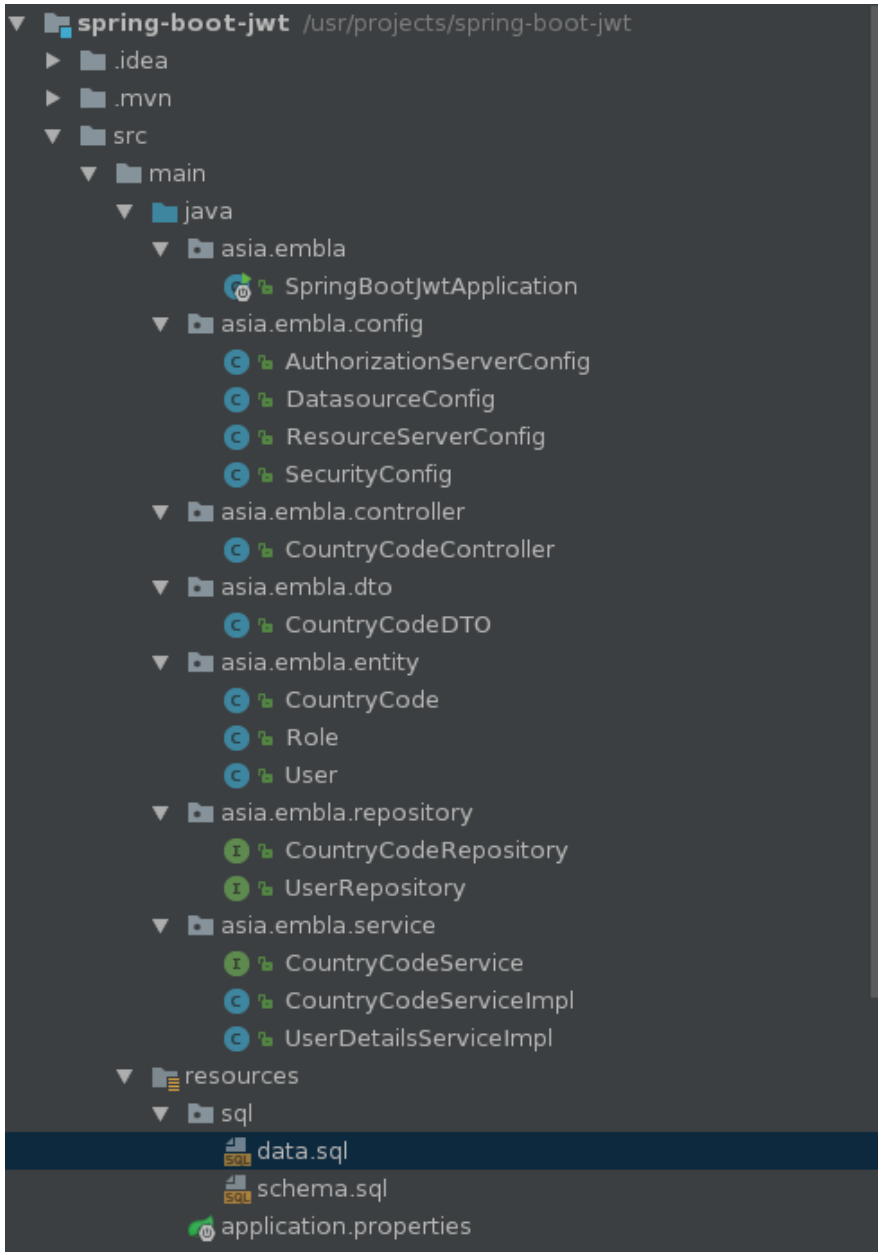
> *$ cd spring-boot-security-with-jwt*

> *$ mvn spring-boot:run*

If everything works as expected, our RESTful Spring Boot API will be up and running. We can test API endpoints by sending requests using

tools like **Postman** or **curl**, but let's have a look on some of the important code segments before we jump into testing.

When you clone the project from Github and import it into your favorite IDE, its structure will look similar to the following image. Here, I am using a screenshot from IntelliJ Community Edition IDE. Yours would look slightly different depending on the IDE you use.


spring-boot-jwt /usr/projects/spring-boot-jwt
.idea
.mvn
src
main
java
asia.embla
SpringBootJwtApplication
asia.embla.config
AuthorizationServerConfig
DatasourceConfig
ResourceServerConfig
SecurityConfig
asia.embla.controller
CountryCodeController
asia.embla.dto
CountryCodeDTO
asia.embla.entity
CountryCode
Role
User
asia.embla.repository
CountryCodeRepository
UserRepository
asia.embla.service
CountryCodeService
CountryCodeServiceImpl
UserDetailsServiceImpl
resources
sql
data.sql
schema.sql
application.properties

Now let's try to understand some of the key code segments in the project one by one.

## Configure Spring Security

Thanks to Spring Boot's auto configuration we will need minimal customization to set. Spring Boot applies its typical convention over configuration approach to application.properties file and all of the configurations in that file will be auto-detected.

## Security configuration class

**@EnableWebSecurity**—This enables spring security and hints Spring Boot to apply all the security steps.

**@EnableGlobalMethodSecurity**—Allows to have method level access control

A custom implementation of UserDetailsService, named UserDetailsServiceImpl (see code in GitHub for details) is injected in order to retrieve user details from the database.

## Configure Authorization Server

**@EnableAuthorizationServer**—This will hint Spring boot to enable an authorization server.

## Configure Resource Server

**@EnableResourceServer**—Resource server will be instantiated by Spring container when a bean is identified by this annotation. By default this annotation creates a security filter which authenticates requests via an incoming OAuth2 token. The resource server has the

authority to define the permission for any endpoint. Here it has been specified to authenticate all end points with “countrycodes”.

## Exposing Resources via a REST Controller

Following are three endpoints are exposed for external http access

1. List country codes—This endpoint is accessible to all authenticated users.

2. Create country codes—This endpoint is accessible only to admin, super admin users

3. Delete Country code—This endpoint is accessible only to super admin users

## Running and Testing the Application

In order to run this application following basic pieces of information is required.

> *1. Client—EmblaMagazineClient*

> *2. Secret -f2a1ed52710d4533bde25be6da03b6e3*

> *3. User names—user, admin, superadmin*

> *4. Password—string*

### Step 1: Generate an access token

**If you try to access the url prior to authenticate yourself you may get the following response.**

> *$ curl http://localhost:8080/countrycodes/*

> *$ {"error":"unauthorized","error_description":"Full authentication is required to access this resource"}*

Hence get the access token by authenticating with correct credentials. Use the following generic command to generate an access token.

> *$ curl <client-id> : <client-secret>@<api-end-point-url >/oauth/token -d grant_type=password -d username=<username> -d password=< password >*

Replace the parameters as per your set up and run the command.

> *$ curl EmblaMagazineClient:f2a1ed52710d4533bde25be6da03b6e3@localhost:8080/oauth/token -d grant_type=password -d username=user -d password=string*

You may receive a response similar to below.

> *{"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MjQ2OTc0NTYsInVzZXJfbmFtZSI6InVzZXIiLCJzY29wZSI6WyJyZWFkIiwid3JpdGUiXSwiYXV0aG9yaXRpZXMiOlsiVVNFUiJdLCJhdWQiOlsiRW1iaGFNYWdhemluZVJlc3RBUEkiXSwianRpIjoiMTEzOGJkZmEtMjRkNC00O*

> *WU4LWEyNTctOTFjNWEwOGNjOGVlIiwiY2xpZW50X2lkIjoiRW1ibGFNYWdhemluZUNsaWVudCJ9.MvoQSVsEvGRuKK1YGNneMEOJt0VjHdIOVGyD4CywbJI","token_type":"bearer","refresh_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1MjcyNDYyNTYsInVzZXJfbmFtZSI6InVzZXIiLCJzY29wZSI6WyJyZWFkIiwid3JpdGUiXSwiYXV0aG9yaXRpZXMiOlsiVVNFUiJdLCJhdWQiOlsiRW1ibGFNYWdhemluZVJlc3RBUEkiXSwianRpIjoiOGE3OWFmNWItZTRhYy00NTBmLTg3OGEtMzM0YjU1OTA0MDE0IiwiY2xpZW50X2lkIjoiRW1ibGFNYWdhemluZUNsaWVudCIsImF0aSI6IjExMzhiZGZhLTI0ZDQtNDllOC1hMjU3LTkxYzVhMDhjYzhlZSJ9.4bTntI4uRNhu5zy-UGaY3ihajoUPNkzJDhr5qsfNAr0",”expires_in”:43199,”scope”:”read write”,”jti”:”1138bdfa-24d4–49e8-a257–91c5a08cc8ee”}*

## Step 2: Use the token to access resources through your RESTful API

Use the generated token as the value of the Bearer in the Authorization header as follows.

> *$curl http://localhost:8080/countrycodes/ -H “Authorization: Bearer GENERATED_AUTH_TOKEN”*

The response will look like.

> *[{“id”:1,”name”:”Sri Lanka”, “code”:”94", “isoCode”:”LK” }, {“id”:2, “name”:”India”, “code”:”91", “isoCode”:”IN”} , {“id”:3,”name”:”United Kingdom”,”code”:”44","isoCode”:”GB”},{“id”:4,”name”:”United States”,”code”:”1","isoCode”:”US”}, {“id”:5,”name”:”Singapore”,”code”:”65","isoCode”:”SG”}]*

### If you try to access the DELETE end point using this token your will get following error message.

Hence in order to access the other end points you need to log as relevant users who has authority to access those resources and use the received authentication token in the request header respectively. Please use the following commands to create and delete country codes respectively.

> *# Issue a POST request to create country code*

> *$ curl -H “Content-Type: application/json” -H “Authorization: Bearer GENERATED_AUTH_TOKEN”*

> *-X POST -d ‘{“id”:”100",”name”:”Australia”, “code”:”61", “isoCode”:”AU”}’ http://localhost:8080/countrycodes/*

> *# Issue a DELETE request to delete country code*

> *$ curl -H “Content-Type: application/json” -H “Authorization: Bearer GENERATED_AUTH_TOKEN”*

> *-X DELETE ‘http://localhost:8080/countrycodes/{id}?id=100'*

## Conclusion

Securing RESTful Spring Boot API with JWTs is not a hard task. This article showed that by creating simple project. This will helps us to protect our endpoints from unknown users, enable users to register themselves, and authenticate existing users based on JWTs.

## References & Useful Readings

1. https://medium.com/@nydiarra/secure-a-spring-boot-rest-api-with-json-web-token-reference-to-angular-integration-e57a25806c50

2. https://auth0.com/blog/implementing-jwt-authentication-on-spring-boot/

3. https://dzone.com/articles/secure-spring-rest-with-spring-security-and-oauth2