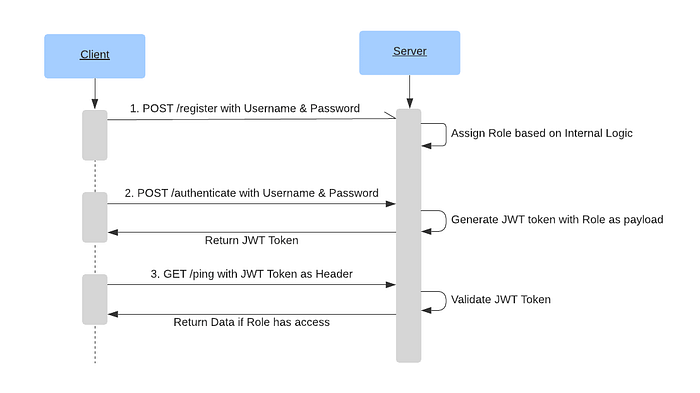
**Spring Boot API Security with JWT and Role-Based Authorization**

**Steps to use JWT authentication and authorization using Spring Boot Rest API, Spring Data JPA:**

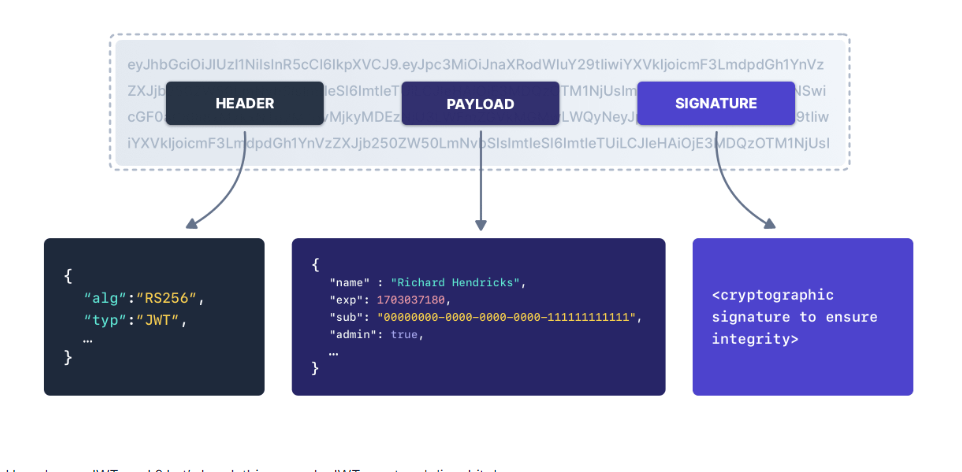
* **User Registration API**, where each user is assigned a Role.
* **User Authentication**, where valid users are retuned a JWT Token.
* **Role-based access** to specific API targets by means of providing a valid JW

**JSON Web Token (JWT)**

*JSON Web Token (JWT) is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret or a public/private key pair using****RSA****or****ECDSA****.*

Detailed description of JWT -  [here](https://jwt.io/introduction/" \t "_blank).:

JWT will have Header, Payload and Signature.



**What can we solve using JWT?**

* We can securely transmit and store verified and trusted information.
* We will use this capability to store the user’s role as part of the JWT Token’s payload.
* We will generate a JWT Token on the server as soon as the user is able to verify their credentials.
* When a user wants to hit an API, we will append the token to the request’s header.
* This will not only inform the server of who is trying to make the request but also the role that will help the API server determine if that person has access to the API or not.
* For each API, we will assign which roles are able to access them.
* If the user’s role matches the roles allowed by that API, the request goes through. Else, we return a **403 Forbidden Request.**

**Project Setup**

Add all the requisite dependencies to your project using [**Spring Initializr**](https://start.spring.io/)**.**

The dependencies we will be using are

1. Spring Web
2. Spring Data JPA
3. Spring Security
4. MySQL Driver
5. JSON Web Token

Sample pom.xml

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt</artifactId>  
 <version>0.9.1</version>  
 </dependency>  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
</dependencies>

And here is the project structure. I have added several new packages (in bold) that will help us organize our classes better.

├── pom.xml  
└── src  
└── main  
├── java  
│ └── com  
│ └── codegnan  
│ └── ecom  
│ ├── SpringRoleJwtApplication.java  
**│ ├── config  
│ ├── controller  
│ ├── dao  
│ ├── model  
│ └── service**  
└── resources  
├── application.properties

**Spring Security Configurations**

Spring Security will be used for 2 tasks:

1. *Authentication*: The Check to see if the given user has the right username and password.
2. *Authorization*: The Check to see if the user has enough permissions to access the API that they tried to hit.

***WebSecurityConfigurerAdapter***

1. Create a class WebSecurityConfig that extends the ***WebSecurityConfigurerAdapter.***
   1. Start off by annotating the class as @Configuration.
   2. The @EnableWebSecurity is a marker annotation. It allows Spring to find and automatically apply this class to the security of the application.
   3. We need to secure our API’s by restricting which roles are able to execute a particular method. This is achieved by adding the annotation @EnableGlobalMethodSecurity(prePostEnabled = true}.

2. **Authentication using AuthenticationManager**

* Spring Security uses something called an **AuthenticationManager** to validate if a given user has the right credentials (based on username and password).
* The **AuthenticationManager** Interface has exactly one method ***authenticate***which is called to verify if the username and password provided by a user are correct.
* But the **AuthenticationManager** needs to know where the user’s username and password have been stored.
* That is why we override the ***configure***method where Spring will pass an **AuthenticationManagerBuilder** in the class **WebSecurityConfig**

@Override

**public** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {

auth.userDetailsService (userDetailsService).passwordEncoder(encoder());

}

* The **AuthenticationManagerBuilder**accepts a custom implementation of the **UserDetailsService**interface (which we will implement when we are building our services).
* Meaning, we will have to implement this interface in a class.

Eg **UserServiceImpl**

* Also at this stage, if we are using some form of encryption to store our password in the database, the **AuthenticationManager**needs to know about that as well.
* It’s actually a very bad idea to store a password as plaintext. Here, we will be using BCrypt to encode our passwords.
* The **AuthenticationManager** we just configured is added to the Spring Application Context and is added as a bean by overriding the ***authenticationManagerBean*** method.

1. **Authorization**

* To set up Authorization, we need to provide the configuration by overriding the ***configure***method, where we are passed a reference to the default **HttpSecurity** configuration.

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.cors().and().csrf().disable()

.authorizeRequests()

.antMatchers("/users/authenticate", "/users/register").permitAll()

.anyRequest().authenticated()

.and()

.exceptionHandling().authenticationEntryPoint(unauthorizedEntryPoint).and()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.***STATELESS***);

http.addFilterBefore(authenticationTokenFilterBean(), UsernamePasswordAuthenticationFilter.**class**);

}

* For the graceful handling of Unauthorized requests, we pass along a class that implements **AuthenticationEntryPoint**. We will return a *401 Unauthorized* when we encounter an exception.

In the line:   
exceptionHandling().authenticationEntryPoint(unauthorizedEntryPoint,

It uses an autowired object of the class UnauthorizedEntryPoint.

* Because we are using JWT to store roles, we need to translate that into something that Spring Securitycan understand. The JWT Token needs to be parsed to fetch roles that the **SpringSecurityContext** needs to become aware of before it goes on to check if the API’s permissions will allow it.
* Hence we pass the JwtAuthenticationFilter (Which we will come to in a later step).

**JWT Token Creation - Steps to create a JWT Token using TokenProvider class**

* Create a component class that will handle all things related to JWT.
* We will call the methods within this class to generate the JWT as well as validate the JWT when the user sends it back to us.
* The ***generateToken***method builds and signs the JWT Token that we will pass along to the user as soon as they authenticate.
* Here is where as part of the payload, we will add the *username*, *roles*(comma separated), and the *issuedAt* and *expiration* timestamps.
* ***validateToken***basically checks if the username on the token payload matches the UserDetails. It also checks if the token has expired.

Spring Security Context holds the information of an authenticated user represented as an *Authentication* object. In order to construct this Authentication object, we need to provide a UsernamePasswordAuthenticationToken which will later be used by our AuthenticationManager (Which we configured previously) to Authenticate our user. To construct, we are passing along the user details as well as a collection of authorities(roles) that we parse from the JWT Token. This is exactly what's happening inside ***getAuthenticationToken*.**

**JWT Authentication Filter**

J**WTAuthenticationFilter**  will filter out requests that have JWT as header and translate that to something Spring Security can understand using the methods from the Token Provider we just created. This extends the **OncePerRequestFilter**meaning it's going to look for the JWT token in every single request and update the SecurityContext.

This concludes our Security Configurations, Let’s move on to creating the services.

We are using some constants that we are fetching by key. These are stored in the .properties file

jwt.token.validity=18000  
jwt.signing.key=signingkey  
jwt.authorities.key=roles  
jwt.token.prefix=Bearer  
jwt.header.string=Authorization

**Service Layer**

**Model**

Define a class User.

1. Username (Unique)
2. Password
3. Name
4. Business Title
5. Roles

A single user can have multiple roles. We define Roles to have

1. Name
2. Description

So let's create 2 classes under the model subdirectory, **User,** and **Role.**As mentioned earlier, we will be storing all this information in a MySQL Database and so **User**& **Role**classes can be written as Entity classes. What that really means is that we will be annotating the member variables as appropriate.

There exists a Many-to-Many relationship between User and Roles, meaning that each user can assume multiple roles and each role can be assumed by many users. So we will annotate that accordingly.

@Entity  
public class User {  
  
 @Id  
 @GeneratedValue(strategy= GenerationType.*IDENTITY*)  
 private long id;  
  
 @Column  
 private String username;  
  
 @Column  
 @JsonIgnore  
 private String password;  
  
 @Column  
 private String email;  
  
 @Column  
 private String phone;  
  
 @Column  
 private String name;  
  
 @Column  
 private String businessTitle;  
  
 @ManyToMany(fetch = FetchType.*EAGER*, cascade = CascadeType.*ALL*)  
 @JoinTable(name = "USER\_ROLES",  
 joinColumns = {  
 @JoinColumn(name = "USER\_ID")  
 },  
 inverseJoinColumns = {  
 @JoinColumn(name = "ROLE\_ID") })  
 private Set<Role> roles;  
  
}

**@*JsonIgnore*** annotation is used because we store password along with other user details, we don’t want it returned as a response to a query that returns the user object. This takes care of stripping the password from the API response.

@Entity  
public class Role {  
 @Id  
 @GeneratedValue(strategy= GenerationType.*IDENTITY*)  
 private long id;  
  
 @Column  
 private String name;  
  
 @Column  
 private String description;  
}

**Data Transfer Object** for User entity, the call to **UserDto**. **getUserFromDto**is used for converting into User Object .

public class UserDto {  
   
 private String username;  
 private String password;  
 private String email;  
 private String phone;  
 private String name;  
 private String businessTitle;  
  
 public User getUserFromDto(){  
 User user = new User();  
 user.setUsername(username);  
 user.setPassword(password);  
 user.setEmail(email);  
 user.setPhone(phone);  
 user.setName(name);  
 user.setBusinessTitle(businessTitle);  
   
 return user;  
 }   
}

**JPA configuration**. This can be added to the application.properties file.

spring.datasource.url=jdbc:mysql://localhost:3306/springsecurity  
spring.datasource.username=root  
spring.datasource.password=password  
spring.jpa.show-sql=**true**spring.jpa.hibernate.ddl-auto=create-drop  
spring.user.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5

**DAO**

To perform read and writes on our database, we will create the UserDao and RoleDao repositories (annotated with @). These are interfaces that extend *CrudRepository<?,?>*where the 1st parameter represents the object model, in this case, **User** and **Role**. In each case, the 2nd parameter should be datatype of the unique id of each user or role object.

Now, in addition to the methods the CRUD repository brings to the table, we can add additional derived queries.

A derived query method name has two main components. An introducer such asfind, read, query, count, or get. And a criterion that starts after the first By keyword. The first By behaves as the delimiter to indicate the start of the actual query criteria. The amazing thing about derived queries is that the method names are automatically parsed into queries, so we don't have to do much else apart from defining the methods themselves.

For the user, we want to be able to query by *username*and for the role, we want to be able to lookup by name. So we will introduce a **findByUsername**method in the UserDao and **findRoleByName**in the RoleDao.

@Repository  
public interface UserDao extends CrudRepository<User, Long> {  
 User findByUsername(String username);  
}@Repository  
public interface RoleDao extends CrudRepository<Role, Long> {  
 Role findRoleByName(String name);  
}

**Service**

We can now get to the step of writing the services. The services will ultimately use the service methods in our controller, so ultimately, this is where our core business logic might live.

In the interest of best practice, we will be drawing out blueprints in the form of interfaces for our user and role services.

public interface UserService {  
 User save(UserDto user);  
 List<User> findAll();  
 User findOne(String username);  
}public interface RoleService {  
 Role findByName(String name);  
}

We will implement these interfaces into our Services.

Let’s start with the **RoleServiceImpl**. Pretty Straightforward. Implement the RoleService Interface and Override the **findByName** method. Now in order to fetch the role, we will access our RoleDao which has the requisite methods to query the DB and find role by name. Simply Autowire the RoleDao into our Service and our job here is pretty much done.

@Service(value = "roleService")  
public class RoleServiceImpl implements RoleService {  
  
 @Autowired  
 private RoleDao roleDao;  
  
 @Override  
 public Role findByName(String name) {  
 Role role = roleDao.findRoleByName(name);  
 return role;  
 }  
}

**UserServiceImpl** class:  
Implement the UserDetailsService with the lone ***loadUserByUsername***method.

* We can have our UserServiceImpl also implement the UserDetailsService interface in addition to the UserService Interface.
* In the method  **loadByUsername,** looks up the user through the DAO repository for the user object and returns a new **org.springframework.security.core.userdetails.User**constructed with username, password, and a Set of granted authorities.
* The implementations of ***findAll(), findOne() and save();***are self explanatory. You will notice that in the save() method, we are assigning every user with “ROLE\_USER”. Here we get the chance to add some additional business logic to determine if the user is also an admin. For the purposes of this article, I am assuming that any user with the domain @admin.edu will additionally have admin privileges.

**Controller**

The first thing the user needs to do is to register. The bare minimum that we need to provide is a username and password. simply calling the service method to save the user does the trick.

In order to access the APIs, we need to pass along a server-generated JWT Token. We have done all the groundwork for that in our TokenProvider. We use the generateTokenMethod and pass along the response.

The other 2 API’s are just here for testing.

Add the 2 new DTOs **Token**and **LoginUser**

public class LoginUser {  
 private String username;  
 private String password;  
}public class AuthToken {  
 private String token;  
}

That completes our application using JWT.

**Finishing Steps**

create the tables in DB populate them where needed.

Create another file called ***query.sql*** file alongside our .properties file and configure the data source to point to our “*springsecurity”* database.

The first step is to create the tables. Add the following statements. Note, you can only have DDL execute here.

**drop table if exists** role;  
**drop table if exists user**;  
**drop table if exists** user\_roles;**create table** role (id **bigint not null auto\_increment**, description **varchar**(255), **name varchar**(255), **primary key** (id)) **engine**=MyISAM;**create table** user\_roles (user\_id **bigint not null**, role\_id **bigint not null**, **primary key** (user\_id, role\_id)) **engine**=MyISAM;

With that step completed, **start your server!**

You will notice that upon startup the 3 new tables are created in your MySql Database.

Time to populate our **role**Table with all the roles. Run the following statements

**INSERT INTO** role (id, description, **name**) **VALUES** (4, 'Admin role', 'ADMIN');  
**INSERT INTO** role (id, description, **name**) **VALUES** (5, 'User role', 'USER');

The code for the above implementation is provided in :  
https://github.com/sudheercg/ecomjwtbackend/tree/main/spring-role-jwt

EndPoints to be tested:

For Registration:

<http://localhost:8080/users/register>

Sample Request Parameters:  
 {

"username":"sudheer",

"password":"password",

"email":"code@codegnan.com",

"phone":"123456789",

"name":"codeman",

"businessTitle":"Developer"

 }

Sample response :

{

    "id": 6,

    "username": "sudheer",

    "email": "code@codegnan.com",

    "phone": "123456789",

    "name": "codeman",

    "businessTitle": "Developer",

    "roles": [

        {

            "id": 2,

            "name": "USER",

            "description": "User role"

        }

    ]

}

Authentication(login):  
<http://localhost:8080/users/authenticate>

Sample request:

{

“username":"sudheer",

"password":"password"

}

Sample Response:

{

    "token": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJzdWRoZWVyIiwicm9sZXMiOiJST0xFX0FETUlOIiwiaWF0IjoxNzIxNDc0Mzk2LCJleHAiOjE3MjE0OTIzOTZ9.VzgGQ58vY-HmNevnuXTRniXa3JkbMxth0DVBr6-xU20"

}

The above response is the token generated by JWT.

View Products:

We have defined 2 roles, ADMIN and USER.

By default when user is registered using /register endpoint, the user will be allocated ‘USER’ privilege.

For testing, We can change the privileges in the user\_roles table for a user from the ‘springsecurity’ database.

To view the Products – Users with 'both ‘ADMIN’ AND

‘USER’ roles can view products;

EndPoint using GET:

http://localhost:8080/products

EndPoint using POST, only the user with the Role ADMIN can access this endpoint and can add the products :

<http://localhost:8080/products>

Add bearer token to the request in Authorization header.

Sample request:

{

"name":"Keyboard",

"price":2000.33

}

Sample response

{

    "id": 8,

    "name": "Keyboard",

    "price": 2000.33

}

To view Cart and its items:

Sample request: <http://localhost:8080/cart?userId=1>

Sample response:

{

    "id": 2,

    "user": {

        "id": 1,

        "username": "sudheer",

        "email": "s@sud.com",

        "phone": "123456789",

        "name": "sudheery",

        "businessTitle": "Developer",

        "roles": [

            {

                "id": 1,

                "name": "ADMIN",

                "description": "Admin role"

            }

        ]

    },

    "items": [

        {

            "id": 3,

            "product": {

                "id": 2,

                "name": "Songbook",

                "price": 2000.0

            },

            "quantity": 2

        },

        {

            "id": 2,

            "product": {

                "id": 1,

                "name": "Bible",

                "price": 2000.0

            },

            "quantity": 2

        }

    ]

}

Sample response, if cart is not found for a particular user.

{"message": "Cart not found"}

To add poroduct to the cart:   
<http://localhost:8080/cart/add>

Sample request:

{

"userId":1,

"productId":1,

"quantity":6

}

Sample response:

{"message": "Product added to cart"}

To Update a Product:

<http://localhost:8080/cart/update> , use PUT method

Sample request:  
{

"userId":1,

"productId":1,

"quantity":33

}

Sample response:

{"message": "Cart updated"}

To delete a product:

Sample request:

{

"userId":1,

"productId":1,

"quantity":33

}

Sample response:

{"message": "Product removed from cart"}