Spring Security 6 with Spring Boot 3:

Spring Boot provides a “spring-boot-starter-security” starter that aggregates Spring security related dependencies together.

Some Deprecated things in SpringBoot3:

1 .webSecurityConfigurerAdapter

2. EnableGlobalMethodSecurity

NOTE:  
Earlier we were using “@EnableGlobalMethodSecurity “ .But now it is deprecated and spring security 6 provides a new annotation called “@EnableMethodSecurity” to provide method level security .By default this annotation enables “PreAuthorize ,PostAuthorize, PreFilter and PostFilter “annotations

3.antMatchers

4.authorizeRequests

SpringBoot Auto configuration for Spring Security:

1. “Spring-boot-starter-security “ starter that aggregates Spring Security-ralated dependencies together.
2. Enables Spring security’s default configuration which creates a servlet Filter as a bean named “SpringSecurityFilterChain” provides default login form for us.

This securityFilterChain bean enables component based spring security. Earlier we were using the old style like extending “webSecurityConfigureAdapter” from our security config class.

The benefits we were getting from this webSecurityConfigureAdapter is usually we were getting 3 overloaded configure methods.

1. “AuthenticationManagerBuilder” method where we can provide userdetails to our in memory.
2. Protected void configure(HttpSecurity http) throws Exception {

}

We can have all our authourization related configuration here like what all endpoints we want enable security or what all endpoints we want to bypass without implementing security.

3)PasswordEncoder.

Ex:

securityConfig extends webSecurityConfigureAdapter{

}

Instead of using those configure methods we have to create bean “UserDetailsService” to define as many users we want to define.

EXAMPLE:  
@Bean  
 public UserDetailsService userDetailsservice(){  
 UserDetails purushotham= User.*builder*()  
 .username("purushotham")  
 .password(*passwordEncoder*().encode("purushotham"))  
 .roles("ADMIN")  
 .build();

UserDetails rishi= User.*builder*()  
 .username("rishi")  
 .password(*passwordEncoder*().encode("rishi"))  
 .roles("USER")  
 .build();  
 return new InMemoryUserDetailsManager(purushotham,rishi);  
  
}

Now, we are using “Security filterchain “to enable component based spring security. This method helps us to configure Authourization related stuff.

We can have all our authourization related configuration here like what all endpoints we want enable security or what all endpoints we want to bypass without implementing security.

@Bean  
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
 http  
 .authorizeHttpRequests((authorize) -> authorize  
 .requestMatchers("/get").permitAll()  
 .requestMatchers("/users").hasRole("ADMIN")  
 .anyRequest().authenticated()  
 )  
 .httpBasic(*withDefaults*())  
 .formLogin(*withDefaults*());  
 return http.build();

NOTE:

“build( )” method returns “DefaultSecurityFilterChain” class .

“SecurityFilterChain” is an interface and “DefaultSecurityFilterChain” is the implementation class.

1. Create default user name as “user” and randomly generated password that is logged to the console.
2. Springboot provides properties to customize default user’s username and password.
3. Protects the password storage with BCrypt algorithm.
4. Lets the user log out(default logout feauture).
5. CSRF attack prevention(enabled by default).
6. If spring security is on the classpath, SpringBoot automatically secures all HTTP enpoints with basic authentication.

UserDetailService:

“UserdetailService” interface plays a crucial role in providing user authentication and authourization. In simple terms it is responsible for fetching user information and convert it into a format that SpringSecurity can use to authenticate and authourize users in our application.

The userdetailservice returns a special object called “userdetails “ which represents the authenticated user. This object contains user’s username and password, and authourities.

Once the “UserDetailService” returns the “UserDetails” object, Spring Security takes over and performs the authentication process.

Difference b/w UserDetails and UserDetailsService:

Once “userDetails” is an interface in spring security that represents the authenticated users information.

It contains methods to retrieve the user’s username, password, authourities and other account related attributes.

Implementation of userdetails interface encapsulates user specific data and are returned by the “userDetailsService”

USER DETAILS SERVICE:

“userDetailsService” is an interface in spring security that provides user authentication by fetching userinformation from a datasource(e.g database or an external user service).

It defines a single method called “loadBy UserName” that takes username as input and returns fully populated “UserDetails” object for that user.

The custom implementation of “UserDetailsService” is responsible for converting the user data retrieved from the source into a “userDetails” object.

UserDetails is an interface that represents the user's information and contains details such as username, password, and authorities. On the other hand, UserDetailsService is an interface that provides a way to fetch user information and returns a fully populated UserDetails object for authentication purposes. The UserDetailsService implementation is customized to fetch user data from the desired data source and convert it into a UserDetails object to be used by Spring Security during the authentication process.

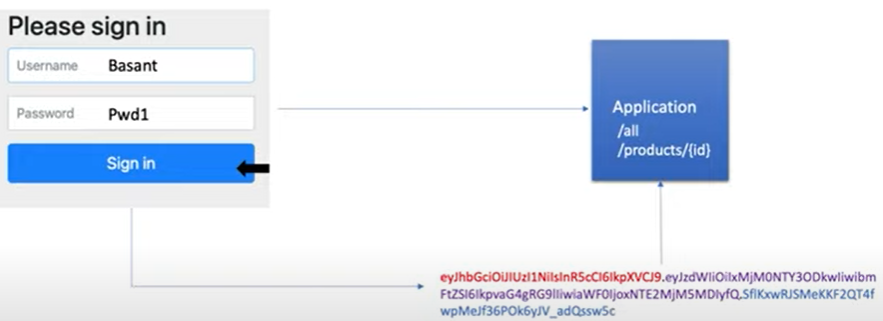
Every time when the user needs to access any end point he needs to give username and password if user is valid then it will navigate to url. So these type of problems We are using JWT.



JWT:

Here we are allowing user to give his username and password first time then when user sends a request it will generate one token for the user so that going forward he can pass this token to access any endpoint of the application.

To build a jwt token we need to understand few components .



The JWT token is in different colors each color is pointing some different component .

JWT TOKEN contains 3 components.

1. HEADER:

{

"alg": "HS256",

"typ": "JWT"

}

Header contains the algorithm what we want to use for encryption and the type of token.

1. PAYLOAD

{

"sub": "1234567890",

"name": "John Doe",

"iat": 1516239022

}

Payload contains users username and when our token will be expired these type of information.

3)VERIFY SIGNATURE

HMACSHA256(

Base64UrlEncode(header) + ” . ” +

Base64UrlEncode(payload).

Your-256-bit-secret

)

VERIFY SIGNATURE just contains how we are encrypting our token.

How to generate a token?

To generate a token we need to add 3 dependencie

<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-api</artifactId>  
 <version>0.11.5</version>  
</dependency>  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-impl</artifactId>  
 <version>0.11.5</version>  
</dependency>  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt-jackson</artifactId>  
 <version>0.11.5</version>  
</dependency>

Public String generateToken( String username){

//all the components like header, payload and verify signature called as a claims

Map<String,object>claims= new HashMap<>( );

Return createToken(claims,username);

}

public String generateToken(  
 Map<String, Object> extraClaims,  
 UserDetails userDetails  
)

//all the components like header, payload and verify signature is called //as // as claims.

{  
 return buildToken(extraClaims, userDetails, jwtExpiration);  
}

private String buildToken(  
 Map<String, Object> extraClaims,  
 UserDetails userDetails,  
 long expiration  
) {  
 return Jwts  
 .*builder*()  
 .setClaims(extraClaims)  
 .setSubject(userDetails.getUsername())  
 .setIssuedAt(new Date(System.*currentTimeMillis*()))  
 .setExpiration(new Date(System.*currentTimeMillis*() + expiration))  
 .signWith(getSignInKey(), SignatureAlgorithm.*HS256*)  
 .compact();  
}

After generating the token we need to allow all users to generate token without any restriction.

@PostMapping("/authenticate")  
public ResponseEntity<AuthenticationResponse> authenticate(  
 @RequestBody AuthenticationRequest request  
) {  
 return ResponseEntity.*ok*(service.authenticate(request));  
}

public AuthenticationResponse authenticate(AuthenticationRequest request) {  
 authenticationManager.authenticate(  
 new UsernamePasswordAuthenticationToken(  
 request.getEmail(),  
 request.getPassword()  
 )  
 );  
 var user = repository.findByEmail(request.getEmail())  
 .orElseThrow();  
 var jwtToken = jwtService.generateToken(user);  
 revokeAllUserTokens(user);  
 saveUserToken(user, jwtToken);  
 return AuthenticationResponse.*builder*()  
 .accessToken(jwtToken)  
 .build();  
}

AuthenticationManager:

To generate the token we need to allow only the users who are registered. (who are there in our DB)

The authentication manager is responsible for that.

First we need to inject authentication nmanager.

private final AuthenticationManager authenticationManager;

public AuthenticationResponse authenticate(AuthenticationRequest request)

{

//Here we are checking the user is registered or not.

authenticationManager.authenticate(  
 new UsernamePasswordAuthenticationToken(  
 request.getEmail(),  
 request.getPassword()  
 )  
 );

//code to generate token

AUTHENTICATION FILTER IN JWT:

The job of an authentication filter in JWT (JSON Web Token) authentication is to validate and process incoming requests containing JWT tokens to authenticate users. JWT is a popular mechanism for implementing stateless authentication in web applications, and the authentication filter plays a crucial role in this process.

We need to write a logic to validate token. Once we receive the token going forward we will pass this token to access each and every endpoint.

How our spring authentication manager will know what is this token?

For that we need to extract the token and we need to get the information from the token then only will allow this token to validate.

We need to write a filter and in that filter we need to write logic to extract this particular token from the header and then get username from the token and check in db and do all type of validation whether this token is correct or not.

SECURITY CONTEXT HOLDER:

The security context contains information about the authenticated user, including their authentication details, roles, and other security-related information.

After a user is successfully authenticated using a JWT token, the **SecurityContextHolder** is responsible for storing the user's security context. This context typically includes an instance of **Authentication**, which represents the user's authentication status and credentials.

HIBERNATE:

If we want to get a particular student data from a particular email id

Below is a JPQL query:

@Query(“select s from student s where s.emaild=?1)

Student getStudentByEmailAddress(String emailId)\_;

Native Query Example:

If we are not able define JPQL queries in those cases we can define Native sql queries also.

When ever we have very complex queries to work with we can go with native queries.

Example:

@Query(

value=” select \*from tbl\_student s where s.email\_address = ?1“,

nativeQuery = true

)

Student getStudentByEmailAddressNative(String emailId);

Qyery Named params:

Now suppose if we have a multiple parameters passing over here.

Passing with ?1 and ?2 is not a good approach then we can use these Query Named Parameters.

Example:

@Query(

Value = “SELECT \* FROM tbl-student s where s.email\_address = :emailId,

nativeQuery = true

)

Student getStudentByEmailAddressNativeNamedParam(@Param(“emailId “)String emailId);

@Transactional And @Modifying annotation:

The above all the queries like JPQL and Native Queries and Native queries with parameter names.

What if we want to update the data or delete the data?

This is possible using the @Modifying annotation .

As we are modifying the data there should be a transaction for the data so all the methods that we want make it transactionable a transaction is created and some operation is performed on it and transaction is committed back to the database. All these things we can do by using @Transactional annotation.

Example:

@Modifying

@Transactional

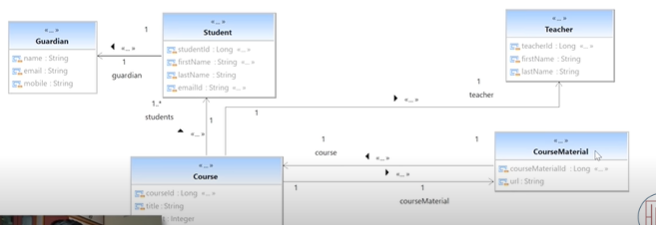
@Query(

Value = “update tbl\_student set first\_name=?1 where email\_sddress = ?2”,

nativeQuery = true

Int updateStudentNameByEmailId(String firstName,String emailId);

MAPPINGS:



One To One relationship:

In the above class diagram we have two classes that is course class and course material for these two classes there is one to one relationship. Because for 1 course there will be a 1 course material.

The course is the primary key in course table and this particular course id will act as a foreign key in course material table.

Now we need to define for which particular column that foreign key will be applied. For that we need to use @JoinColumn.

Course:

private Long courseId;  
private String title;  
private String credit;

@OneToOne(

mappedBy=”course”

)

Private courseMaterial;

courseMaterial:

private long courseMaterialId;  
private String url;  
@OneToOne(

cascade=CascadeType.All)  
@JoinColumn(  
 name ="course\_id",  
 referencedColumnName = "courseId"  
)  
private Course course;

here we need to define name of the column it is going to create for foreign key i.e.,

name =”course\_id”

and the “referenceColumnName” means which particular column it is referencing to in course table.

Cascading:

Cascading means to pass the properties or to pass the permissions to our child elements.

FetchType:

How we want to fetch the data. There are two types available i.e.,

1)EAGER fetching.

2)LAZY fetching.

If we are having the relationship between two tables once we call the data of a parent do we need to fetch the data of a child also or we just need the parent only. To specify this we can use fetchtype.

fetchType.EAGER will fetch everything including child.

fetchType.LAZY will not fetch only the parent table information until we specifically ask to bring fetch the child.

OneToMany:

In the above diagram the course and teacher tables have one to many relationship i.e., (one teacher can teach many courses and reverse is many courses would be taught by one teacher).

@Id  
@SequenceGenerator(  
 name="teacher\_sequence",  
 sequenceName = "teacher\_sequence",  
 allocationSize=1  
)  
@GeneratedValue(strategy = GenerationType.*SEQUENCE*,generator="teacher\_sequence")

@GeneratedValue(strategy = GenerationType.*SEQUENCE*,generator="teacher\_sequence")  
private long teacherId;  
private String firstName;  
private String lastName;  
@OneToMany(cascade=CascadeType.*ALL*)  
@JoinColumn(  
 name="teacher\_id",  
 referencedColumnName ="teacherId"  
)  
private List<Course> courses;

**Spring Integration:**

Spring Integration enables lightweight messaging within spring-based applications and supports integration with external systems via declarative adapters.

