Integrating JWT with Spring Security 6 in Spring Boot 3

Introduction: -

In our previous article, we covered the fundamentals of Spring Security implementation for our Spring Boot project. Building on that foundation, this article will shift its focus towards a more advanced topic: integrating JWT (JSON Web Token) with Spring Security in our Spring Boot application. This will enable us to enhance our security framework by incorporating robust authentication and authorization mechanisms using JWT.

Objective: - **To ensure that critical endpoints are accessible only with a valid JWT token.**

In our Spring Boot project, we primarily have two critical REST endpoints: one for fetching all employee data, and another for adding new employees, which are secured and necessitates JWT-based authentication for access.

In our Spring Boot project, we primarily have two critical REST endpoints: one for fetching all employee data, and another for adding new employees, which are secured and necessitates JWT-based authentication for access.

We have made essential endpoints like user registration and login available without any security. When a user logs in, they receive a JWT (JSON Web Token), which must be used for authorization in all subsequent API requests.

Upon completing this article, you will be well-equipped to integrate JWT with Spring Security 6 in a Spring Boot 3 project, thereby bolstering your application's security. Additionally, you'll be able to adapt and modify this implementation as your project requirements evolve.

Introduction to JWT

What is JWT?

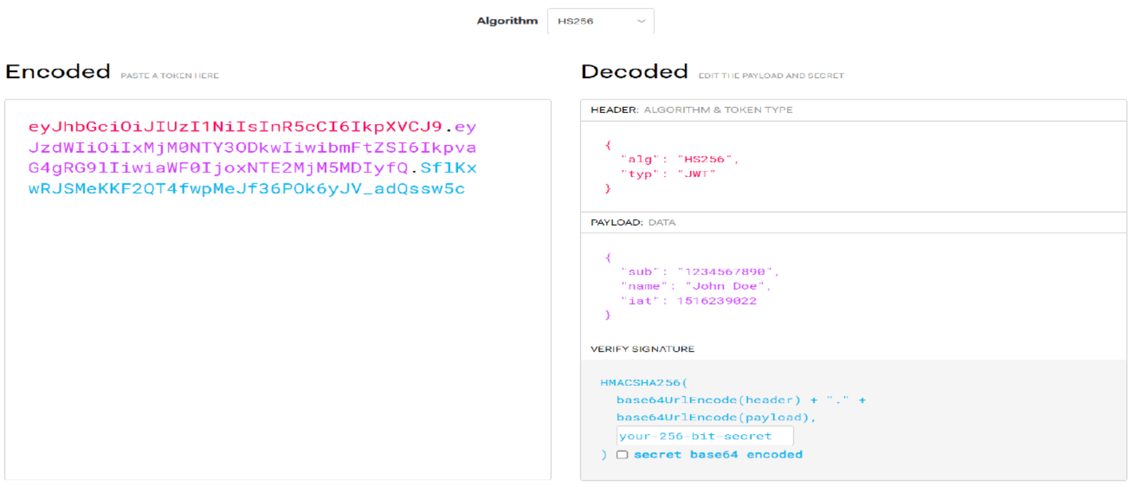
JWT, or JSON Web Tokens, are like digital passes that help keep web applications secure. When someone logs into an application, the server gives them a JWT.

It’s like getting a badge that proves who you are every time you come back. Each time the user wants to access a secure part of the application, they show this badge, and the server knows it’s safe to let them in.

Here’s why JWTs are so great:

1. **Easy to Manage**: They are simple strings of text, making them easy to handle and send between computers and devices.
2. **All-in-One**: Everything the server needs to verify the user is packed into the JWT. This means the server doesn’t have to keep asking a database for information, speeding things up.
3. **Safe**: They can be encrypted and signed, making it hard for unauthorized people to mess with them.

A JWT token typically appears as shown on the **left side** in the provided image below:



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

Parts of a JWT token: -

A JWT consists of three main parts, each separated by a dot (.). Here's an easy breakdown:

1. **Header**
   - **Purpose**: The header typically tells us the type of the token, which is JWT, and the algorithm that is used for signing the token, like **HMAC SHA256 or RSA**.
   - **Example**: **{"alg": "HS256", "typ": "JWT"}**
   - **Explanation**: This is a simple JSON object that states the JWT is using the 'HS256' algorithm for encryption. It's encoded in Base64Url format to make it URL-safe.
2. **Payload**
   - **Purpose**: The payload contains the claims. Claims are statements about an entity (typically, the user) and additional data.
   - **Example**: **{"sub": "1234589670", "name": "Gaurav Sharma", "admin": true}**
   - **Explanation**: This JSON object contains information about the user and other metadata. It says that the subject (user) of the JWT has the ID "**1234589670**", the name "**Gaurav Sharma**", and has administrative rights.
3. **Signature**
   - **Purpose**: The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.
   - **How It's Made**: To create the signature, you take the encoded header, the encoded payload, a secret, the algorithm specified in the header (like HMAC SHA256), and sign that.
   - **Explanation**: For example, if you are using the HMAC SHA256 algorithm, the signature will be created by applying the HMAC SHA256 algorithm to the combined string of Base64Url encoded header and payload, using a secret key.

These three parts together form the JWT: header.payload.signature. When the JWT is sent, it appears as a string of three Base64Url-encoded parts, separated by dots, looking something like this: **eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c.**

That's essentially what a JWT is made up of and how each part functions.

💡 ***Note:*** JWTs are encoded but not encrypted, making them readable if intercepted. Therefore, avoid storing sensitive data like passwords directly in JWTs, as anyone with access can decode and extract this information.

💡 ***Note:*** Instead, JWTs should contain only the necessary information required for user identification and session management, such as user ID or username, and permissions or roles that help with access control decisions. For securing the integrity and authenticity of the token, it is digitally signed using a secret key known only to the server. Even though the contents can be viewed, the signature helps ensure that the token hasn't been altered after it was issued.

Why JWT?

One might wonder: if we can simply use a username and password to access secure endpoints in a web application, why then do we opt for using JWT with Spring Security?

Choosing JWT (JSON Web Tokens) with Spring Security instead of just using usernames and passwords has several benefits, especially for modern web applications:

1. **No Need for Constant Check-ins**: JWTs contain all the user information right inside them, so servers can verify who you are without having to ask the database every time you make a request. **This is great for handling lots of users because it cuts down on database traffic.**
2. **Extra Security**: While you still use a username and password to log in, JWT provides a more secure way to keep checking who you are after you're logged in. The tokens are protected and have **a built-in expiration time**, which helps prevent them from being misused.
3. **Works Everywhere**: JWTs are good at working across different systems and devices. This means that once you're logged in to one part of a system, you can use other parts too without having to log in again.
4. **Faster**: Because your server isn't always asking the database who you are, things can run faster. With JWT, your server just needs to check the token you send with your requests, which speeds things up.
5. **Carries More Info**: JWTs can also include extra details like what permissions you have, which helps the server know what you're allowed to do without having to look it up all the time.

In simple terms, JWT makes things more secure, quicker, and easier to manage, especially when you have lots of users or need to work across different parts of a system. This makes them a strong choice for modern web applications.

In the case of microservices, where there are multiple servers involved:

💡 JWTs (JSON Web Tokens) are extensively used in microservices architectures, where web applications are built with multiple servers handling different services. In such environments, retrieving usernames and passwords from a specific server every time a user makes a request can be inefficient and slow down the system.

💡 JWTs address this challenge by encapsulating the user's identity and authorization details within a secure token. This token is issued once, typically after the user logs in, and then used for subsequent requests across different services. This method eliminates the need to continuously query a central authentication server or database, thereby streamlining authentication processes across multiple servers and enhancing overall system performance.

In a microservices setup, where multiple small services work together, JWT (JSON Web Tokens) offer some clear benefits for managing security efficiently:

- **Independent Verification**: Each microservice can check who you are on its own using the information in the JWT. There's no need to keep asking a central server to verify your identity, which cuts down on delays.
- **Less Network Traffic**: Because JWTs carry all needed user info right inside them, there's no need for services to constantly talk to each other or to a central database to check user details. This reduces network traffic and speeds things up.
- **Works Across Services**: JWTs are great for setups that stretch over different areas or systems. Once you're logged in and have a JWT, any service within the system can recognize and trust it.
- **No Memory Needed**: JWTs help keep the system simple because services don't need to remember user sessions. Everything needed is in the JWT, which supports the stateless operation of microservices.

Overall, JWT makes security smoother, quicker, and more scalable in a microservices architecture, fitting well with the need for each service to operate independently yet securely.

Chapter-0: This article is divided into three sections: -

1. The first section involves developing a basic API to add an employee and retrieve all employees.
2. The second section focuses on implementing user-related classes in Spring Security.
3. The third section covers the integration of JWT with Spring Security and accessing the critical end points of the project using jwt in header of the request.

Chapter 1: Building APIs for Employee Management in Spring Boot:

Chapter 1 offers a basic overview of creating APIs in Spring Boot to add and retrieve employees within an employee management system.

Employee.java

```
package com.jwt.jwt.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer employeeId;
    private String name;
    private String email;
    private String department;
    private String company;

}
```

Explanation for the above code:

The **Employee** class is a Java entity model that represents an employee with attributes like ID, name, email, department, and company, and it utilizes annotations for automatic database integration and boilerplate code generation.

EmployeeRepository.java

```java
package com.jwt.jwt.repository;

import com.jwt.jwt.model.Employee;
import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
}
```

Explanation for the above code:

The **EmployeeRepository** interface provides automatic CRUD operation methods for Employee objects using Spring Data JPA.

EmployeeService.java

```java
package com.jwt.jwt.services;

import org.springframework.stereotype.Service;
import com.jwt.jwt.model.Employee;
import com.jwt.jwt.repository.EmployeeRepository;
import java.util.List;

@Service
public class EmployeeService {

    private final EmployeeRepository repository;

    public EmployeeService(EmployeeRepository repository) {
        this.repository = repository;
    }

    public List<Employee> findAll() {
        return repository.findAll();
    }

    public Employee save(Employee employee) {
        return repository.save(employee);
    }
}
```

Explanation for the above code:

The **EmployeeService** class provides methods to manage employee data, including retrieving all employees and saving a new or updated employee, by utilizing a repository layer.

EmployeeController.java

```java
package com.jwt.jwt.controllers;

import org.springframework.web.bind.annotation.*;
import com.jwt.jwt.model.Employee;
import com.jwt.jwt.services.EmployeeService;
import java.util.List;

@RestController
@RequestMapping("/api")
public class EmployeeController {

    private final EmployeeService service;

    public EmployeeController(EmployeeService service) {
        this.service = service;
    }

    @GetMapping("/employees")
    public List<Employee> getAllEmployees() {
        return service.findAll();
    }

    @PostMapping("/employee")
    public Employee createEmployee(@RequestBody Employee employee) {
        return service.save(employee);
    }
}
```

Explanation for the above code:

The **EmployeeController** class provides web endpoints to retrieve all employees and create a new employee through HTTP GET and POST requests, respectively, using the **EmployeeService**.

💡 Our application is configured to automatically populate the database with initial data at startup, utilizing SQL files named **schema.sql** and **data.sql**.

schema.sql

```sql
CREATE TABLE IF NOT EXISTS employee (
                                        employee_id SERIAL PRIMARY KEY,
                                        name VARCHAR(255),
    email VARCHAR(255),
    department VARCHAR(255),
    company VARCHAR(255)
    );
```

Explanation for the above code:

This SQL statement creates a new table named **employee** with columns for employee ID, name, email, department, and company, ensuring the table only exists if it hasn't been defined previously.

```
data.sql
```

```sql
INSERT INTO Employee (name, email, department, company) VALUES ('Aarav Kumar',
'aarav.kumar@example.com', 'HR', 'Tech Innovations Pvt Ltd');
INSERT INTO Employee (name, email, department, company) VALUES ('Diya Sharma',
'diya.sharma@example.com', 'Marketing', 'Creative Minds Ltd');
INSERT INTO Employee (name, email, department, company) VALUES ('Rohan Gupta',
'rohan.gupta@example.com', 'Finance', 'Financial Solutions Inc');
INSERT INTO Employee (name, email, department, company) VALUES ('Isha Patel',
'isha.patel@example.com', 'IT', 'Tech Solutions Pvt Ltd');
INSERT INTO Employee (name, email, department, company) VALUES ('Aditya Singh',
'aditya.singh@example.com', 'Operations', 'Manufacturing Corp');
```

Explanation for the above code:

These SQL commands insert records into the **Employee** table, adding new employees with specified names, emails, departments, and companies.

```
application.yml
```

```yaml
server:
  port: 8081      # Local Port number to run the application
spring:
  datasource:
    url:
${SPRING_DATASOURCE_URL:jdbc:oracle:thin:@//localhost:1521/XE}     #SPRING_DATASOURCE_URL
which is defined in docker-compose file
    username:
${SPRING_DATASOURCE_USERNAME:system}                          #SPRING_DATASOURCE_USERNAME
which is defined in docker-compose file
    password:
${SPRING_DATASOURCE_PASSWORD:root}                            #SPRING_DATASOURCE_PASSWORD
which is defined in docker-compose file
    driver-class-name: ${SPRING_DATASOURCE_DRIVER:oracle.jdbc.OracleDriver}
#SPRING_DATASOURCE_DRIVER which is defined in docker-compose file
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    properties:
      hibernate:
        dialect: org.hibernate.dialect.OracleDialect
```

Chapter 2: Implementing User related Classes in Spring Security: -

💡 **Note**: Between Chapters 2 and 3, I've inserted brief comments throughout the code for better clarity, as integrating JWT with Spring Boot and Spring Security is a heavy lifting task.

To set up our security system, we need to create a user class that includes fields such as username and password. This allows us to store user information in the database and authenticate users based on these credentials.

However, there's an important aspect to note: Spring Security does not automatically recognize this custom user class. Instead, it works with its predefined **UserDetails** interface.

In simple terms, **UserDetails** is a special interface in Spring Security designed to handle user information in a way that Spring Security can understand. This means that for Spring Security to work with our custom user class, we need to adapt our class to fit this interface. Essentially, we need to convert our user class into one that implements the **UserDetails** interface.

```
User.java


package com.jwt.jwt.model;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * The UserInfo class represents a user entity in the application.
 * It is used to store user-related data such as name, password etc.
 */
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String userName;
    private String password;

}
```

Explanation for the above code:

This code sets up a simple User class to store user information in a database, specifically their ID, username, and password.

implementing the **UserDetails** interface provided by spring security:

UserPrincipal.java

```java
package com.jwt.jwt.model;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import java.util.Collection;
import java.util.Collections;
import java.util.Set;

public class UserPrincipal implements UserDetails {

    private static final long serialVersionUID = 1L;

    String userName = null;
    String password = null;
    Set<SimpleGrantedAuthority> authorities;

    public UserPrincipal(User user) {
        userName = user.getUserName();
        password = user.getPassword();
        authorities = Collections.singleton(new SimpleGrantedAuthority("USER"));
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public String getUsername() {
        return userName;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }
    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

Explanation of the above code:

The **UserPrincipal** class in the code is a custom implementation of the **UserDetails** interface provided by Spring Security. This class is used to integrate your application's user entities with Spring Security's authentication and authorization mechanisms.

Here's a breakdown of its functionality:

- **Fields**: The class has three main fields - **userName**, **password**, and **authorities**. These represent the user's login credentials and their roles or permissions respectively.
- **Constructor**: The **UserPrincipal** constructor takes a **User** object as an argument. It extracts the username and password from this user object, and initializes the user's authorities.
- **getAuthorities()**: This method specifies the roles or authorities granted to the user. In this case, every user is given a single authority of "USER".
- **getPassword() and getUsername()**: These methods simply retrieve the password and username from the **User** instance, respectively.
- **Account Status Methods**: The methods **isAccountNonExpired()**, **isAccountNonLocked()**, **isCredentialsNonExpired()**, and **isEnabled()** are all overridden to return **true**. These methods are used by Spring Security to determine if the account is still active, locked, has expired credentials, or is enabled. Returning **true** from all these methods suggests that in this simple implementation, these checks are not being used to restrict user access.

UserRepo.java

```
package com.jwt.jwt.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.jwt.jwt.model.User;
import java.util.Optional;


@Repository
public interface UserRepo extends JpaRepository<User, Integer> {
    /**
     * findByName method is used to retrieve a user by their username.
     * It returns an Optional of UserInfo, which will be empty if no user is found.
     */
    Optional<User> findByUserName(String userName);
}
```

Explanation for the above code:

The findByUsername(String username) method in the UserRepo interface is a specialized function that lets you find and retrieve a User based on their username.

UserService.java

```java
package com.jwt.jwt.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import com.jwt.jwt.model.User;
import com.jwt.jwt.model.UserPrincipal;
import com.jwt.jwt.repository.UserRepo;

import java.util.Optional;

/**
 * UserInfoService provides user-related services including loading user details
 * and managing user data in the repository.
 */
@Service
public class UserService implements UserDetailsService {

    @Autowired
    private UserRepo userRepo;

    @Autowired
    private PasswordEncoder passwordEncoder;


    // Loads a user's details given their userName.
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Optional<User> user = userRepo.findByUserName(username);
        return user.map(UserPrincipal::new)
                .orElseThrow(() -> new UsernameNotFoundException("UserName not found: " +
username));
    }


    // Adds a new user to the repository and encrypting password before saving it.
    public String addUser(User user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        userRepo.save(user);
        return "user added successfully";
    }
}
```

Explanation for the above code:

The **UserService** class implements the **UserDetailsService** interface from Spring Security, providing a method to load user details from the database. This ensures that the class works seamlessly with Spring Security's authentication processes.

The **UserService** class is responsible for handling user information within a Spring Security framework. It includes two main methods:

1. **loadUserByUsername**: This method retrieves user details from the database using the provided username. If the user exists, it returns their details wrapped in a **UserPrincipal** object for Spring Security's authentication processes. If the username isn't found, it throws an exception.
2. **addUser**: This method takes a **User** object, encrypts the user's password for security using the **PasswordEncoder**, and saves the updated user to the database. It confirms the addition with a success message.

UserController.java

```java
package com.jwt.jwt.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.jwt.jwt.model.User;
import com.jwt.jwt.services.JwtService;
import com.jwt.jwt.services.UserService;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;


@RestController
@RequestMapping("/auth")
public class UserController {

    @Autowired
    private UserService userService;

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtService jwtService;

    @PostMapping("/addUser")
    public String addUser(@RequestBody User user) {
       return userService.addUser(user);
    }

    @PostMapping("/login")
    public String loginUser(@RequestBody User user) {

        Authentication authenticate = authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(user.getUserName(), user.getPassword()));
        if(authenticate.isAuthenticated()) {
            return jwtService.generateToken(user.getUserName());
        }else{
            throw new UsernameNotFoundException("Invalid username or password");
        }
    }
}
```

Explanation for the above code:

//ToDo

Chapter-3:Integrating JWT with Spring Security

To implement JSON Web Tokens (JWT) for authorization in a Spring Boot Maven project, we need to include specific dependencies in our **pom.xml** file.

```xml
 <dependency>
            <groupId>io.jsonwebtoken</groupId>
            <artifactId>jjwt-jackson</artifactId>
            <version>0.11.5</version>
        </dependency>
        <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>
```

Explanation for the above dependencies:

1. jjwt-jackson

- **What it does**:Integrates the Jackson library with JWT operations, optimizing how your Java application encodes and decodes JSON data within JWTs.

2. jjwt-api

- **What it does**: Supplies the fundamental classes and interfaces for constructing and validating JWTs, providing the building blocks for JWT functionality in Java.

3. jjwt-impl

- **What it does**: Provides the actual implementation for the interfaces from **jjwt-api**, enabling the generation, parsing, and management of JWTs according to your application's security protocols.

Our final pom.xml file should look something like this:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.3.7</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.jwt</groupId>
    <artifactId>jwt</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>jwt</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <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>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>com.oracle.database.jdbc</groupId>
            <artifactId>ojdbc11</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>

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


    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

each dependency included in our **pom.xml** file:

- **Spring Boot Starter Data JPA**: Integrates Spring Data JPA with Spring Boot to simplify database interactions using JPA repositories.
- **Spring Boot Starter Security**: Adds security features like authentication and authorization to your Spring Boot application.
- **Spring Boot Starter Web**: Provides all the necessities for building web applications, including RESTful applications, using Spring MVC.
- **Spring Boot DevTools**: Offers tools for automatic restarts and live reload capabilities to enhance developer productivity.
- **PostgreSQL Driver**: Enables JDBC connectivity to PostgreSQL databases, allowing for data transactions between the app and the database.
- **Lombok**: Facilitates reducing boilerplate code in Java applications by auto-generating getters, setters, constructors, and more.
- **jjwt-jackson**: Provides Jackson JSON processing capabilities to the JJWT library for handling JSON web tokens.
- **jjwt-api**: Offers API support for creating and verifying JSON Web Tokens (JWTs) efficiently.
- **jjwt-impl**: Implements the JJWT API, providing the necessary code to manage JWTs.
-

Now, we will define classes that will allow us to integrart jwt into our security mechanism.

💡 The **JwtService** class **creates, checks, and validates security tokens (JWTs)** to help confirm user identities in an application.

```
JwtService.java


package com.jwt.jwt.services;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;


//JwtService is responsible for handling JWT (JSON Web Token) operations
// such as token generation, extraction of claims, and token validation.

@Component
public class JwtService {

    // Secret Key for signing the JWT. It should be kept private.
    private static final String SECRET =
"TmV3U2VjcmV0S2V5Rm9ySldUU2lnbmluZ1B1cnBvc2VzMTIzNDU2Nzg=\r\n" + "";


    // Generates a JWT token for the given userName.
    public String generateToken(String userName) {
        // Prepare claims for the token
        Map<String, Object> claims = new HashMap<>();

        // Build JWT token with claims, subject, issued time, expiration time, and signing
algorithm
          // Token valid for 3 minutes
        return Jwts.builder()
                .setClaims(claims)
                .setSubject(userName)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 3))
                .signWith(getSignKey(), SignatureAlgorithm.HS256).compact();
    }


    // Creates a signing key from the base64 encoded secret.
    //returns a Key object for signing the JWT.
    private Key getSignKey() {
        // Decode the base64 encoded secret key and return a Key object
        byte[] keyBytes = Decoders.BASE64.decode(SECRET);
        return Keys.hmacShaKeyFor(keyBytes);
    }


    // Extracts the userName from the JWT token.
    //return -> The userName contained in the token.
    public String extractUserName(String token) {
        // Extract and return the subject claim from the token
        return extractClaim(token, Claims::getSubject);
    }


    // Extracts the expiration date from the JWT token.
    //@return The expiration date of the token.
    public Date extractExpiration(String token) {
        // Extract and return the expiration claim from the token
        return extractClaim(token, Claims::getExpiration);
    }


    // Extracts a specific claim from the JWT token.
    // claimResolver A function to extract the claim.
    // return-> The value of the specified claim.
    private <T> T extractClaim(String token, Function<Claims, T> claimResolver) {
        // Extract the specified claim using the provided function
        final Claims claims = extractAllClaims(token);
        return claimResolver.apply(claims);
    }

    //Extracts all claims from the JWT token.
    //return-> Claims object containing all claims.
```

```java
    private Claims extractAllClaims(String token) {
        // Parse and return all claims from the token
        return Jwts.parserBuilder()
                .setSigningKey(getSignKey())
                .build().parseClaimsJws(token).getBody();
    }


    //Checks if the JWT token is expired.
    //return-> True if the token is expired, false otherwise.
    public Boolean isTokenExpired(String token) {
        // Check if the token's expiration time is before the current time
        return extractExpiration(token).before(new Date());
    }

    //Validates the JWT token against the UserDetails.
    //return-> True if the token is valid, false otherwise.

    public Boolean validateToken(String token, UserDetails userDetails) {
        // Extract username from token and check if it matches UserDetails' username
        final String userName = extractUserName(token);
        // Also check if the token is expired
        return (userName.equals(userDetails.getUsername()) && !isTokenExpired(token));
    }
}
```

key methods in the **JwtService** class:

1. **generateToken:** Makes a secure token for a user that includes their username and how long the token is good for.

2. **getSignKey:** Creates a special key from a secret key to help keep the token safe.

💡 The secret key in the JwtService class acts like a digital signature for securing JSON Web Tokens (JWTs). This key is encoded in base64, a method that transforms the key into a string format that's easier to handle in programming environments. **Its primary role is to ensure that the data within the JWT hasn't been altered by unauthorized parties.** It is crucial to keep this key confidential; if it falls into the wrong hands, someone could create counterfeit tokens, potentially gaining unauthorized access to user data and privileged system functions. This makes maintaining the secrecy of this key vital for the security of your application.

> ***A little demo in case you want to generate your own secret key***: asecuritysite.com
>
> **Demo for the same: YouTube: https://youtu.be/Y9qd10o9Rfo**

3. **extractUserName:** Gets the username from a token.

4. **extractExpiration:** Finds out when the token will stop working.

5. **extractClaim:** The **extractClaim** method in the JwtService class uses a function passed as a parameter to retrieve a specific claim, such as the username or expiration date, from a parsed JWT token.

6. **isTokenExpired:** Checks if the token has run out of time and is no longer valid.

7. **validateToken:** Ensures the token is still good and matches the right user, confirming it can be used safely for logging in.

💡 The **JwtFilter** class checks every incoming request to make sure the user has a valid login token and sets up their login status if everything checks out.

JwtFilter.java

```java
package com.jwt.jwt.filter;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import com.jwt.jwt.services.JwtService;
import com.jwt.jwt.services.UserService;

import java.io.IOException;

@Component
public class JwtFilter extends OncePerRequestFilter {

    @Autowired
    private JwtService jwtService;

    @Autowired
    private ApplicationContext applicationContext;

    // Method to lazily fetch the UserService bean from the ApplicationContext
    // This is done to avoid Circular Dependency issues
    private UserService getUserService() {
        return applicationContext.getBean(UserService.class);
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain) throws ServletException, IOException {
        // Extracting token from the request header
        String authHeader = request.getHeader("Authorization");
        String token = null;
        String userName = null;

        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            // Extracting the token from the Authorization header
            token = authHeader.substring(7);
            // Extracting username from the token
            userName = jwtService.extractUserName(token);
        }
        // If username is extracted and there is no authentication in the current
SecurityContext
        if (userName != null && SecurityContextHolder.getContext().getAuthentication() ==
null) {
            // Loading UserDetails by username extracted from the token
            UserDetails userDetails = getUserService().loadUserByUsername(userName);

            // Validating the token with loaded UserDetails
            if (jwtService.validateToken(token, userDetails)) {
                // Creating an authentication token using UserDetails
                UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
                // Setting authentication details
                authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
                // Setting the authentication token in the SecurityContext
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }

        // Proceeding with the filter chain
        filterChain.doFilter(request, response);
    }
}
```

each method in the **JwtFilter** class:

1. **getUserService:** The getUserService method in the JwtFilter class is designed to retrieve the UserService instance from the Spring application context only when it's needed. This approach is used to prevent problems related to the order in which Spring beans are loaded, known as circular dependency issues. Essentially, it ensures that UserService is available and fully initialized when accessed, avoiding any startup errors due to dependency conflicts.

2. **doFilterInternal:**

    - **Purpose:** Examines each incoming HTTP request for a JWT in the Authorization header.
    - **Process:** If a valid JWT is found, it extracts the username and checks with SecurityContextHolder to determine if the user is already authenticated for the current session.
    - **User Authentication:** If the user is not logged in, it retrieves user details, verifies the token against these details, and if validated, sets the user's authentication in the SecurityContextHolder.
    - **Session Management:** This mechanism centralizes security information, facilitating easy access and management of user authentication and authorization across the application.
    - **Continuation:** Allows the request to move forward to the next stage in the filter chain or the intended final destination.

💡 **SecurityContextHolder:** This is a part of Spring Security that holds the security context (i.e., details about the current user and their permissions) for the current thread. It is used throughout an application to grant or restrict access to different parts of the system.

💡 The SecurityConfig.java class sets up security rules for a web application to ensure that only authorized users can access specific areas.

SecurityConfig.java

```java
package com.jwt.jwt.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationCon
figuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.web.bind.annotation.CrossOrigin;
import com.jwt.jwt.filter.JwtFilter;
import com.jwt.jwt.services.UserService;

@Configuration
@EnableWebSecurity
@CrossOrigin
public class SecurityConfig {

    @Autowired
    private JwtFilter jwtFilter;

    // Defines a UserDetailsService bean for user authentication
    @Bean
    public UserDetailsService userDetailsService() {
        return new UserService();
    }


    // Configures the security filter chain
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws
Exception {
        return httpSecurity
                .cors(Customizer.withDefaults()) // Apply CORS
                .csrf(csrf -> csrf.disable()) // Disable CSRF protection
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/auth/addUser", "/auth/login")
                        .permitAll()// Permit all requests to certain URLs
                        .anyRequest().authenticated()) // Require authentication for all
other requests
                .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // Set session management to
stateless
                .authenticationProvider(authenticationProvider()) // Register the
authentication provider
                .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class) //
Add the JWT filter before processing the request
                .build();
    }


    // Creates a DaoAuthenticationProvider to handle user authentication
    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(userDetailsService());
        authenticationProvider.setPasswordEncoder(passwordEncoder());
        return authenticationProvider;
    }


    // Defines a PasswordEncoder bean that uses bcrypt hashing by default for password
encoding
   @Bean
    PasswordEncoder passwordEncoder() {
        return PasswordEncoderFactories.createDelegatingPasswordEncoder();
    }


    // Defines an AuthenticationManager bean to manage authentication processes
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config)
throws Exception {
        return config.getAuthenticationManager();
    }

}
```

Explantion for the above code:

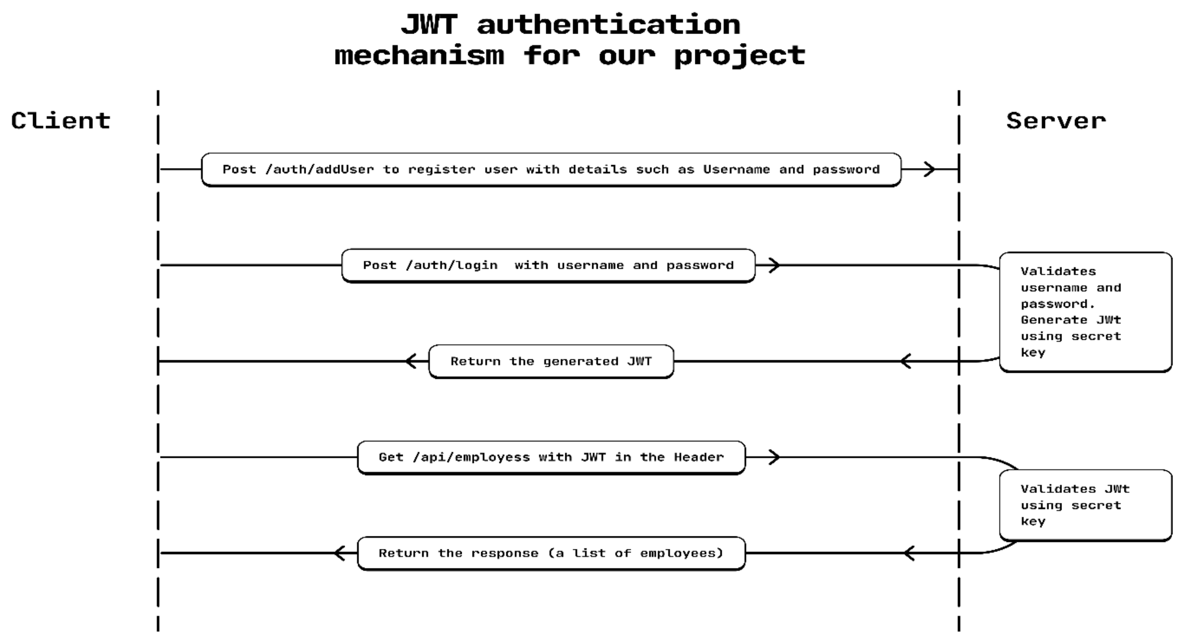
each method defined in the SecurityConfig class:

1. **userDetailsService()**:
   - Initializes a service for loading user-specific data. It connects the application to a user service that retrieves necessary user information for authentication.
2. **securityFilterChain(HttpSecurity httpSecurity)**:
   - Configures security policies for HTTP requests within the application. It sets up how requests are authenticated and authorized. Key configurations include:
     - **CORS Configuration**: Applies default settings for Cross-Origin Resource Sharing to manage resource interaction between different origins.
     - **CSRF Protection**: Disables Cross-Site Request Forgery protection, typically for APIs using token-based authentication instead of cookies.

- **Authorization Rules**: Defines access rules, allowing unrestricted access to certain paths like **/auth/addUser** and **/auth/login**, and requiring authentication for all other requests.
- **Session Management**: Sets sessions to be stateless, meaning the server does not maintain any session state between requests.
- **Authentication Provider**: Integrates a **DaoAuthenticationProvider** that uses a user details service and a password encoder to authenticate users.
- **JWT Filter Integration**: Incorporates a JWT filter before the standard username-password authentication filter to process and validate JWTs in requests.

3. **authenticationProvider()**:
   - Creates an authentication provider that uses a database to handle user authentication. This provider retrieves user details and verifies passwords using a specified encoder.
4. **passwordEncoder()**:
   - Establishes a method for password encoding using BCrypt by default, an algorithm that hashes passwords securely before they are stored.
5. **authenticationManager(AuthenticationConfiguration config)**:
   - Provides a manager to see the authentication process, crucial for processing authentication requests and central to Spring Security's authentication framework.

JWT Authentication Mechanism for Our Project


JWT authentication mechanism for our project
Client
Server
Post /auth/addUser to register user with details such as Username and password
Post /auth/login with username and password
Validates username and password. Generate JWt using secret key
Return the generated JWT
Get /api/employess with JWT in the Header
Validates JWt using secret key
Return the response (a list of employees)

Demo: YouTube Video: https://youtu.be/8qAWCkLJEvg

Understanding JWT Tokens Issued After User Login: YouTube Video: https://youtu.be/ia_3UG32NnA