SpringBoot Security Module

Spring Security:

Spring Security is a powerful and highly customizable authentication and access-control framework. Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements. It overcomes all the problems that come during creating non spring security applications and manage new server environment for the application. The main goal of Spring Security is to make it easy to add security features to your applications. It follows a modular design, allowing you to choose and configure various components according to your specific requirements. Some of the key features of Spring Security include:

Authentication: Spring Security supports multiple authentication mechanisms, such as form-based, HTTP Basic, HTTP Digest, and more. It integrates seamlessly with various authentication providers, including in-memory, JDBC, LDAP, and OAuth.

Authorization: Spring Security enables fine-grained authorization control based on roles, permissions, and other attributes. It provides declarative and programmatic approaches for securing application resources, such as URLs, methods, and domain objects.

Session Management: Spring Security offers session management capabilities, including session fixation protection, session concurrency control, and session timeout handling. It allows you to configure session-related properties and customize session management behaviour.

Security Filters: Spring Security leverages servlet filters to intercept and process incoming requests. It includes a set of predefined filters for common security tasks, such as authentication, authorization, and request/response manipulation. You can easily configure and extend these filters to meet your specific needs.

Integration with Spring Framework: Spring Security seamlessly integrates with the Spring ecosystem. It can leverage dependency injection and aspect-oriented programming features provided by the Spring Framework to enhance security functionality.

Customization and Extension: Spring Security is highly customizable, allowing you to override default configurations, implement custom authentication/authorization logic, and integrate with third-party libraries or existing security infrastructure.

Overall, Spring Security simplifies the process of implementing robust security features in Java applications. It provides a flexible and modular framework that addresses common security concerns and allows developers to focus on building secure applications.

This module targets two major areas of application are **authentication** and **authorization**.

What is Authentication?

Authentication in Spring refers to the process of verifying the identity of a user or client accessing a system or application. It is a crucial aspect of building secure applications to ensure that only authorized individuals can access protected resources.

In the context of Spring Security, authentication involves validating the credentials provided by the user and establishing their identity. Spring Security offers various authentication mechanisms and supports integration with different authentication providers.

Here's a high-level overview of how authentication works in Spring Security:

User provides credentials: The user typically provides credentials, such as a username and password, in order to authenticate themselves.

Authentication request: The application receives the user's credentials and creates an authentication request object.

Authentication manager: The authentication request is passed to the authentication manager, which is responsible for validating the credentials and performing the authentication process.

Authentication provider: The authentication manager delegates the actual authentication process to one or more authentication providers. An authentication provider is responsible for verifying the user's credentials against a specific authentication mechanism, such as a user database, LDAP server, or OAuth provider.

Authentication result: The authentication provider returns an authentication result, indicating whether the user's credentials were successfully authenticated or not. If successful, the result typically contains the authenticated user details, such as the username and granted authorities.

Security context: If the authentication is successful, Spring Security establishes a security context for the authenticated user. The security context holds the user's authentication details and is associated with the current thread.

Access control: With the user authenticated, Spring Security can enforce access control policies based on the user's granted authorities or other attributes. This allows the application to restrict access to certain resources or operations based on the user's role or permissions.

Spring Security provides several authentication mechanisms out-of-the-box, including form-based authentication, HTTP Basic/Digest authentication, JWT token, OAuth-based authentication. Spring also supports customization and extension, allowing you to integrate with your own authentication providers or implement custom authentication logic to meet your specific requirements.

By integrating Spring Security's authentication capabilities into your application, you can ensure that only authenticated and authorized users have access to your protected resources, helping to safeguard your application against unauthorized access.

What is Authorization?

Authorization, also known as access control, is the process of determining what actions or resources a user or client is allowed to access within a system or application. It involves enforcing permissions and restrictions based on the user's identity, role, or other attributes. Once a user is authenticated, authorization is used to control their access to different parts of the application and its resources.

Here are the key concepts related to authorization in Spring Security:

Roles: Roles represent a set of permissions or privileges granted to a user. They define the user's high-level responsibilities or functional areas within the application. For example, an application may have roles such as "admin," "user," or "manager."

Permissions: Permissions are specific actions or operations that a user is allowed to perform. They define the granular level of access control within the application. For example, a user with the "admin" role may have permissions to create, update, and delete resources, while a user with the "user" role may only have read permissions.

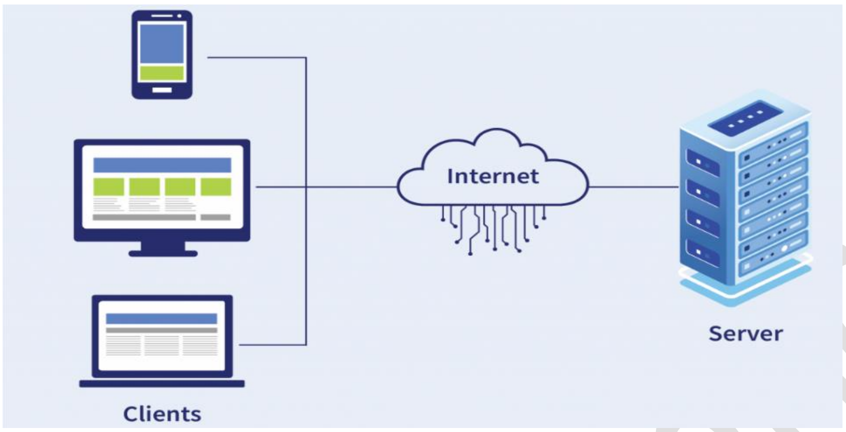
Security Interceptors: Spring Security uses security interceptors to enforce authorization rules. These interceptors are responsible for intercepting requests and checking whether the user has the required permissions to access the requested resource. They can be configured to protect URLs, methods, or other parts of the application.

Role-Based Access Control (RBAC): RBAC is a common authorization model in which access control is based on roles. Users are assigned roles, and permissions are associated with those roles. Spring Security supports RBAC by allowing you to define roles and assign them to users.

By implementing authorization in your Spring application using Spring Security, you can ensure that users have appropriate access privileges based on their roles and permissions. This helps protect sensitive resources and data from unauthorized access and maintain the overall security and integrity of your application.

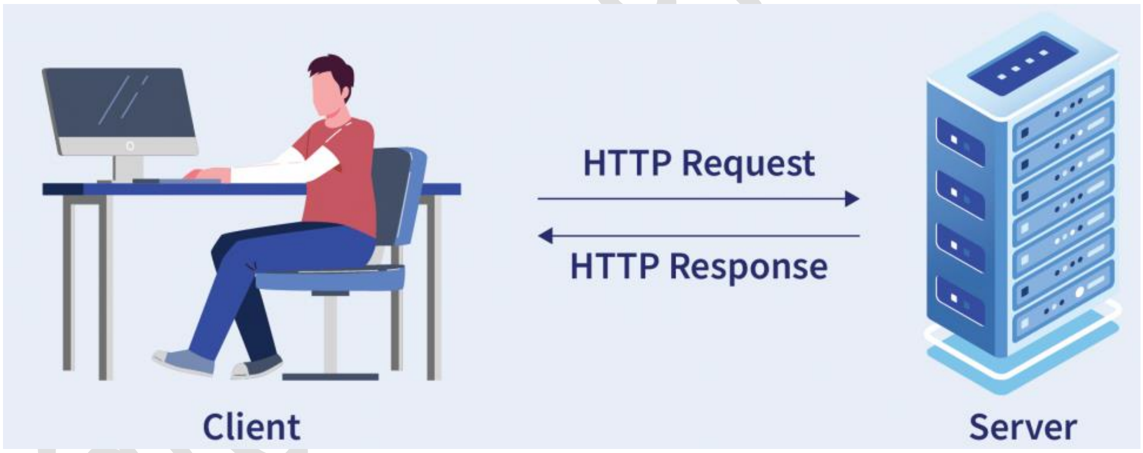
Stateless and Stateful Protocols:

In the context of the protocol, "stateless" and "stateful" refer to different approaches in handling client-server interactions and maintaining session information. Let's explore each concept:



Stateless:

In a stateless protocol, such as HTTP, the server does not maintain any information about the client's previous interactions or session state. Each request from the client to the server is considered independent and self-contained. The server treats each request as if it is the first request from the client.

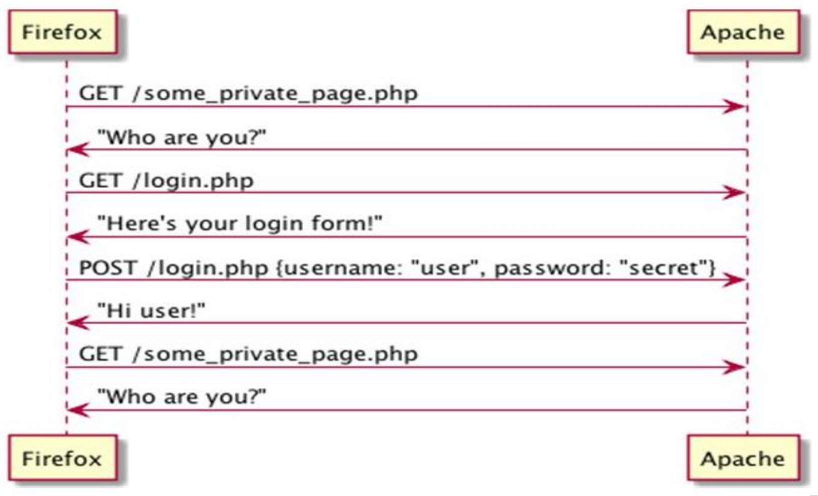


Stateless protocols have the following characteristics:

- No client session information is stored on the server.
- Each request from the client must contain all the necessary information for the server to process the request.
- The server responds to each request independently, without relying on any prior request context.

HTTP is primarily designed as a stateless protocol. When a client makes an HTTP request, the server processes the request and sends back a response. However, the server does not maintain any information about the client after the response is sent. This approach simplifies the server's implementation and scalability but presents challenges for handling user sessions and maintaining continuity between multiple requests.

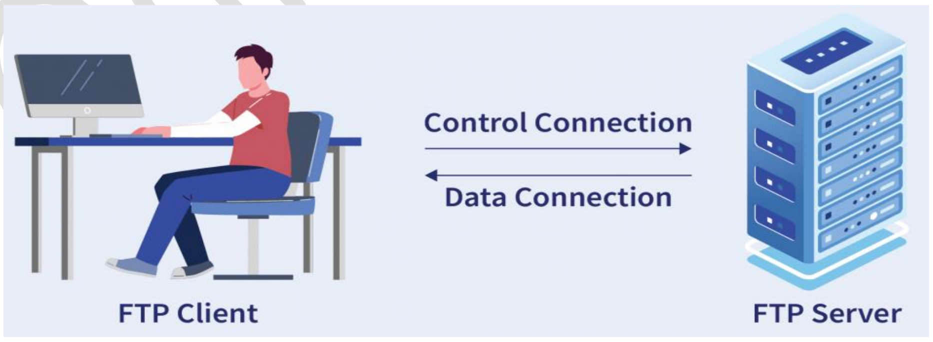
Stateless Protocol (Technical)


Firefox
Apache
GET /some_private_page.php
"Who are you?"
GET /login.php
"Here's your login form!"
POST /login.php {username: "user", password: "secret"}
"Hi user!"
GET /some_private_page.php
"Who are you?"
Firefox
Apache

Stateful:

In contrast, a stateful protocol maintains information about the client's interactions and session state between requests. The server stores client-specific information and uses it to provide personalized responses and maintain continuity across multiple requests.

However, the major feature of stateful is that it maintains the state of all its sessions, be it an authentication session, or a client's request for information. Stateful are those that may be used repeatedly, such as online banking or email. They're carried out in the context of prior transactions in which the states are stored, and what happened in previous transactions may have an impact on the current transaction. Because of this, stateful apps use the same servers every time they perform a user request. An example of stateful is FTP (File Transfer Protocol) i.e. File transferring between servers. For FTP session, which often includes many data transfers, the client establishes a Control Connection. After this, the data transfer takes place.



Stateful protocols have the following characteristics:

- The server keeps track of client session information, typically using a session identifier

- The session information is stored on the server.
- The server uses the session information to maintain context between requests and responses.

While HTTP itself is stateless, developers often implement mechanisms to introduce statefulness. For example, web applications often use cookies or tokens to maintain session state. These cookies or tokens contain session identifiers that the server can use to retrieve or store client-specific data.

By introducing statefulness, web applications can provide a more personalized and interactive experience for users. However, it adds complexity to the server-side implementation and may require additional considerations for scalability and session management.

It's important to note that even when stateful mechanisms are introduced, each individual HTTP request-response cycle is still stateless in nature. The statefulness is achieved by maintaining session information outside the core HTTP protocol, typically through additional mechanisms like cookies, tokens, or server-side session stores.

Q&A:

What is the difference between stateful and stateless?

The major difference between stateful and stateless is whether or not they store data regarding their sessions, and how they respond to requests. Stateful services keep track of sessions or transactions and respond to the same inputs in different ways depending on their history. Clients maintain sessions for stateless services, which are focused on activities that manipulate resources rather than the state.

Is stateless better than stateful?

In most cases, stateless is a better option when compared with stateful. However, in the end, it all comes down to your requirements. If you only require information in a transient, rapid, and temporary manner, stateless is the way to go. Stateful, on the other hand, might be the way to go if your app requires more memory of what happens from one session to the next.

Is HTTP stateful or stateless?

HTTP is stateless because it doesn’t keep track of any state information. In HTTP, each order or request is carried out in its own right, with no awareness of the demands that came before it.

Is REST API stateless or stateful?

REST APIs are stateless because, rather than relying on the server remembering previous requests, REST applications require each request to contain all of the information necessary for the server to understand it. Storing session state on the server violates the REST architecture’s stateless requirement. As a result, the client must handle the complete session state.

Security Implementation:

Stateless Security and **Stateful Security** are two approaches to handling security in systems, particularly in the context of web applications. Let's explore the differences between these two approaches:

Stateless Security:

Stateless security refers to a security approach where the server does not maintain any session state or client-specific information between requests. It is often associated with stateless protocols, such as HTTP, where each request is independent and self-contained. Stateless security is designed to provide security measures without relying on server-side session state.

In the context of web applications and APIs, stateless security is commonly implemented using mechanisms such as JSON Web Tokens (JWT) or OAuth 2.0 authentication scheme. These mechanisms allow authentication and authorization to be performed without the need for server-side session storage.

Here are the key characteristics and advantages of stateless security:

- **No server-side session storage:** With stateless security, the server does not need to maintain any session-specific information for each client. This eliminates the need for server-side session storage, reducing the overall complexity and resource requirements on the server side.
- **Scalability:** Stateless security simplifies server-side scaling as there is no need to replicate session state across multiple instances of application deployed to multiple servers. Each server can process any request independently, which makes it easier to distribute the load and scale horizontally.
- **Decentralized authentication:** Stateless security allows for decentralized authentication, where the client sends authentication credentials (such as a JWT token) with each request. The server can then validate the token's authenticity and extract necessary information to authorize the request.
- **Improved performance:** Without the need to perform expensive operations like session lookups or database queries for session data, stateless security can lead to improved performance. Each request carries the necessary authentication and authorization information, reducing the need for additional server-side operations.

It's important to note that while stateless security simplifies server-side architecture and offers advantages in terms of scalability and performance, it also places additional responsibilities on the client-side. The client must securely store and transmit the authentication token and include it in each request.

Stateless security is widely adopted in modern web application development, especially in distributed systems and microservices architectures, where scalability, performance, and decentralized authentication are important considerations.

In stateless security:

- **Authentication:** The client provides credentials (e.g., username and password or a token) with each request to prove its identity. The server verifies the credentials and grants access based on the provided information.
- **Authorization:** The server evaluates each request independently, checking if the user has the necessary permissions to access the requested resource.

Cons of Stateless Security:

- **Increased overhead:** The client needs to send authentication information with each request, which can increase network overhead, especially when the authentication mechanism involves expensive cryptographic operations.

Stateful Security:

Stateful security involves maintaining session state on the server. Once the client is authenticated, the server stores session information and associates it with the client. The server refers to the session state to validate subsequent requests and provide appropriate authorization.

In stateful security:

- **Authentication:** The client typically authenticates itself once using its credentials (e.g., username and password or token). After successful authentication, the server generates a session identifier or token and stores it on the server.

Session Management: The server maintains session-specific data, such as user roles, permissions, and other contextual information. The session state is referenced for subsequent requests to determine the user's authorization level.

Pros of Stateful Security:

- **Enhanced session management:** Session state allows the server to maintain user context, which can be beneficial for handling complex interactions and personalized experiences.
- **Reduced overhead:** Since the client doesn't need to send authentication information with each request, there is a reduction in network overhead.

Cons of Stateful Security:

- **Scalability challenges:** The server needs to manage session state, which can be a scalability bottleneck. Sharing session state across multiple servers or implementing session replication techniques becomes necessary.
- **Complexity:** Implementing stateful security requires additional effort to manage session state and ensure consistency across requests.

The choice between stateless security and stateful security depends on various factors, including the specific requirements of the application, performance considerations, and the desired level of session management and personalization. Stateless security is often preferred for its simplicity and scalability advantages, while stateful security is suitable for scenarios requiring more advanced session management capabilities.

JWT Authentication & Authorization:

JWTs or JSON Web Tokens are most commonly used to identify an authenticated user. They are issued by an authentication server and are consumed by the client-server (to secure its APIs).

What is a JWT?

JSON Web Token is an open industry standard used to share information between two entities, usually a client (like your app's frontend) and a server (your app's backend). They contain JSON objects which have the information that needs to be shared. Each JWT is also signed using cryptography (hashing) to ensure that the JSON contents (also known as JWT claims) cannot be altered by the client or a malicious party.

A token is a string that contains some information that can be verified securely. It could be a random set of alphanumeric characters which point to an ID in the database, or it could be an encoded JSON that can be self-verified by the client (known as JWTs).

Structure of a JWT:

A JWT contains three parts:

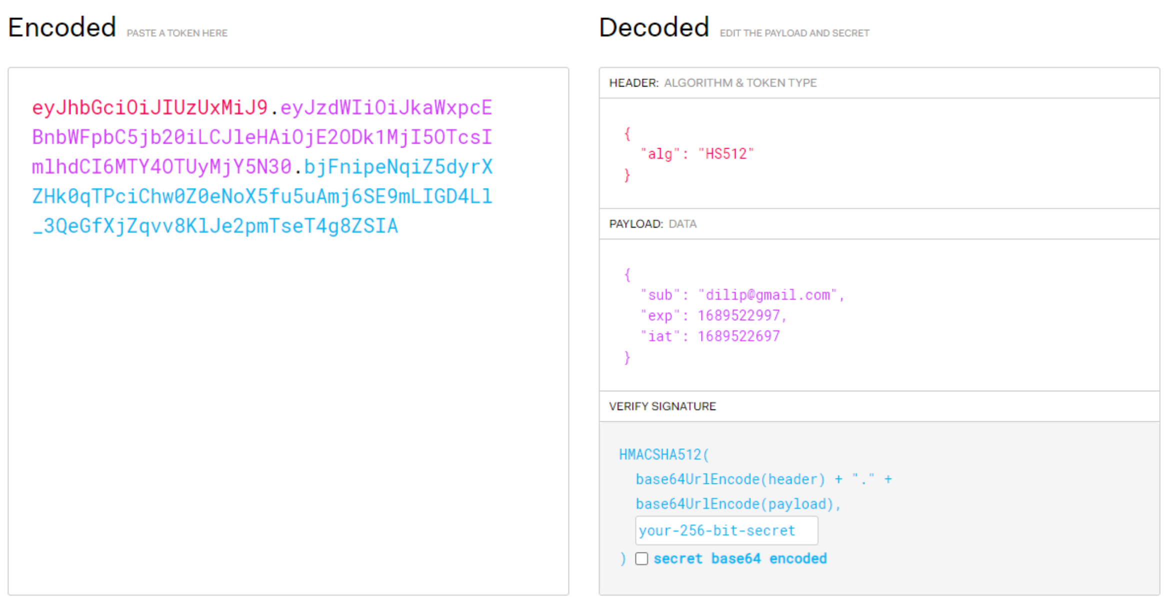
- **Header**: Consists of two parts:
  - The signing algorithm that's being used.
  - The type of token, which, in this case, is mostly "JWT".
- **Payload**: The payload contains the claims or the JSON object of clients.
- **Signature**: A string that is generated via a cryptographic algorithm that can be used to verify the integrity of the JSON payload.

In general, whenever we generated token with JWT, token generated in the format of `<header>.<payload>.<signature>` in side JWT.

Example:

```
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaWxpcEBnbWFpbC5jb20iLCJleHAiOjE2ODk
1MjI5OTcsImlhdCI6MTY4OTUyMjY5N30.bjFnipeNqiZ5dyrXZHk0qTPciChw0Z0eNoX
5fu5uAmj6SE9mLIGD4Ll_3QeGfXjZqvv8KlJe2pmTseT4g8ZSIA
```

Following image showing details of Encoded Token.


Encoded PASTE A TOKEN HERE
Decoded EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE
PAYLOAD: DATA
VERIFY SIGNATURE

```
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaWxpcE
BnbWFpbC5jb20iLCJleHAiOjE2ODk1MjI5OTcsI
mlhdCI6MTY4OTUyMjY5N30.bjFnipeNqiZ5dyrX
ZHk0qTPciChw0Z0eNoX5fu5uAmj6SE9mLIGD4Ll
_3QeGfXjZqvv8KlJe2pmTseT4g8ZSIA
```

```
{
  "alg": "HS512"
}
```

```
{
  "sub": "dilip@gmail.com",
  "exp": 1689522997,
  "iat": 1689522697
}
```

```
HMACSHA512(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

JWT Token Creation and Validation:

We are using Java JWT API for creation and validation of Tokens.

- Create A Maven Project
- Add Below both dependencies, required for java JWT API.

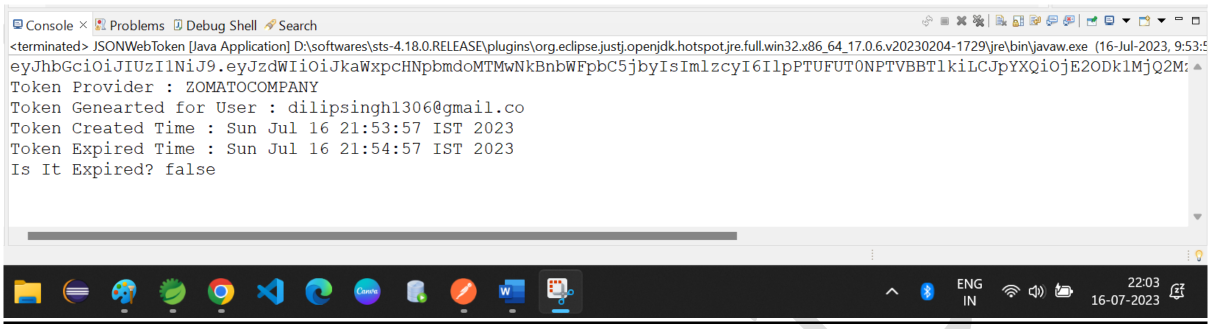
```
<dependencies>
          <dependency>
               <groupId>io.jsonwebtoken</groupId>
               <artifactId>jjwt</artifactId>
               <version>0.9.1</version>
          </dependency>
          <dependency>
               <groupId>javax.xml.bind</groupId>
               <artifactId>jaxb-api</artifactId>
               <version>2.3.0</version>
          </dependency>
</dependencies>
```

- **Now Write a Program for creating, claiming and validating JWT tokens :**
  JSONWebToken.java

```java
import java.util.Date;
import java.util.concurrent.TimeUnit;
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

//JWT Token Generation
public class JSONWebToken {

	static String key = "ZOMATO";

	public static void main(String ar[]) {

		// Creating/Producing Tokens
		String token = Jwts.builder()
						.setSubject("dilipsingh1306@gmail.co") // User ID
						.setIssuer("ZOMATOCOMPANY")
						.setIssuedAt(new Date(System.currentTimeMillis()))
	.setExpiration(new Date(System.currentTimeMillis() + TimeUnit.MINUTES.toMillis(1)))
						.signWith(SignatureAlgorithm.HS256, key.getBytes())
						.compact();

		System.out.println(token);

		// Reading/Parsing Token Details
		claimToken(token);

		//Checking Expired or not.
		boolean isExpired = isTokenExpired(token);
		System.out.println("Is It Expired? " + isExpired);

	}

	public static void claimToken(String token) {

		// Claims : Reading details from generated token by passing secret
		Claims claim = (Claims)
		Jwts.parser().setSigningKey(key.getBytes()).parse(token).getBody();

		Date createdDateTime = claim.getIssuedAt();
		Date expDateTime = claim.getExpiration();
		String issuer = claim.getIssuer();
		String subject = claim.getSubject();

		System.out.println("Token Provider : " + issuer);
		System.out.println("Token Genearted for User : " + subject);
		System.out.println("Token Created Time : " + createdDateTime);
		System.out.println("Token Expired Time : " + expDateTime);
	}

	public static boolean isTokenExpired(String token) {
```

```
                    Claims claim = (Claims)
Jwts.parser().setSigningKey(key.getBytes()).parse(token).getBody();
                    Date expDateTime = claim.getExpiration();
                    return expDateTime.before(new Date(System.currentTimeMillis()));
          }

}
```

<u>Output:</u>



```
<terminated> JSONWebToken [Java Application] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.exe  (16-Jul-2023, 9:53:5
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJkaWxpcHNpbmdoMTMwNkBnbWFpbC5jbyIsImlzcyI6IlpPTUFUT0NPTVBBTlkiLCJpYXQiOjE2ODk1MjQ2M:
Token Provider : ZOMATOCOMPANY
Token Genearted for User : dilipsingh1306@gmail.co
Token Created Time : Sun Jul 16 21:53:57 IST 2023
Token Expired Time : Sun Jul 16 21:54:57 IST 2023
Is It Expired? false
```

The above program written for understanding of how tokens are generated and how we are parsing/claiming details from JSON token.

Now we will re-use above logic as part of SpringBoot Security Implementation. Let's start SpringBoot Security with JWT.

GitHub Repository Link : https://github.com/dilipsingh1306/javaJWTToken

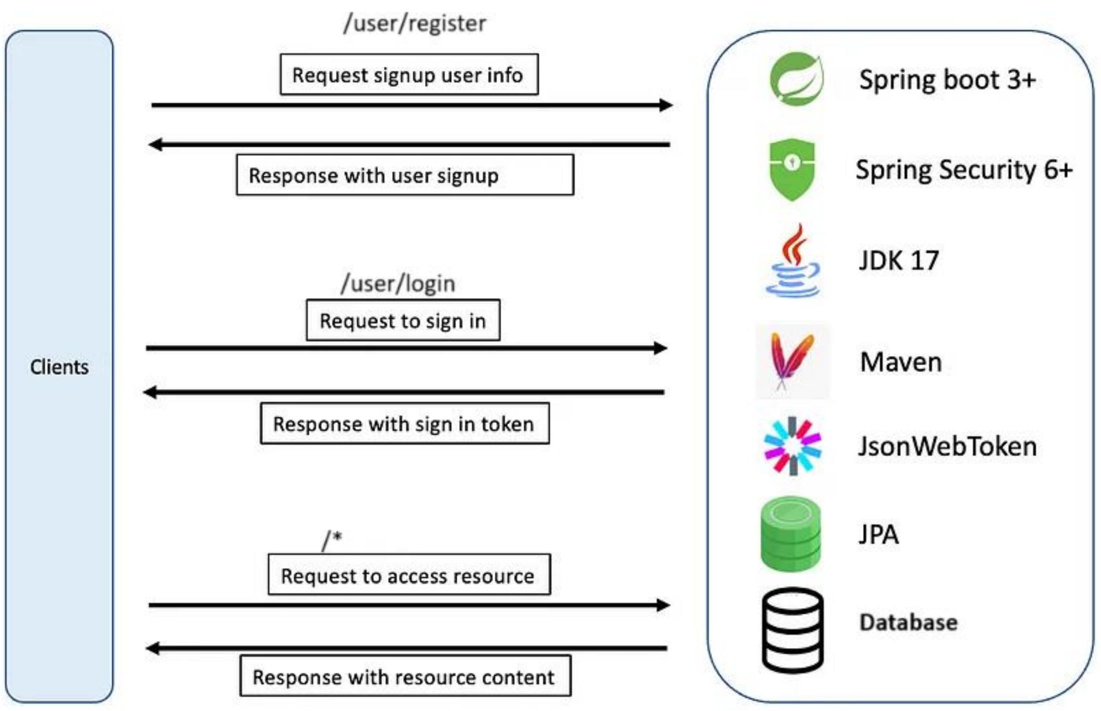
SpringBoot Security + (JSON WEB TOKEN):

The application we are going to develop will handle user authentication and authorization with JWT's for securing an exposed REST API Services.

NOTE: We are using SpringBoot Version 3.1.1 in our training. SpringBoot 3.X Internally uses or implemented with Spring Framework Version 6. Spring 6 Security API Updated with new Classes and Methods comparing with Spring 5.

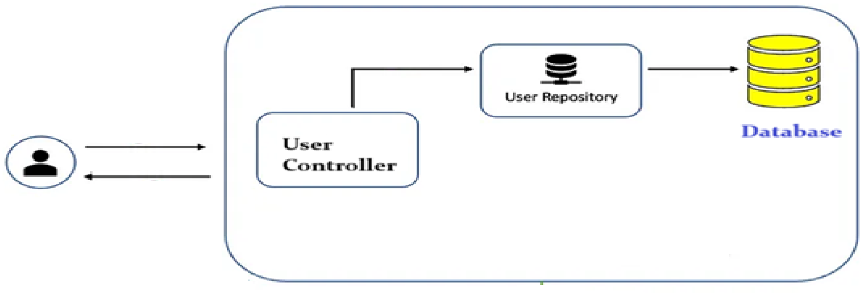
Application Architecture: Scenarios:

- User makes a request to the service, for create an account.
- User submits login request to the service to authenticate their account.
- An authenticated user sends a request to access resources/services.



Sign Up Process:

Step 1: Implement Logic for User Sign Up Process. The Sign-up process is very simple. Please understand following Signup Flow Diagram.



Signup Flow

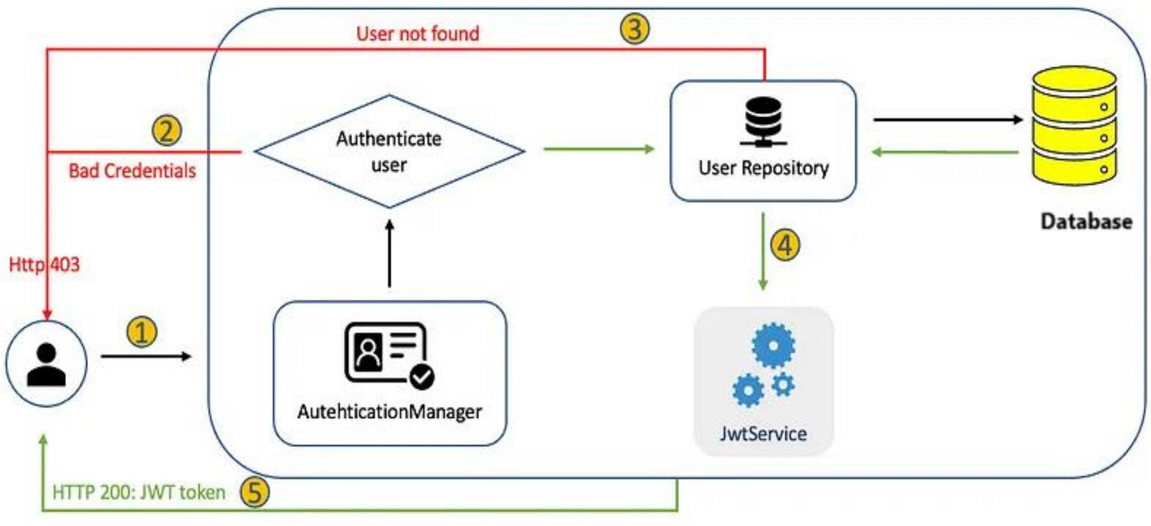
- The process starts when a user submits a request to our service. A user object is then generated from the request data, and we should encode password before storing inside Database. The password being encoded by using Spring provided Password Encoders.

It is important that we must inform Spring about the specific password encoder utilized in the application, In this case, we are using **BCryptPasswordEncoder**. This information is necessary for Spring to properly authenticate users by decoding their passwords. We will have more information about Password Encoder further.

In our application requirement is, For User Sign-up provide details of email ID, Password, Name and Mobile Number. Email ID and Password are inputs for Sign-In operation.

<u>Sign-In Activity:</u>

Internal Process and Logic Implementation:



Sign-In Flow

1. The process begins when a user sends a sign-in request to the Service**.** An Authentication object called **UsernamePasswordAuthenticationToken** is then generated, using the provided username and password.

2. The **AuthenticationManager** is responsible for authenticating the Authentication object, handling all necessary tasks. **If the username or password is incorrect**, an exception is thrown as Bad Credentials, and a response with HTTP Status 403 is returned to the user.
3. **After successful authentication**, Once we have the user information, we call the JwtService to generate the JWT for that User Id.
4. The JWT is then encapsulated in a JSON response and returned to the user.

Two new concepts are introduced here, and I'll provide a brief explanation for each.

UsernamePasswordAuthenticationToken: A type of Authentication object which can be created from a username and password that are submitted.

AuthenticationManager: Processes authentication object and will do all authentication jobs for us.

Resource/Services Accessibility:

When User tries to access any other resources/REST services of application, then we will apply security rules and after success authentication and authorization of user, we will allow to access/execute services. If Authentication failed, then we will send Specific Error Response codes usually 403 Forbidden.

Internally how we are going to enabling Security with JSON web token:

This process is secured by Spring Security, Let's defne its flow as follows.

1. When the Client sends a request to the Service, The request is first intercepted by **JWTTokenFilter**, which is a custom filter integrated into the **SecurityFilterChain.**

2. As the API is secured, if the JWT is missing as part of Request Body header, a response with HTTP Status 403 is sent to the client.

3. When an existing JWT is received**, JWTTokenFilter** is called to extract the user ID from the JWT. If the user ID cannot be extracted, a response with HTTP Status 403 is sent to the user.

4. If the user ID can be extracted, it will be used to query the user's authentication and authorization information via **UserDetailsService** of Spring Security.

5. If the user's authentication and authorization information does not exist in the database, a response with HTTP Status 403 is sent to the user.

6. If the JWT is expired, a response with HTTP Status 403 is sent to the user.

7. After successful authentication, the user’s details are encapsulated in a **UsernamePasswordAuthenticationToken** object and stored in the **SecurityContextHolder**.

8. The Spring Security Authorization process is automatically invoked.

9. The request is dispatched to the controller, and a successful JSON response is returned to the user.

This process is a little bit tricky because involving some new concepts. Let’s have some information about all new items.

SecurityFilterChain: In Spring Security, the **SecurityFilterChain** is responsible for managing a chain of security filters that process and enforce security rules for incoming requests in order to decide whether rules applies to that request or not. It plays a crucial role in handling authentication, authorization, and other security-related tasks within a Spring Security-enabled application. The **SecurityFilterChain** interface represents a single filter chain configuration. If we want, we can define multiple **SecurityFilterChain** instances to handle different sets of URLs or request patterns, allowing over security rules based on specific requirements.

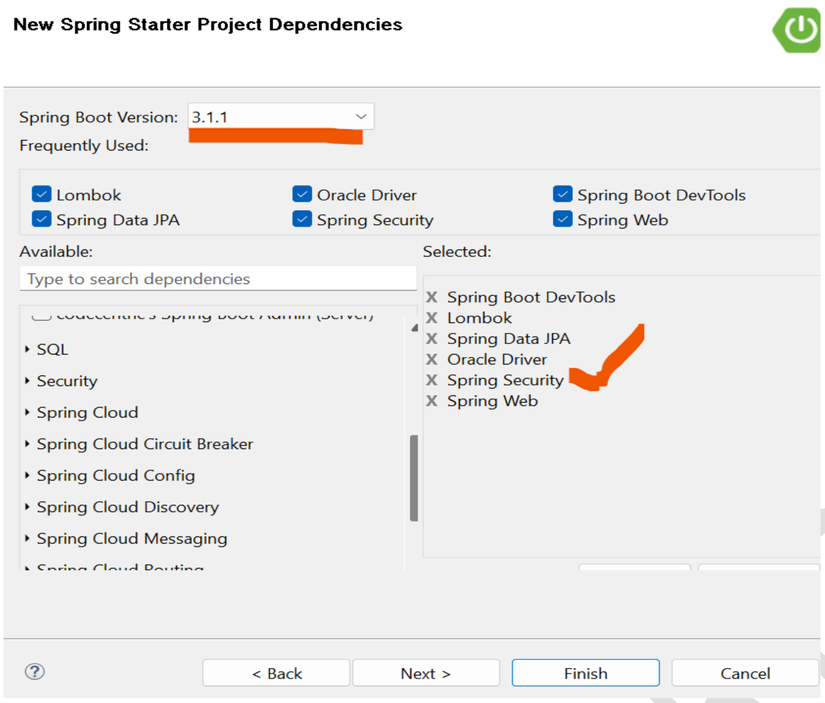
SecurityContextHolder: The **SecurityContextHolder** class is responsible for managing the **SecurityContext** object, which holds the security-related information. The **SecurityContext** contains the Authentication object representing the current user's authentication details, such as their principal (typically a user object) and their granted authorities. You can access the **SecurityContext** using the static methods of **SecurityContextHolder**.

UserDetailsService: In Spring Security, the **UserDetailsService** interface is used to retrieve user-related data during the authentication process. It provides a mechanism for Spring Security to load user details (such as username, password, and authorities) from database or any other data source. The **UserDetailsService** interface defines a single method:

UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

The **loadUserByUsername()** method is responsible for retrieving the user details for a given username. It returns an implementation of the **UserDetails** interface, which represents the user's security-related data.

Security Logic Implementation: Now Create Spring Boot Application with Security API:


New Spring Starter Project Dependencies

Spring Boot Version: 3.1.1

Frequently Used:

- [✓] Lombok
- [✓] Oracle Driver
- [✓] Spring Boot DevTools
- [✓] Spring Data JPA
- [✓] Spring Security
- [✓] Spring Web

Selected:

- Spring Boot DevTools
- Lombok
- Spring Data JPA
- Oracle Driver
- Spring Security
- Spring Web

- After Successsful Project creation, we should add JWT librarys dependecis inside Maven pom.xml file because by default SpringBoot not provding support of JWT.

```
<dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt</artifactId>
        <version>0.9.1</version>
</dependency>
<dependency>
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
        <version>2.3.0</version>
</dependency>
```

Now Configure Application Port Number, Context-Path along with Database Details inside **application.properties** file.

```
#App Details
server.port=8877
server.servlet.context-path=/zomato

#DB Details
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

Now create Controller, Service and Repository Layers.

Note: In this application, we are using Lombok library so we are using Lombok annotations instead of writing setters, getters and constructors. Please add import statements for used annoatations.

➢ Defining Request and Response Classes for Signup nad SingIn user services.

UserRegisterRequest.java

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@ToString
public class UserRegisterRequest {
        private String emailId;
        private String password;
        private String name;
        private long mobile;
}
```

UserRegisterResponse.java

```
@Data
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class UserRegisterResponse {
        private String emailId;
        private String message;
}
```

UserLoginRequest.java

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@ToString
public class UserLoginRequest {
        private String emailId;
        private String password;
}
```

UserLoginResponse.java

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
```

```
@Data
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class UserLoginResponse {
        private String token;
        private String emailId;
}
```

➢ **Now Add Signup and Sign-in Services in Controller class with Authentication Layer.**

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.dilip.jwt.token.JWTTokenHelper;
import com.dilip.user.request.UserLoginRequest;
import com.dilip.user.request.UserRegisterRequest;
import com.dilip.user.response.UserLoginResponse;
import com.dilip.user.response.UserRegisterResponse;
import com.dilip.user.service.UsersRegisterService;

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

        Logger logger = LoggerFactory.getLogger(UserController.class);

        @Autowired
        UsersRegisterService usersRegisterService;

        @Autowired
        JWTTokenHelper jwtTokenHelper;

        @Autowired
        AuthenticationManager authenticationManager;
```

```
	@Autowired
	BCryptPasswordEncoder passwordEncoder;

	@GetMapping("/hello")
	public String syaHello() {
		return "Welcome to Security";
	}

	// User Sing-Up Operation
	@PostMapping("/register")
	public ResponseEntity<UserRegisterResponse> createUserAccount(@RequestBody
	UserRegisterRequest request) {

	request.setPassword(passwordEncoder.encode(request.getPassword()));

	String result = usersRegisterService.createUserAccount(request);
	return ResponseEntity.ok(new UserRegisterResponse(request.getEmailId(), result));
	}

	// 2. Login User
	@PostMapping("/login")
	public ResponseEntity<UserLoginResponse> loginUser(@RequestBody
	UserLoginRequest request) {

	// logic for authentication of user login time
	this.doAuthenticate(request.getEmailId(), request.getPassword());

	String token = this.jwtTokenHelper.generateToken(request.getEmailId());
	return ResponseEntity.ok(new UserLoginResponse(token, request.getEmailId()));

	}
	private void doAuthenticate(String emailId, String password) {

	logger.info("Authentication of USer Credentils");

	UsernamePasswordAuthenticationToken authentication = new
	UsernamePasswordAuthenticationToken(emailId, password);
	try {
		authenticationManager.authenticate(authentication);
	} catch (BadCredentialsException e) {
			throw new RuntimeException("Invalid UserName and Password");
	}
	}

}
```

From the above controller class, we defined login service and as part of that we are enabling authentication with **AuthenticationManager** as part of Security Module by passing **UsernamePasswordAuthenticationToken** with requester user Id and password.

➢ Now create Service Layer for User Registration Logic**:** **UsersRegisterService.java**

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.dilip.user.entity.UserRegister;
import com.dilip.user.repository.UsersRegisterRepository;
import com.dilip.user.request.UserRegisterRequest;

@Service
public class UsersRegisterService {

	@Autowired
	UsersRegisterRepository usersRegisterRepository;

	public String createUserAccount(UserRegisterRequest request) {

		UserRegister register = UserRegister.builder()  // Initilizing based on builder
				.emailId(request.getEmailId())
				.password(request.getPassword())
				.name(request.getName())
				.mobile(request.getMobile())
				.build();  // Create instance with provided values

		usersRegisterRepository.save(register);
		return "Registerded Successfully";
	}
}
```

Logic Implementation:

- Create Custom Entity Class by implanting **UserDetails** Interface of Spring Security API. So that we can directly Store Repository Data of User Credentials and roles in side UserDetails. Now same will be utilized by Spring Authentication and Authorization Modules internally.

```
import java.util.Collection;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
```

```java
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Table(name = "user_register")
@Builder
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserRegister implements UserDetails {

	@Id
	private String emailId;
	private String password;
	private String name;
	private long mobile;

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


	@Override
	public String getUsername() {
		return emailId;
	}
	@Override
	public boolean isAccountNonExpired() {
		return true;
	}
	@Override
	public boolean isAccountNonLocked() {
		return true;
	}
	@Override
	public boolean isCredentialsNonExpired() {
		return true;
	}
	@Override
	public boolean isEnabled() {
		return true;
	}

}
```

- **Add a method in Repository, to retrieve User details based on user ID i.e. email ID in our case. This method will be used as part of Authentication Service Implementation in following steps.**

```
import java.util.Optional;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.dilip.user.entity.UserRegister;

@Repository
public interface UsersRegisterRepository extends JpaRepository<UserRegister, String>{

        Optional<UserRegister> findByEmailId(String emailId);
}
```

- Now Define Authentication UserService i.e. Implantation **UserDetailsService** interface from Spring Security to retrieve User Details from database for internal use by Authentication Security Filters.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
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.stereotype.Service;

import com.dilip.user.entity.UserRegister;
import com.dilip.user.repository.UsersRegisterRepository;

@Service
public class UserAuthenticationServiceImpl implements UserDetailsService{

Logger logger = LoggerFactory.getLogger(UserAuthenticationServiceImpl.class);

@Autowired
UsersRegisterRepository repository;

@Override
public UserDetails loadUserByUsername(String emailId) throws UsernameNotFoundException {

logger.info("Fetching UserDetails");

UserRegister user =  repository.findByEmailId(emailId).orElseThrow(() -> new
UsernameNotFoundException("Invalid User Name"));

return user;
}
}
```

- Create **JWTTokenHelper.java** as Component, So SpringBoot will create bean Object. This is responsible for all JWT operations i.e. creation and validation of tokens.

```
import java.util.Date;
import org.springframework.stereotype.Component;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

@Component
public class JWTTokenHelper {

  long JWT_TOKEN_VALIDITY_MILLIS =  5 * 60000; // 5 mins
  String secret = fasfafacasdasfasxASFACASDFACASDFASFASFDAFASFASDAADSCSDFADCVSGCFVADX";

  //retrieve username from jwt token
  public String getUsernameFromToken(String token) {
    return Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBody().getSubject();
  }

  //check if the token has expired
  private Boolean isTokenExpired(String token) {
        final Date expiration = Jwts.parser()
                                    .setSigningKey(secret)
                                    .parseClaimsJws(token)
                                    .getBody()
                                    .getExpiration();
    return expiration.before(new Date());
  }

  //generate JWT using the HS512 algorithm and secret key.
  public String generateToken(String userName) {
        return Jwts
        .builder()
        .setSubject(userName)
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + JWT_TOKEN_VALIDITY_MILLIS))
        .signWith(SignatureAlgorithm.HS512, secret)
         .compact();
  }

  //validate token i.e. user name and time interval validation.
  public Boolean validateToken(String token, String userName) {
    final String username = getUsernameFromToken(token);
    return (username.equals(userName) && !isTokenExpired(token));
  }
}
```

- Now Define a custom filter by extending OncePerRequestFilter to handle JWT token for every incoming new request.
- This New Filter is responsible for checking like token available or not as part of Request
- If token available, This Filter validates token w.r.to user Id and Expiration time interval.
- This Filter is responsible for cross checking User Id of Token with Database user details.

```
import java.io.IOException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
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.dilip.jwt.token.JWTTokenHelper;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@Component
public class JWTTokenFilter extends OncePerRequestFilter {

	Logger logger = LoggerFactory.getLogger(JWTTokenFilter.class);

	@Autowired
	JWTTokenHelper jwtTokenHelper;

	@Autowired
	UserAuthenticationServiceImpl authenticationService;

	@Override
	protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
FilterChain filterChain) throws ServletException, IOException {

		logger.info("validation of JWT token by OncePerRequestFilter");

		String token = request.getHeader("Authorization");

		logger.info("JWT token : " + token);
		String userName = null;

		if (token != null) {

			userName = this.jwtTokenHelper.getUsernameFromToken(token);
			logger.info("JWT token USer NAme : " + userName);
		} else {
			logger.info("ToKen is Misisng. Please Come with Token");
```

```
                    }

if (userName != null && SecurityContextHolder.getContext().getAuthentication() == null) {

// fetch user detail from username
UserDetails userDetails = this.authenticationService.loadUserByUsername(userName);
Boolean isValidToken = this.jwtTokenHelper.validateToken(token, userDetails.getUsername());

if (isValidToken) {

UsernamePasswordAuthenticationToken authenticationToken = new
UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());

authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
SecurityContextHolder.getContext().setAuthentication(authenticationToken);
                              }
                    }
                    filterChain.doFilter(request, response);
          }
}
```

- Now create a Authentication Configuration class responsible for Creating **AuthenticationManager**, **PasswordEncryptor** and **SecurityFilterChain** to define strategies of incoming requests like which should be authenticated with JW and which should be ignored by Security layer.

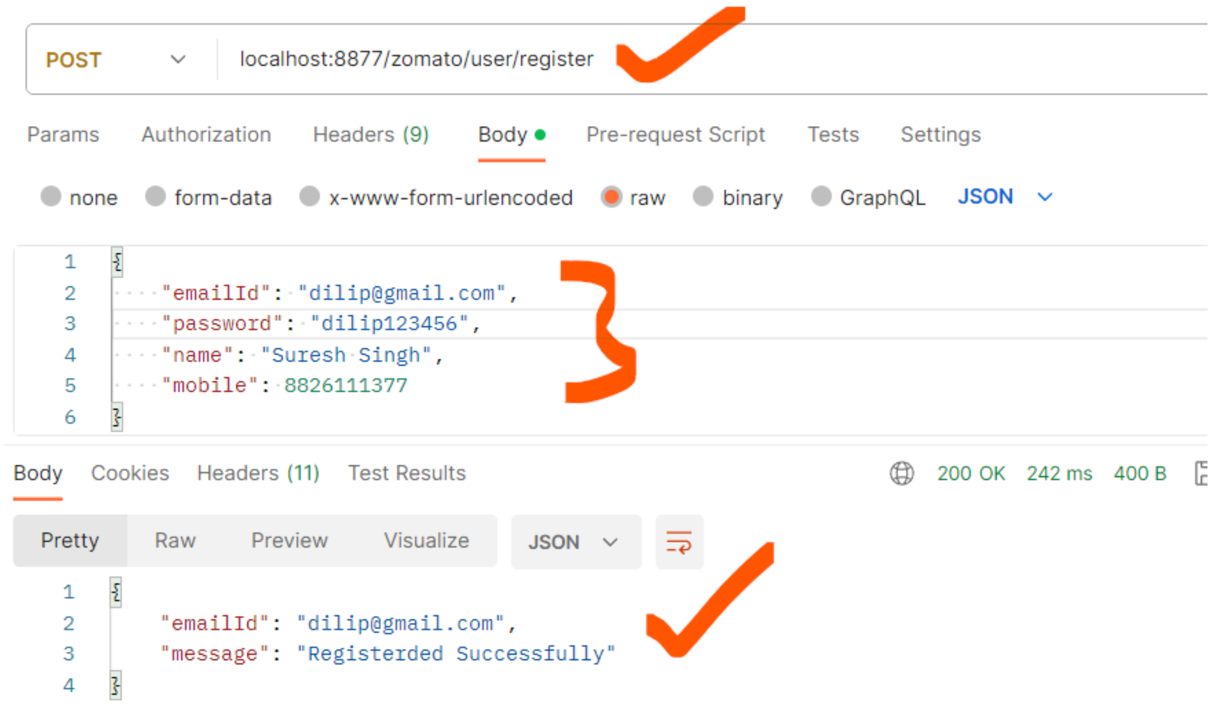
```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
public class AppSecurityConfig {

          Logger logger = LoggerFactory.getLogger(AppSecurityConfig.class);

          @Autowired
          JWTTokenFilter jwtTokenFilter;

          @Bean
          AuthenticationManager      getAuthenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
                logger.info("Initilizing Bean AuthenticationManager");
                return authenticationConfiguration.getAuthenticationManager();
          }

          @Bean
          BCryptPasswordEncoder getBCryptPasswordEncoder() {
                logger.info("Initilizing Bean BCryptPasswordEncoder");
                return new BCryptPasswordEncoder();
          }

          @Bean
          public SecurityFilterChain getSecurityFilterChain(HttpSecurity security) throws Exception {

                logger.info("Configuring SecurityFilterChain Layer  of URl patterns");

                security.csrf(csrf -> csrf.disable())
                        .cors(cors -> cors.disable())
                        .authorizeHttpRequests(
                            auth ->  auth.requestMatchers("/user/login","/user/register")
                                                    .permitAll()
                                                    .anyRequest()
                                                    .authenticated()
                        )
          .addFilterBefore(this.jwtTokenFilter,UsernamePasswordAuthenticationFilter.class);

                return security.build();
          }

}
```

In Above Configuration Class, We created SecurityFilterChain Bean with Security rules defined for every incoming request. We are defined Security Configuration as, permitting all incoming requests without JWT token validation for both URI mappings of **"/user/login","/user/register".** Apart from these 2 mappings , any other request should be authenticated with JWT Token i.e. every request should come with valid token always then only we are allowing to access actual Resources or Services.

Testing : Let's Test our application as per our requirement and security configuration.

Case 1: Now Create User Accounts with **localhost:8877/zomato/user/register**

This is Open API Service means security not applicable for this. i.e. to execute no need to provide JWT.

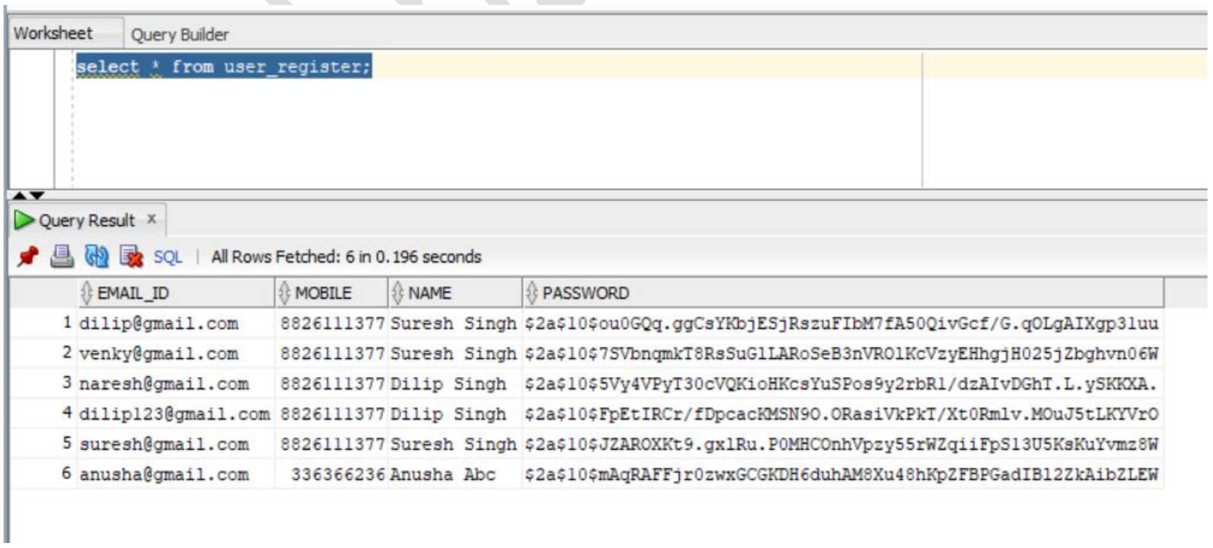

POST localhost:8877/zomato/user/register

```
{
    "emailId": "dilip@gmail.com",
    "password": "dilip123456",
    "name": "Suresh Singh",
    "mobile": 8826111377
}
```

200 OK 242 ms 400 B

```
{
    "emailId": "dilip@gmail.com",
    "message": "Registerded Successfully"
}
```

Now verify password value in database how it is stored because we encoded it.



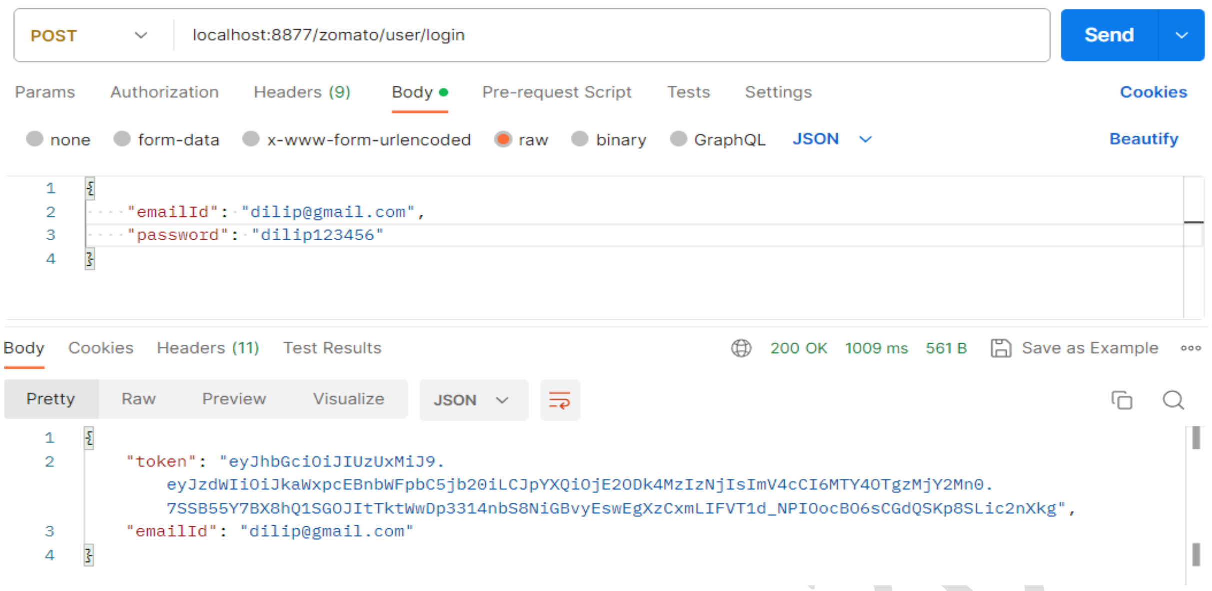
```
select * from user_register;
```

All Rows Fetched: 6 in 0.196 seconds

	EMAIL_ID	MOBILE	NAME	PASSWORD
1	dilip@gmail.com	8826111377	Suresh Singh	$2a$10$ou0GQq.ggCsYKbjESjRszuFIbM7fA50QivGcf/G.qOLgAIXgp3luu
2	venky@gmail.com	8826111377	Suresh Singh	$2a$10$7SVbnqmkT8RsSuGlLARoSeB3nVROlKcVzyEHhgjH025jZbghvn06W
3	naresh@gmail.com	8826111377	Dilip Singh	$2a$10$5Vy4VPyT30cVQKioHKcsYuSPos9y2rbRl/dzAIvDGhT.L.ySKKXA.
4	dilip123@gmail.com	8826111377	Dilip Singh	$2a$10$FpEtIRCr/fDpcacKMSN9O.ORasiVkPkT/Xt0Rmlv.MOuJ5tLKYVrO
5	suresh@gmail.com	8826111377	Suresh Singh	$2a$10$JZAROXKt9.gxlRu.POMHCOnhVpzy55rWZqiiFpS13U5KsKuYvmz8W
6	anusha@gmail.com	336366236	Anusha Abc	$2a$10$mAqRAFFjr0zwxGCGKDH6duhAM8Xu48hKpZFBPGadIBl2ZkAibZLEW

Passwords are stored as encoded format. Now Spring also takes care of decoded while authentication because we are created Bean of Password Encryptor.

Case 2: Now try to login with User Details. localhost:8877/zomato/user/login


POST localhost:8877/zomato/user/login

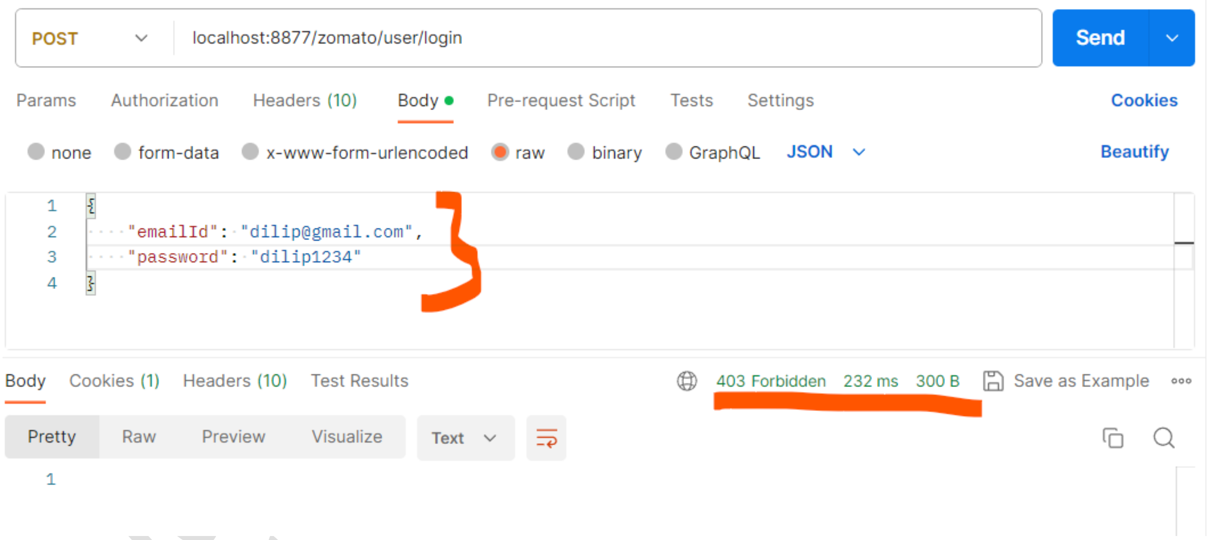
```
{
    "emailId": "dilip@gmail.com",
    "password": "dilip123456"
}
```

200 OK 1009 ms 561 B

```
{
    "token": "eyJhbGciOiJIUzUxMiJ9.
        eyJzdWIiOiJkaWxpcEBnbWFpbC5jb20iLCJpYXQiOjE2ODk4MzIzNjIsImV4cCI6MTY4OTgzMjY2Mn0.
        7SSB55Y7BX8hQ1SGOJItTktWwDp3314nbS8NiGBvyEswEgXzCxmLIFVT1d_NPIOocBO6sCGdQSKp8SLic2nXkg",
    "emailId": "dilip@gmail.com"
}
```

On Successful validation, We received JSON Web token.

If we provide Wrong Credentials: Entered Wrong Password.


POST localhost:8877/zomato/user/login

```
{
    "emailId": "dilip@gmail.com",
    "password": "dilip1234"
}
```

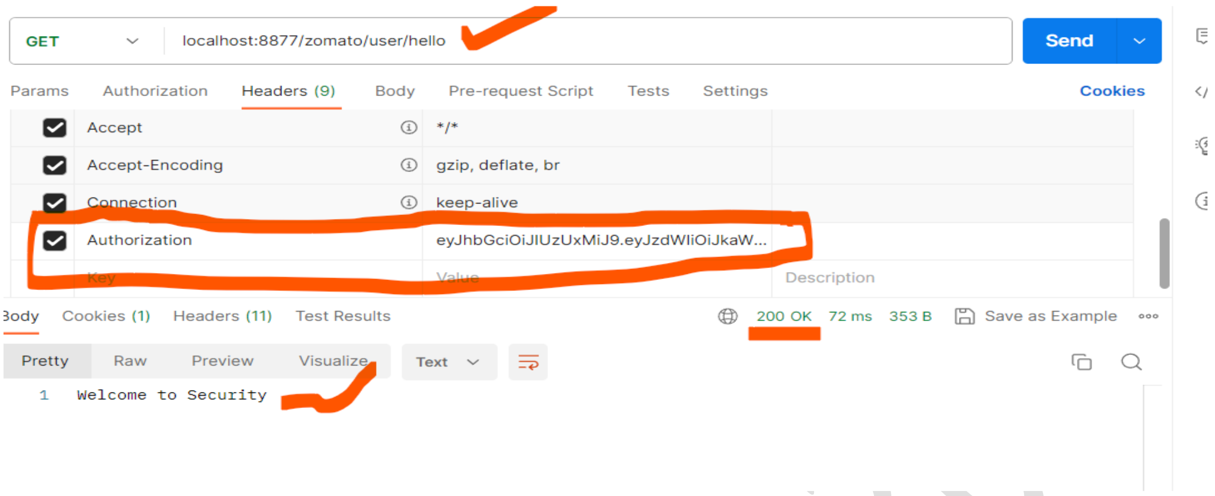
403 Forbidden 232 ms 300 B

Now we got expected and default Response as Forbidden with status code 403:

Forbidden meaning is : **not allowed; banned.**

Case 3: Access Other URI or Services with JWT: **localhost:8877/zomato/user/hello**

Here we will pass Token as **Authorization** Header as part of Request. Copy Token from login response from previous call and pass an value to **Authorization** Header as following.



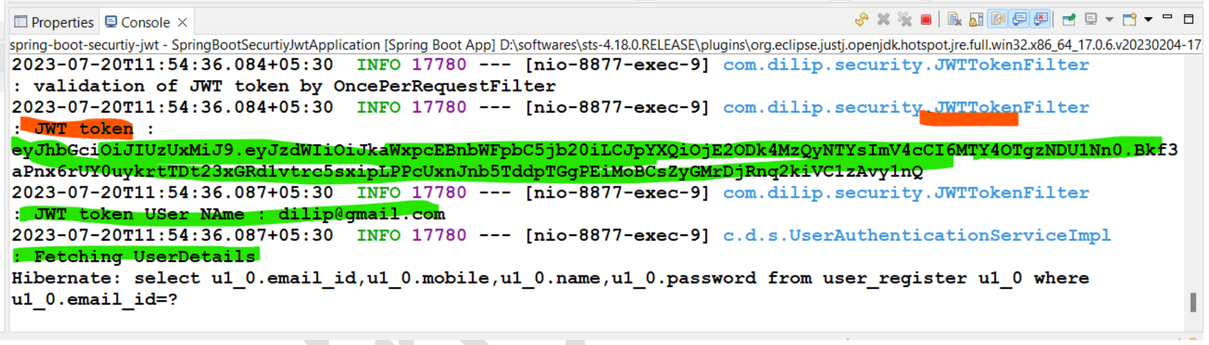
GET localhost:8877/zomato/user/hello

Key	Value	Description
✓ Accept	*/*	
✓ Accept-Encoding	gzip, deflate, br	
✓ Connection	keep-alive	
✓ Authorization	eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaW...	

200 OK 72 ms 353 B

```
1   Welcome to Security
```

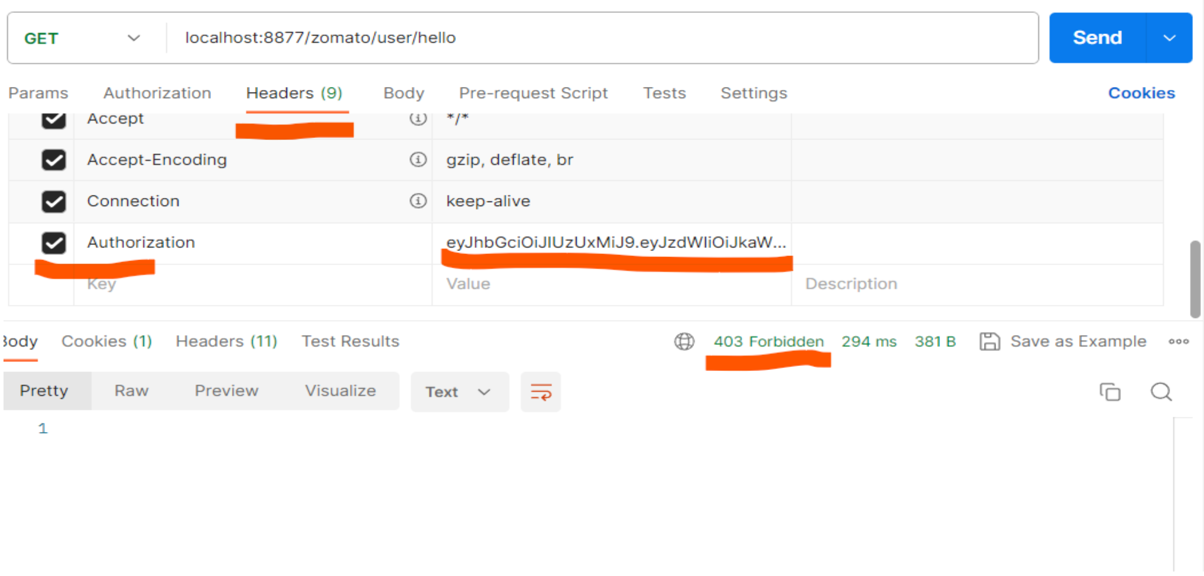
Now received respected Response value and Status code as 200 OK. i.e. Internally JWT is validated with user and database as per our logic implemented. We can see in Logs of application.



```
spring-boot-securtiy-jwt - SpringBootSecurtiyJwtApplication [Spring Boot App] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-17
2023-07-20T11:54:36.084+05:30  INFO 17780 --- [nio-8877-exec-9] com.dilip.security.JWTTokenFilter
: validation of JWT token by OncePerRequestFilter
2023-07-20T11:54:36.084+05:30  INFO 17780 --- [nio-8877-exec-9] com.dilip.security.JWTTokenFilter
: JWT token :
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaWxpcEBnbWFpbC5jb20iLCJpYXQiOjE2ODk4MzQyNTYsImV4cCI6MTY4OTgzNDU1Nn0.Bkf3
aPnx6rUY0uykrtTDt23xGRd1vtrc5sxipLPPcUxnJnb5TddpTGgPEiMoBCsZyGMrDjRnq2kiVC1zAvy1nQ
2023-07-20T11:54:36.087+05:30  INFO 17780 --- [nio-8877-exec-9] com.dilip.security.JWTTokenFilter
: JWT token USer NAme : dilip@gmail.com
2023-07-20T11:54:36.087+05:30  INFO 17780 --- [nio-8877-exec-9] c.d.s.UserAuthenticationServiceImpl
: Fetching UserDetails
Hibernate: select u1_0.email_id,u1_0.mobile,u1_0.name,u1_0.password from user_register u1_0 where
u1_0.email_id=?
```

Case 4: Access Other URI or Services with **Invalid JWT**: **localhost:8877/zomato/user/hello**

i.e. Provided Expired Token.



GET localhost:8877/zomato/user/hello

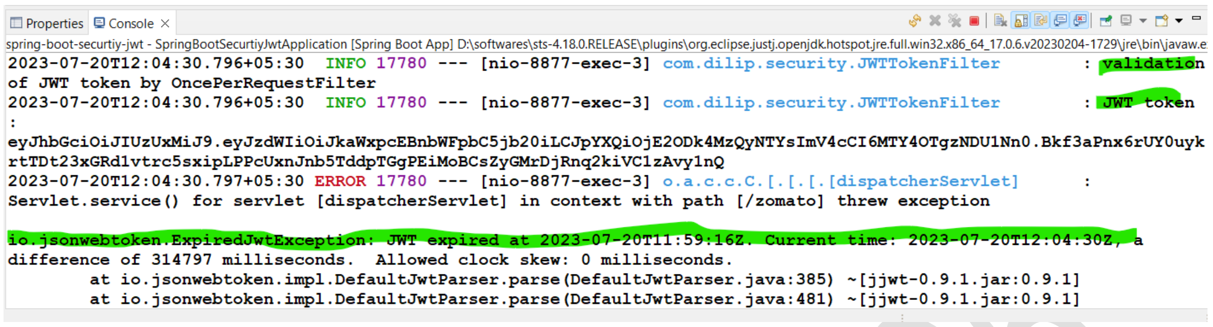
Key	Value	Description
✓ Accept	*/*	
✓ Accept-Encoding	gzip, deflate, br	
✓ Connection	keep-alive	
✓ Authorization	eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaW...	

403 Forbidden 294 ms 381 B

```
1
```

Token validated and found as Expired, so Server application returns Response as 403 Forbidden at client level.

Check Server Level logs:



```
spring-boot-securtiy-jwt - SpringBootSecurtiyJwtApplication [Spring Boot App] D:\softwares\sts-4.18.0.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.6.v20230204-1729\jre\bin\javaw.e
2023-07-20T12:04:30.796+05:30  INFO 17780 --- [nio-8877-exec-3] com.dilip.security.JWTTokenFilter        : validation
of JWT token by OncePerRequestFilter
2023-07-20T12:04:30.796+05:30  INFO 17780 --- [nio-8877-exec-3] com.dilip.security.JWTTokenFilter        : JWT token
:
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaWxpcEBnbWFpbC5jb20iLCJpYXQiOjE2ODk4MzQyNTYsImV4cCI6MTY4OTgzNDU1Nn0.Bkf3aPnx6rUY0uyk
rtTDt23xGRd1vtrc5sxipLPPcUxnJnb5TddpTGgPEiMoBCsZyGMrDjRnq2kiVC1zAvy1nQ
2023-07-20T12:04:30.797+05:30 ERROR 17780 --- [nio-8877-exec-3] o.a.c.c.C.[.[.[.[dispatcherServlet]      :
Servlet.service() for servlet [dispatcherServlet] in context with path [/zomato] threw exception

io.jsonwebtoken.ExpiredJwtException: JWT expired at 2023-07-20T11:59:16Z. Current time: 2023-07-20T12:04:30Z, a
difference of 314797 milliseconds.  Allowed clock skew: 0 milliseconds.
	at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:385) ~[jjwt-0.9.1.jar:0.9.1]
	at io.jsonwebtoken.impl.DefaultJwtParser.parse(DefaultJwtParser.java:481) ~[jjwt-0.9.1.jar:0.9.1]
```

This is how we are applying security layer to our SpringBoot Web Application with JWT exchanging.

I have uploaded this entire working copy project in GitHub.

GitHub Repository Link : https://github.com/dilipsingh1306/spring-boot-3-securtiy-jwt.git

SpringBoot Actuator:

In Spring Boot, an actuator is a set of endpoints that provides various production-ready features to help monitor and manage your application. It exposes useful endpoints that give insights into your application's health, metrics, environment, and more. Actuators are essential for monitoring and managing your Spring Boot application in production environments.

To enable the Spring Boot Actuator, you need to add the relevant dependencies to your project. In most cases, you'll want to include the `spring-boot-starter-actuator` dependency in your pom.xml (Maven) or build.gradle (Gradle) file.

For Maven:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the **health** endpoint provides basic application health information.

The built-in endpoints are auto-configured only when they are available. Most applications choose exposure over HTTP, where the ID of the endpoint and a prefix of **/actuator** is mapped to a URL. For example, by default, the health endpoint is mapped to **/actuator/health**

Some of endpoints are:

ID	Description
beans	Displays a complete list of all the Spring beans in your application.
health	Shows application health information.
info	Displays arbitrary application info.
loggers	Shows and modifies the configuration of loggers in the application.

Exposing Endpoints:

By default, only the **health** endpoint is exposed. Since Endpoints may contain sensitive information, you should carefully consider when to expose them. To change which endpoints are exposed, use the following specific **include** and **exclude** properties:

Property

```
management.endpoints.web.exposure.exclude=<endpoint>,<endpoint>
management.endpoints.web.exposure.include=<endpoint>,<endpoint>
```