## Spring

## 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.

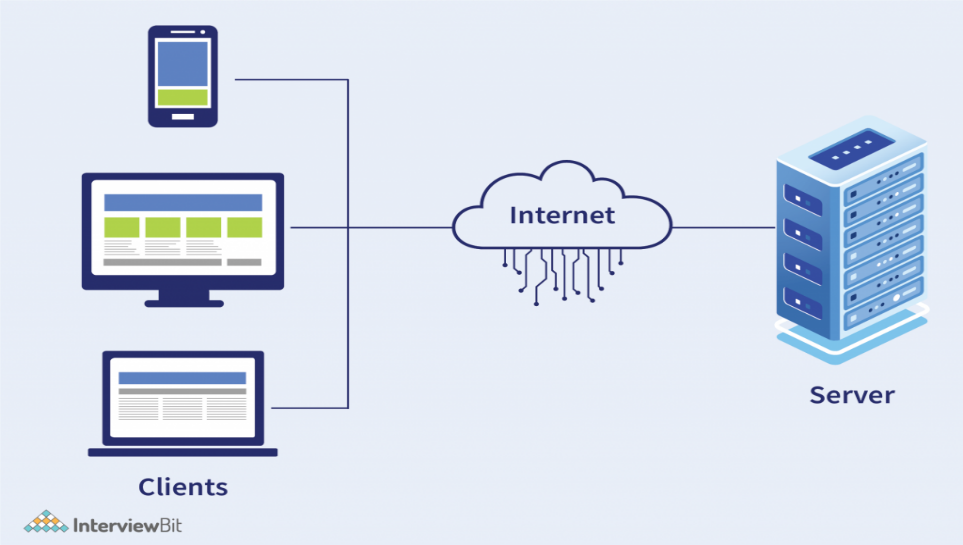
**Security Interceptors:** Spring Security uses security interceptors to enforce authorization rules. These interceptors are responsible for intercepting requests and checking whether the user has the required permissions to access the requested resource. They can be configured to protect URLs, methods,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

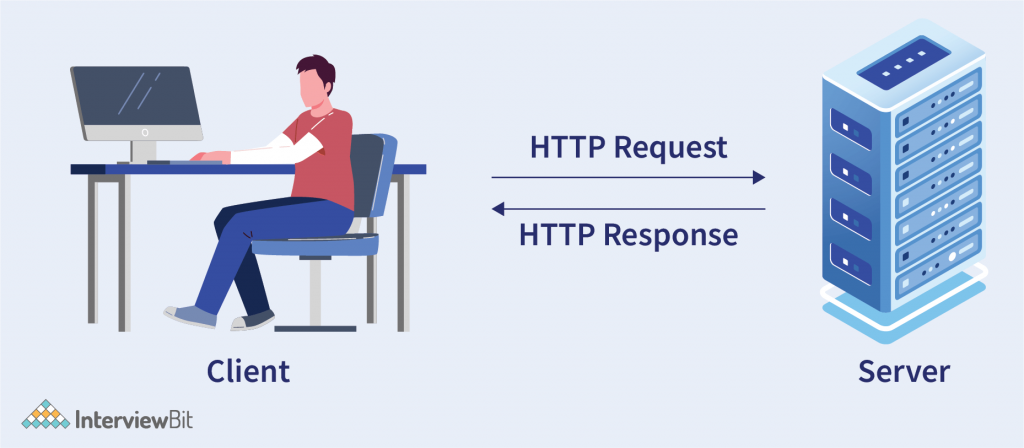
**Stateless and Stateful Protocols:**

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:**

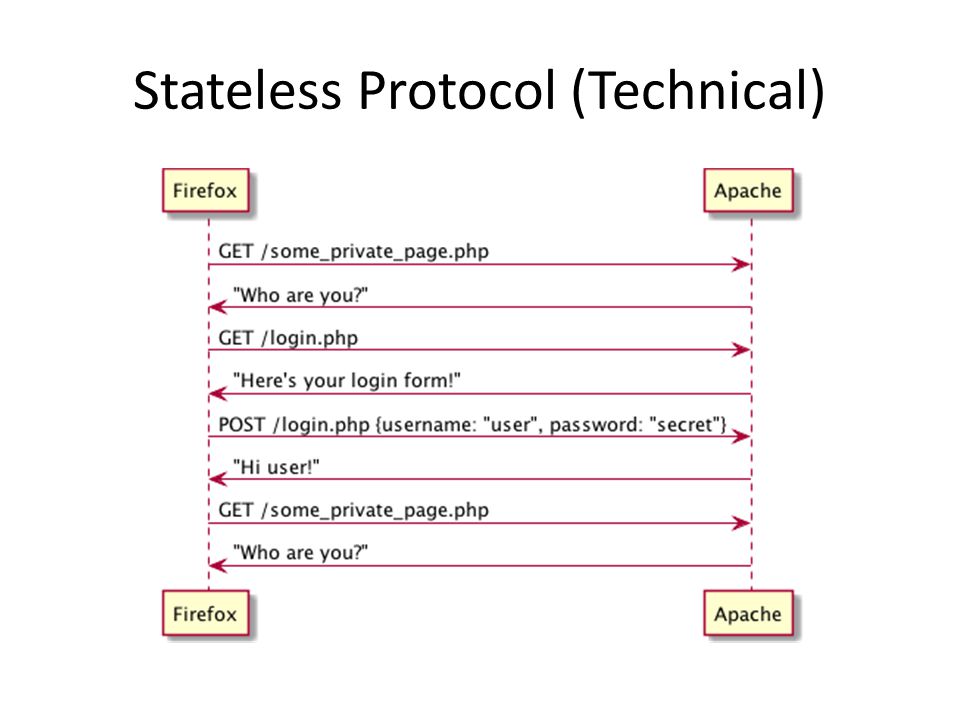
In a stateless protocol, such as HTTP, the server does not maintain any information about the client's previous interactions or session state. Each request from the client to the server is considered independent and self-contained. The server treats each request as if it is the first request from the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

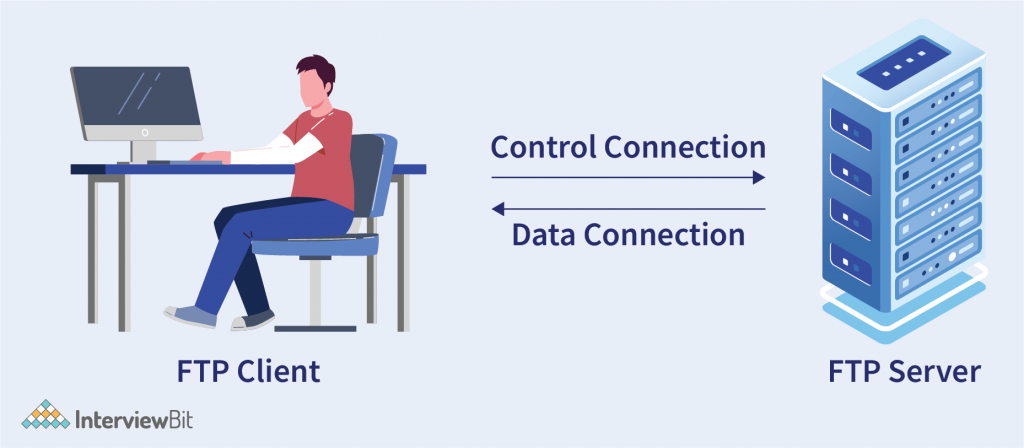
HTTP is primarily designed as a stateless protocol. When a client makes an HTTP request, the server processes the request and sends back a response. However, the server does not maintain any information about the client after the response is sent. This approach simplifies the server's implementation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



**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:

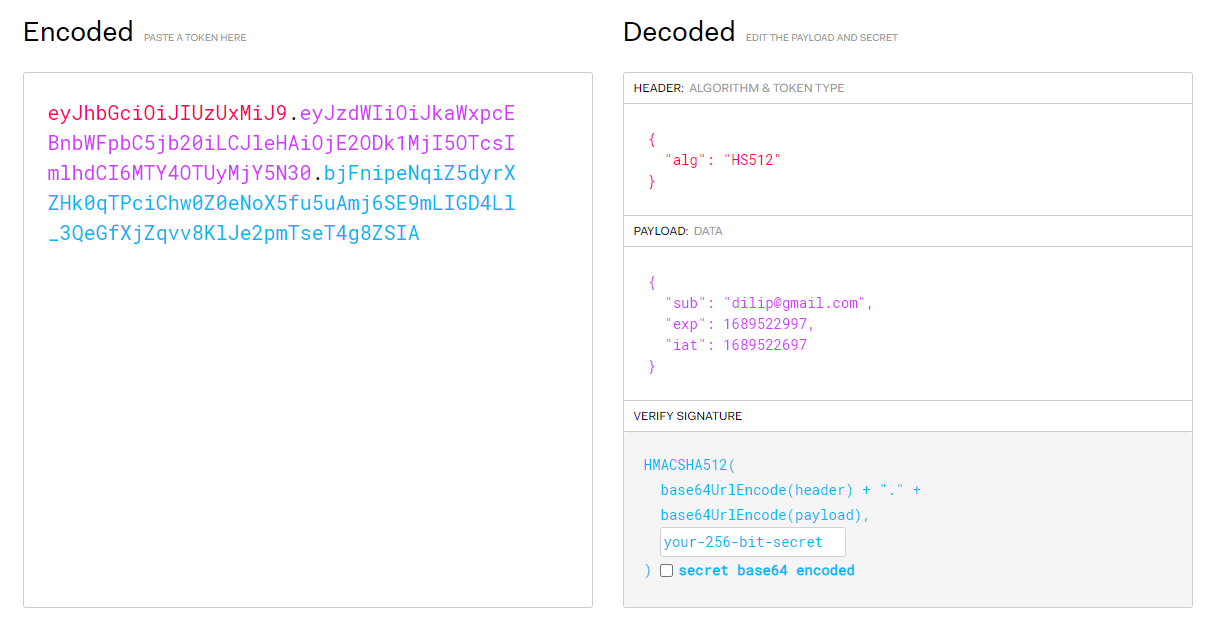
* **Header**:
  + The signing algorithm that’s being used.
* **Payload**: The payload contains the claims or the JSON object of clients.
* **Signature**: A string that is generated via a cryptographic algorithm that can be used to verify the integrity of the JSON payload.

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

**Example:**

|  |
| --- |
| eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJkaWxpcEBnbWFpbC5jb20iLCJleHAiOjE2ODk1MjI5OTcsImlhdCI6MTY4OTUyMjY5N30.bjFnipeNqiZ5dyrXZHk0qTPciChw0Z0eNoX5fu5uAmj6SE9mLIGD4Ll\_3QeGfXjZqvv8KlJe2pmTseT4g8ZSIA |

Following image showing details of Encoded Token.



**JWT Token Creation and Validation:**

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

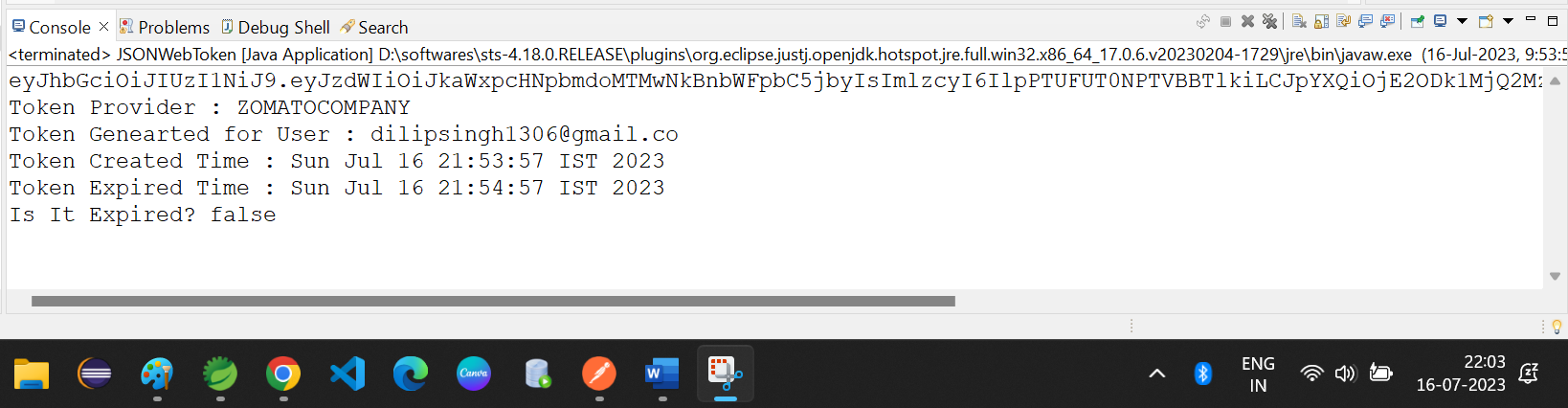
* Create A Maven Project
* Add Below both dependencies, required for java JWT API.

|  |
| --- |
| <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**

|  |
| --- |
| 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 Generated 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*()));  }  } |

**Output:**

****

**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 Spring Security Implementation. Let’s start Spring Security with JWT.**

GitHub Repository Link : <https://github.com/tek-teacher/javaJWTToken>

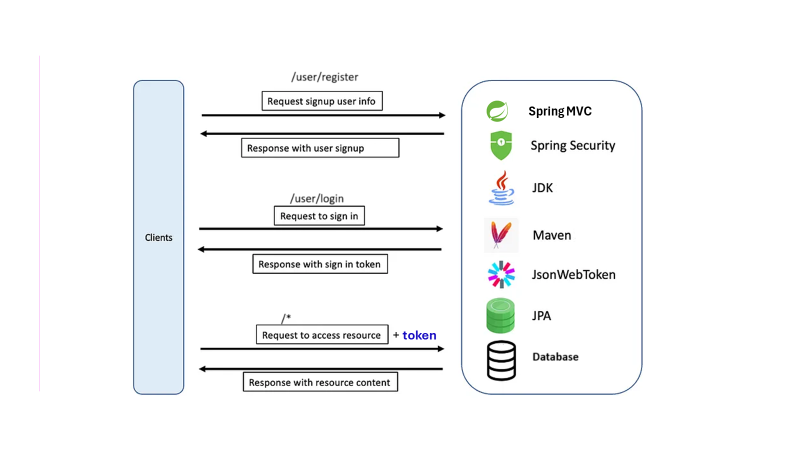
# **Spring Security + (JSON WEB TOKEN):**

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



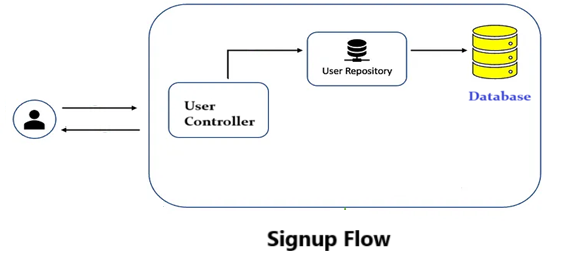
**Application Architecture: Scenarios:**

* 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.



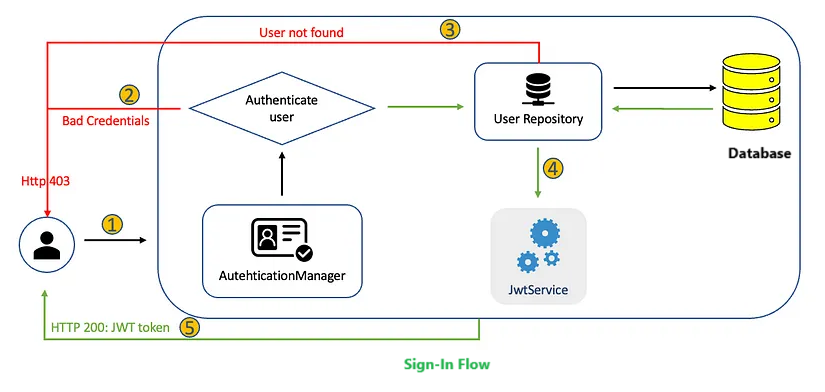
* The process starts when a user submits a request to our service. A user object is then generated from the request data, and we should encode password before storing inside Database. The password being encoded by using Spring provided Password Encoders.

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.**

**Sign-In Activity:**

**Internal Process and Logic Implementation:**

****

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 JWT Service 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**](https://docs.spring.io/spring-security/site/docs/4.0.x/apidocs/org/springframework/security/authentication/UsernamePasswordAuthenticationToken.html)**:** A type of Authentication object which can be created from a username and password that are submitted.

[**AuthenticationManager**](https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/authentication/AuthenticationManager.html)**: Processes the 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 define its flow as follows.

1. When the Client sends a request to the Service, The request is first intercepted by **JWT Token Filter**, 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**, JWT Token Filter** 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 **User Details Service** 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, user details are encapsulated in **UsernamePasswordAuthenticationToken** object and stored inside 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:

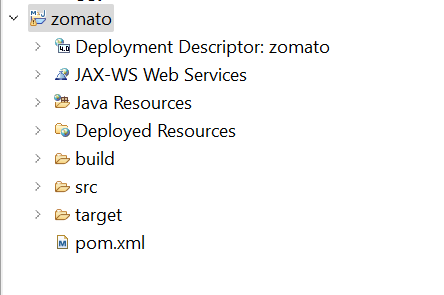
|  |
| --- |
| **UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;** |

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 Application with Security API:**

* Creating Spring MVC+JPA project.
* Create Dynamic Web Project
* Convert Project to Maven project

Right Lcick on Project -> Confgure -> Convert Maven project.



* Now Add Dependecies to **pom.xml** file to support Spring MVC + JPA

|  |
| --- |
| <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>  <groupId>zomato</groupId>  <artifactId>zomato</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>war</packaging>    <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>5.3.29</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>3.0.1</version>  <scope>provided</scope>  </dependency>  <!--JSON : Jackson API jars :-->  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-core</artifactId>  <version>2.15.2</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.15.2</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-annotations</artifactId>  <version>2.15.2</version>  </dependency>  <!--Spring JPA jar files -->  <dependency>  <groupId>com.oracle.database.jdbc</groupId>  <artifactId>ojdbc8</artifactId>  <version>21.9.0.0</version>  </dependency>  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>  <version>5.6.9.Final</version>  </dependency>  <dependency>  <groupId>org.springframework.data</groupId>  <artifactId>spring-data-jpa</artifactId>  <version>2.1.0.RELEASE</version>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <artifactId>maven-war-plugin</artifactId>  <version>3.2.3</version>  </plugin>  <plugin>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.8.1</version>  <configuration>  <release>17</release>  </configuration>  </plugin>  </plugins>  </build>  </project> |

* **Now Add MVC Module Configuration to work with Annotation Based Configuration.**

**MVCConfiguration.java**

|  |
| --- |
| package com.zomato;  import org.springframework.context.annotation.ComponentScan;  import org.springframework.context.annotation.Configuration;  import org.springframework.web.servlet.config.annotation.EnableWebMvc;  import org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;  @Configuration  @ComponentScan("com.\*")  @EnableWebMvc  public class MVCConfiguration extends WebMvcConfigurationSupport {  } |

**SpringWebInitilization.java**

|  |
| --- |
| **package** com.zomato;  **import** org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;  **public** **class** SpringWebInitilization **extends** AbstractAnnotationConfigDispatcherServletInitializer {  @Override  **protected** Class<?>[] getRootConfigClasses() {  **return** **null**;  }  @Override  **protected** Class<?>[] getServletConfigClasses() {  **return** **new** Class[] {MVCConfiguration.**class**} ;  }  @Override  **protected** String[] getServletMappings() {  String[] allowedURLMapping = {"/"};  **return** allowedURLMapping;  }  } |

* **Now Add JPA Configuration Class.**

|  |
| --- |
| package com.zomato;  import java.util.Properties;  import javax.sql.DataSource;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.data.jpa.repository.config.EnableJpaRepositories;  import org.springframework.jdbc.datasource.DriverManagerDataSource;  import org.springframework.orm.jpa.JpaTransactionManager;  import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;  import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;  import org.springframework.transaction.PlatformTransactionManager;  import org.springframework.transaction.annotation.EnableTransactionManagement;  @Configuration  @EnableTransactionManagement  @EnableJpaRepositories("com.\*")  public class SpringJpaConfiguration {  // DB Details  @Bean  public DataSource getDataSource() {  DriverManagerDataSource dataSource = new DriverManagerDataSource();  dataSource.setUrl("jdbc:oracle:thin:@localhost:1521:orcl");  dataSource.setUsername("c##dilip");  dataSource.setPassword("dilip");  dataSource.setDriverClassName("oracle.jdbc.driver.OracleDriver");  return dataSource;  }  @Bean("entityManagerFactory")  LocalContainerEntityManagerFactoryBean createEntityManagerFactory() {  LocalContainerEntityManagerFactoryBean factory =  new LocalContainerEntityManagerFactoryBean();  // 1. Setting Datasource Object // DB details  factory.setDataSource(getDataSource());  // 2. Provide package information of entity classes  factory.setPackagesToScan("com.\*");  // 3. Providing Hibernate Properties to EM  factory.setJpaProperties(hibernateProperties());  // 4. Passing Predefined Hiberante Adaptor Object EM  HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();  factory.setJpaVendorAdapter(adapter);  return factory;  }  // PSrring JPA: configuring data based on your project req.  @Bean("transactionManager")  public PlatformTransactionManager createTransactionManager() {  JpaTransactionManager transactionManager =  new JpaTransactionManager();  transactionManager.setEntityManagerFactory(createEntityManagerFactory().getObject());  return transactionManager;  }  Properties hibernateProperties() {  Properties hibernateProperties = new Properties();  hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "update");  hibernateProperties.setProperty("hibernate.dialect",  "org.hibernate.dialect.Oracle10gDialect");  hibernateProperties.setProperty("hibernate.show\_sql", "true");  return hibernateProperties;  }  } |

**With These Steps and Configuration classes, Our Project will support MVC and JPA Concpets.**

**Let’s Defeine REST Services.**

* **Add User Registaration REST Service.**

**Create Request Body Class : UserRegister.java**

|  |
| --- |
| **package** com.zomato.dto;  **public** **class** UserRegister {  **private** String emailId;  **private** String password;  **private** String fullName;  **private** String conatctNumber;  **public** String getEmailId() {  **return** emailId;  }  **public** **void** setEmailId(String emailId) {  **this**.emailId = emailId;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** String getFullName() {  **return** fullName;  }  **public** **void** setFullName(String fullName) {  **this**.fullName = fullName;  }  **public** String getConatctNumber() {  **return** conatctNumber;  }  **public** **void** setConatctNumber(String conatctNumber) {  **this**.conatctNumber = conatctNumber;  }  } |

* **Now Create REST Service for User Registration In Controller :** ZomatoController.java

|  |
| --- |
| package com.zomato.controller;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.web.bind.annotation.PostMapping;  import org.springframework.web.bind.annotation.RequestBody;  import org.springframework.web.bind.annotation.RestController;  import com.zomato.dto.UserLogin;  import com.zomato.dto.UserRegister;  import com.zomato.service.ZomatoUserService;  @RestController  public class ZomatoController {  @Autowired  ZomatoUserService service;  @PostMapping("/create/user")  public String registerUser(@RequestBody UserRegister request) {  return service.registerUser(request);  }  } |

* **Service Layer Class: ZomatoUserService.java**

|  |
| --- |
| package com.zomato.service;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import com.zomato.dto.UserLogin;  import com.zomato.dto.UserRegister;  import com.zomato.entity.UserEntity;  import com.zomato.repository.ZomatoUserRepository;  import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  @Service  public class ZomatoUserService {  @Autowired  ZomatoUserRepository repository;  @Autowired  BCryptPasswordEncoder passwordEncoder;    public String registerUser(UserRegister request) {  //Convert to Eneity Object  UserEntity entity = new UserEntity();  entity.setEmailId(request.getEmailId());  entity.setPassword(passwordEncoder.encode(request.getPassword()));  entity.setFullName(request.getFullName());  entity.setConatctNumber(request.getConatctNumber());  repository.save(entity);  return "User Created Successfully. Please Login Now.";  }  } |

* **Create Entity Class:** UserEntity.java

|  |
| --- |
| package com.zomato.entity;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.Id;  import javax.persistence.Table;  @Entity  @Table(name = "zomato\_users")  public class UserEntity {  @Id  @Column  private String emailId;  @Column  private String password;  @Column  private String fullName;  @Column  private String conatctNumber;  public String getEmailId() {  return emailId;  }  public void setEmailId(String emailId) {  this.emailId = emailId;  }  public String getPassword() {  return password;  }  public void setPassword(String password) {  this.password = password;  }  public String getFullName() {  return fullName;  }  public void setFullName(String fullName) {  this.fullName = fullName;  }  public String getConatctNumber() {  return conatctNumber;  }  public void setConatctNumber(String conatctNumber) {  this.conatctNumber = conatctNumber;  }  } |

* **Repository Layer:** ZomatoUserRepository.java

|  |
| --- |
| package com.zomato.repository;  import org.springframework.data.jpa.repository.JpaRepository;  import org.springframework.stereotype.Repository;  import com.zomato.entity.UserEntity;  @Repository  public interface ZomatoUserRepository extends JpaRepository<UserEntity, String>{  } |

**Now we are ready with User registartin Service.**

**As per Requirement, User Login REST Service should be integrated with Security Layer and as well as JWT token when User is Valid.**

**Now Add JWT functionality our applicatin, for generating and validationg token based on User Name.**

* We should add JWT librarys dependecis inside Maven **pom.xml** file, because by default Spring not provding support of JWT.

|  |
| --- |
| <dependency>  <groupId>io.jsonwebtoken</groupId>  <artifactId>jjwt-api</artifactId>  <version>0.11.2</version>  </dependency>  <dependency>  <groupId>io.jsonwebtoken</groupId>  <artifactId>jjwt-impl</artifactId>  <version>0.11.2</version>  </dependency>  <dependency>  <groupId>io.jsonwebtoken</groupId>  <artifactId>jjwt-jackson</artifactId>  <version>0.11.2</version>  </dependency> |

* **Now Create JWT Token Creator and Validation Utility class.**

|  |
| --- |
| **package** com.zomato.service;  **import** java.util.Date;  **import** java.util.HashMap;  **import** java.util.Map;  **import** org.springframework.stereotype.Component;  **import** io.jsonwebtoken.Jwts;  **import** io.jsonwebtoken.SignatureAlgorithm;  @Component  **public** **class** JWTTokenUtil {    //5 Mins  **public** **static** **final** **long** ***JWT\_TOKEN\_VALIDITY\_MILLIS*** = 5 \* 60000;  **private** String secret =  "ASFACASDFACASDFASFASFDAFASFASDAADSCSDFADCSFDAFASFASDFACASDFASFASFDAFASFASDAADSCSDFADCSFDAFAS";  // 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 token for user  **public** String generateToken(String userName) {  Map<String, Object> claims = **new** HashMap<>();  **return** doGenerateToken(claims, userName);  }  // while creating the token -  // Sign the JWT using the HS512 algorithm and secret key.  **private** String doGenerateToken(Map<String, Object> claims,  String subject) {  **return** Jwts.*builder*()  .setClaims(claims)  .setSubject(subject)  .setIssuedAt(**new** Date(System.*currentTimeMillis*()))  .setExpiration(**new** Date(System.*currentTimeMillis*()  + ***JWT\_TOKEN\_VALIDITY\_MILLIS***))  .~~signWith~~(SignatureAlgorithm.***HS512***, secret).compact();  }  // validate token  **public** Boolean validateToken(String token, String userName) {  **final** String username = getUsernameFromToken(token);  **return** (username.equals(userName)  && !isTokenExpired(token));  }  } |

* Now Create Rest Service for Login User which should be integrated via Security Layer. Once user is valid, we should send a response back to Cleint with Token value.

In Spring, We should add below Dependecies in **pom.xml** file for Spring Security Support.

|  |
| --- |
| <dependency>  <groupId>org.springframework.security</groupId>  <artifactId>spring-security-config</artifactId>  <version>5.7.10</version>  </dependency>  <dependency>  <groupId>org.springframework.security</groupId>  <artifactId>spring-security-web</artifactId>  <version>5.7.10</version>  </dependency> |

**Security Layer Setup in Spring MVC:**

**AbstractSecurityWebApplicationInitializer with Spring MVC:**

If we were using Spring MVC in our application we probably already had a **WebApplicationInitializer** that is loading our Spring MVC Configuration. We should register Spring Security with the existing **ApplicationContext**. For example, if we are using Spring MVC our **SecurityWebApplicationInitializer** would look something like the following:

* **So Add Below class in our Configuration classes.**

|  |
| --- |
| **package** com.zomato;  **import** org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;  **public** **class** SecurityWebApplicationInitializer **extends** AbstractSecurityWebApplicationInitializer {  } |

This would simply only register the **springSecurityFilterChain** Filter for every URL in your application. After that we would ensure that **WebSecurityConfig** was loaded in our existing ApplicationInitializer.

Now We have to Define User Validation Logic with Security layer Integration. To achieve these functionalities , Spring Security provided few pre-defined interfaces and Classes.

**Step 1:** Customize our Entity Class by Implementing **UserDetails** Interface of Spring Security API. So that we can directly Store Repository/Database Data of User Credentials and roles in side **UserDetails** entity. Now same Entity will be utilized by Spring Authentication and Authorization Modules internally.

Here This interface contains few pre-defined abstract methods, those should be overridden with our logic.

|  |
| --- |
| package com.zomato.entity;  import java.util.Collection;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.Id;  import javax.persistence.Table;  import org.springframework.security.core.GrantedAuthority;  import org.springframework.security.core.userdetails.UserDetails;  @Entity  @Table(name = "zomato\_users")  public class UserEntity implements UserDetails {  @Id  @Column  private String emailId;  @Column  private String password;  @Column  private String fullName;  @Column  private String conatctNumber;  public String getEmailId() {  return emailId;  }  public void setEmailId(String emailId) {  this.emailId = emailId;  }  public String getPassword() {  return password;  }  public void setPassword(String password) {  this.password = password;  }  public String getFullName() {  return fullName;  }  public void setFullName(String fullName) {  this.fullName = fullName;  }  public String getConatctNumber() {  return conatctNumber;  }  public void setConatctNumber(String conatctNumber) {  this.conatctNumber = conatctNumber;  }  @Override  public Collection<? extends GrantedAuthority> getAuthorities() {  return null;  }  @Override  public String getUsername() {  return this.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;  }  } |

**Step 2:** Now we have to Implement User Service layer with pre-defined Interface provided by Spring Security Layer.

**Interface UserDetailsService:**

This Core interface which loads user-specific data. It is used throughout the framework as a user DAO and is the strategy used by the **DaoAuthenticationProvider**. An **AuthenticationProvider** implementation that retrieves user details from a **UserDetailsService**.

So we have to implement **UserDetailsService** interfacefrom our User Service layer class.

* Implantation of Interface from **ZomatoUserService.java**

|  |
| --- |
| **package** com.zomato.service;  **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.bcrypt.BCryptPasswordEncoder;  **import** org.springframework.stereotype.Service;  **import** com.zomato.dto.UserLogin;  **import** com.zomato.dto.UserRegister;  **import** com.zomato.entity.UserEntity;  **import** com.zomato.repository.ZomatoUserRepository;  @Service  **public** **class** ZomatoUserService **implements** UserDetailsService{  @Autowired  ZomatoUserRepository repository;  //for encrypt passwords  @Autowired  BCryptPasswordEncoder passwordEncoder;    **public** String registerUser(UserRegister request) {  //Convert to Eneity Object  UserEntity entity = **new** UserEntity();  entity.setEmailId(request.getEmailId());  entity.setPassword(passwordEncoder.encode(request.getPassword()));  entity.setFullName(request.getFullName());  entity.setConatctNumber(request.getConatctNumber());  repository.save(entity);  **return** "User Created Successfully. Please Login Now.";  }  @Override  public UserDetails loadUserByUsername(String username)  throws UsernameNotFoundException {  UserEntity user = repository.findByEmailId(username).orElseThrow(()  -> new RuntimeException("User Not Found"));  return user;  }  } |

* Now we have to Add **findByEmailId()**method inside Repository:

**ZomatoUserRepository.java**

|  |
| --- |
| package com.zomato.repository;  import java.util.Optional;  import org.springframework.data.jpa.repository.JpaRepository;  import org.springframework.stereotype.Repository;  import com.zomato.entity.UserEntity;  @Repository  public interface ZomatoUserRepository extends JpaRepository<UserEntity, String>{  Optional<UserEntity> findByEmailId(String username);  } |

* Now, we have to create a Filter which is responsible for reading JWT token and validation of token for very incoming Request before it reaching to our Controller Services or endpoint methods. For this we have extend a pre-defined filter OncePerRequestFilter of Spring Security Module.
* As part of this class, we have to override doFilterInternal() method.

|  |
| --- |
| **package** com.zomato.security;  **import** java.io.IOException;  **import** javax.servlet.FilterChain;  **import** javax.servlet.ServletException;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **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.stereotype.Component;  **import** org.springframework.web.filter.OncePerRequestFilter;  **import** com.zomato.service.JWTTokenUtil;  **import** com.zomato.service.ZomatoUserService;  @Component  **public** **class** JWTTokenAuthenticationFilter **extends** OncePerRequestFilter {  @Autowired  ZomatoUserService userService;  @Autowired  JWTTokenUtil jwtTokenUtil;  @Override  **protected** **void** doFilterInternal(HttpServletRequest request,  HttpServletResponse response, FilterChain filterChain)  **throws** ServletException, IOException {  // Authorization  String token = request.getHeader("Authorization");  System.***out***.println(" Header : Authorization : " + token);    String username = **null**;  **if** (token != **null**) {  username = **this**.jwtTokenUtil.getUsernameFromToken(token.trim());  } **else** {  System.***out***.println("Invalid Header Value !! ");  }  **if** (username != **null**  && SecurityContextHolder.*getContext*().getAuthentication() == **null**) {  // fetch user detail from username  UserDetails userDetails =  **this**.userService.loadUserByUsername(username);  Boolean validateToken =  **this**.jwtTokenUtil.validateToken(token, userDetails.getUsername());  **if** (validateToken) {  // set the authentication  UsernamePasswordAuthenticationToken authentication =  **new** UsernamePasswordAuthenticationToken(  userDetails, **null**, userDetails.getAuthorities());    SecurityContextHolder.*getContext*().setAuthentication(authentication);  } **else** {  System.***out***.println("Invalid Token… !!");  }  }  filterChain.doFilter(request, response);  }  } |

**What is UsernamePasswordAuthenticationToken ?**

**org.springframework.security.authentication.UsernamePasswordAuthenticationToken**

**UsernamePasswordAuthenticationToken** is a class in Spring Security, which is a popular framework for implementing authentication and authorization in Java applications. This class is typically used for representing an authentication request where a user provides their username and password for authentication.

Here's a brief explanation of how **UsernamePasswordAuthenticationToken** works:

* **User Provides Credentials:** When a user submits their username and password for authentication, these credentials are typically collected through a login form in a web application.
* **Creation of UsernamePasswordAuthenticationToken:** The application creates a UsernamePasswordAuthenticationToken object to encapsulate the user's credentials. This token is used to represent the authentication request.
* **Authentication Process:** The UsernamePasswordAuthenticationToken is then passed to the Spring Security framework, which initiates the authentication process.
* **Authentication Provider:** Spring Security uses an authentication provider, typically a DaoAuthenticationProvider, to verify the user's credentials. This provider checks the provided username and password against the stored user credentials (usually in a database) to determine if they are valid.
* **Authentication Success or Failure:** Depending on whether the credentials are valid, the authentication process results in either success or failure. If successful, the user is considered authenticated and gains access to the protected resources. If the authentication fails, an exception is thrown or an error response is generated.
* **Security Context:** If authentication is successful, Spring Security stores the authenticated user's details in the security context. This allows the application to access information about the authenticated user during their session.
* **Authorization:** After successful authentication, Spring Security can also handle authorization, determining what actions or resources the authenticated user is allowed to access.

**SecurityContextHolder:**

**SecurityContextHolder** is a central component in Spring Security that manages the security context of an application. It allows you to access and manipulate information about the currently authenticated user and their security details throughout the lifecycle of a user's interaction with your application. This class provides a way to store and retrieve the Authentication object, which represents the currently authenticated principal (user) and their associated authorities.

* Now finally, we have to create Spring Security Context Configuration, where we should define like what kind of incoming requests should be authenticated and authorized. To make sure this Spring provided an Interface **SecurityFilterChain.**

**SecurityFilterChain:**

**SecurityFilterChain** is a core concept in Spring Security that represents a chain of filters responsible for processing HTTP requests and implementing various security features, including authentication and authorization. Each filter in the chain performs a specific security-related task, such as verifying credentials, checking access permissions, or handling session management. SecurityFilterChain is associated with a set of request patterns, such as URL patterns or antMatchers, that specify which requests should be processed by the filters within that chain. This allows you to apply different security configurations to different parts of your application.

**Create A Security Configuration class:** SecurityConfig.java

|  |
| --- |
| package com.zomato.security;  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.config.annotation.web.configuration.EnableWebSecurity;  import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  import org.springframework.security.web.SecurityFilterChain;  import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;  import com.zomato.service.ZomatoUserService;  @Configuration  @EnableWebSecurity  public class SecurityConfig {  @Autowired  JWTTokenAuthenticationFilter jwtTokenAuthenticationFilter;  @Autowired  ZomatoUserService userService;  @Bean  AuthenticationManager authenticationManager(AuthenticationConfiguration builder)  throws Exception {  return builder.getAuthenticationManager();  }  @Bean  BCryptPasswordEncoder passwordEncoder() {  return new BCryptPasswordEncoder();  }  @Bean  public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  http.csrf().disable()  .authorizeRequests().antMatchers("/create/user", "/login/user") .permitAll()  .anyRequest().authenticated()  .and().addFilterBefore(this.jwtTokenAuthenticationFilter,  UsernamePasswordAuthenticationFilter.class);  return http.build();  }  } |

* **Now Define Login REST Service in Controller with Spring Security layer with Authentication, as followed.**
* **In User Login, we will have User Name and Password.** 
  + **User Name : emailId**
  + **Password : password**

**After User Validation, as a Response we will send emailId and token Value.**

**Login Response Class**: UserLoginResponse.java

|  |
| --- |
| **package** com.zomato.dto;  **public** **class** UserLoginResponse {  **private** String emailId;  **private** String token;  **public** UserLoginResponse(String emailId, String token) {  **super**();  **this**.emailId = emailId;  **this**.token = token;  }  **public** UserLoginResponse() {  **super**();  }  **public** String getEmailId() {  **return** emailId;  }  **public** **void** setEmailId(String emailId) {  **this**.emailId = emailId;  }  **public** String getToken() {  **return** token;  }  **public** **void** setToken(String token) {  **this**.token = token;  }  } |

* **Add Login Service in Controller:** ZomatoController.java

|  |
| --- |
| **package** com.zomato.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.security.authentication.AuthenticationManager;  **import** org.springframework.security.authentication.BadCredentialsException;  **import** org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  **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.RestController;  **import** com.zomato.dto.UserLogin;  **import** com.zomato.dto.UserLoginResponse;  **import** com.zomato.dto.UserRegister;  **import** com.zomato.service.JWTTokenUtil;  **import** com.zomato.service.ZomatoUserService;  @RestController  **public** **class** ZomatoController {  @Autowired  ZomatoUserService service;    @Autowired  JWTTokenUtil jwtTokenUtil;    @Autowired  AuthenticationManager authenticationManager;  @PostMapping("/create/user")  **public** String registerUser(@RequestBody UserRegister request) {  **return** service.registerUser(request);  }  @PostMapping("/login/user")  **public** UserLoginResponse loginUser(@RequestBody UserLogin request) {  **this**.doAuthenticate(request.getEmailId(), request.getPassword());  String token = **this**.jwtTokenUtil.generateToken(request.getEmailId());  **return** **new** UserLoginResponse(request.getEmailId(), token);  }    **private** **void** doAuthenticate(String emailId, String password) {  UsernamePasswordAuthenticationToken authentication =  **new** UsernamePasswordAuthenticationToken(emailId, password);  **try** {  authenticationManager.authenticate(authentication);  } **catch** (BadCredentialsException e) {  **throw** **new** RuntimeException("Invalid UserName and Password");  }  }  } |

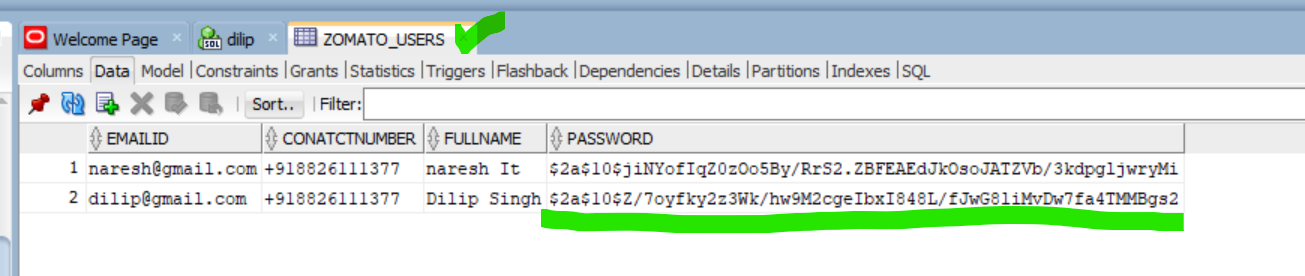
**Testing:** Let’s Test our application as per our requirement and security configuration.

Register User: Security not applicable for this endpoint i.e. to execute below endpoint, no need to provide JWT.

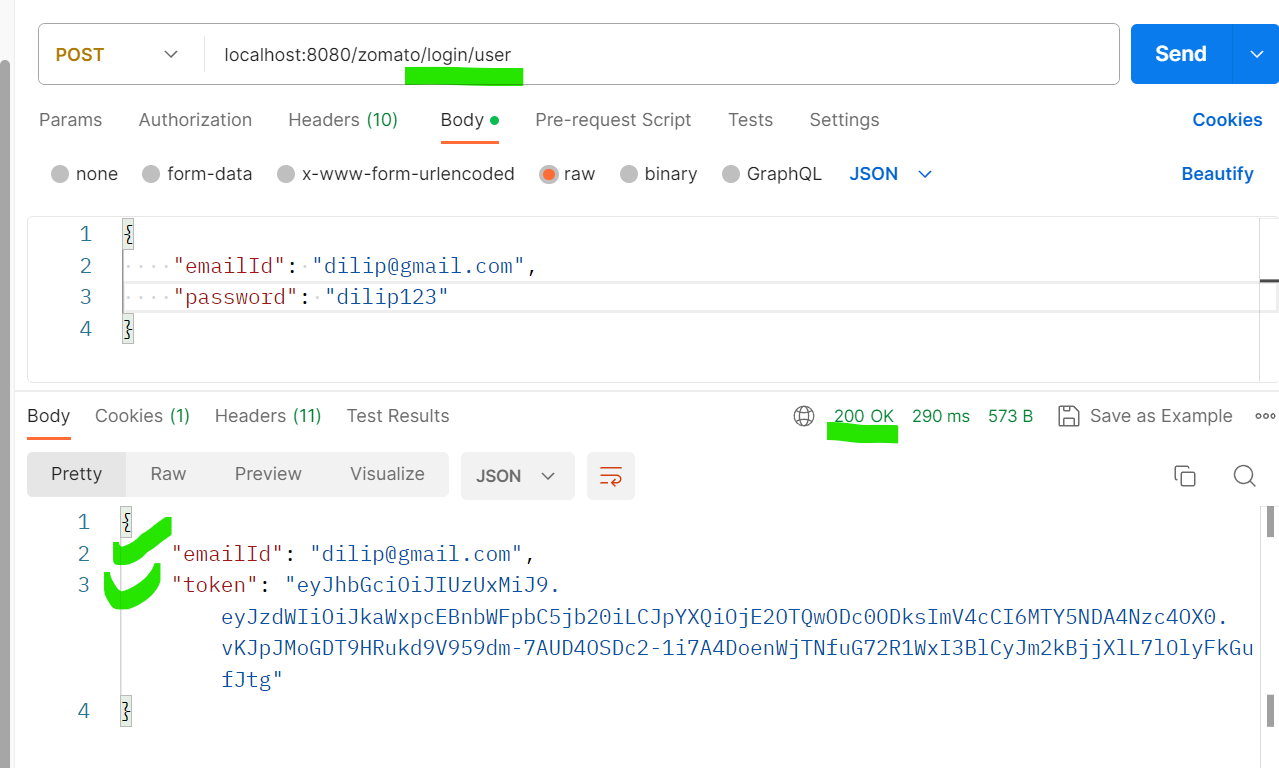
****

**Now Go and Verify in Database Table, How User Details are inserted:**

Because we are used Password Encryptor as per security Layer. Passwords are encrypted.



**Login User:** Security not applicable for this. i.e. to execute we no need to provide JWT. After User Validation, JWT Token comes as part of response.

****

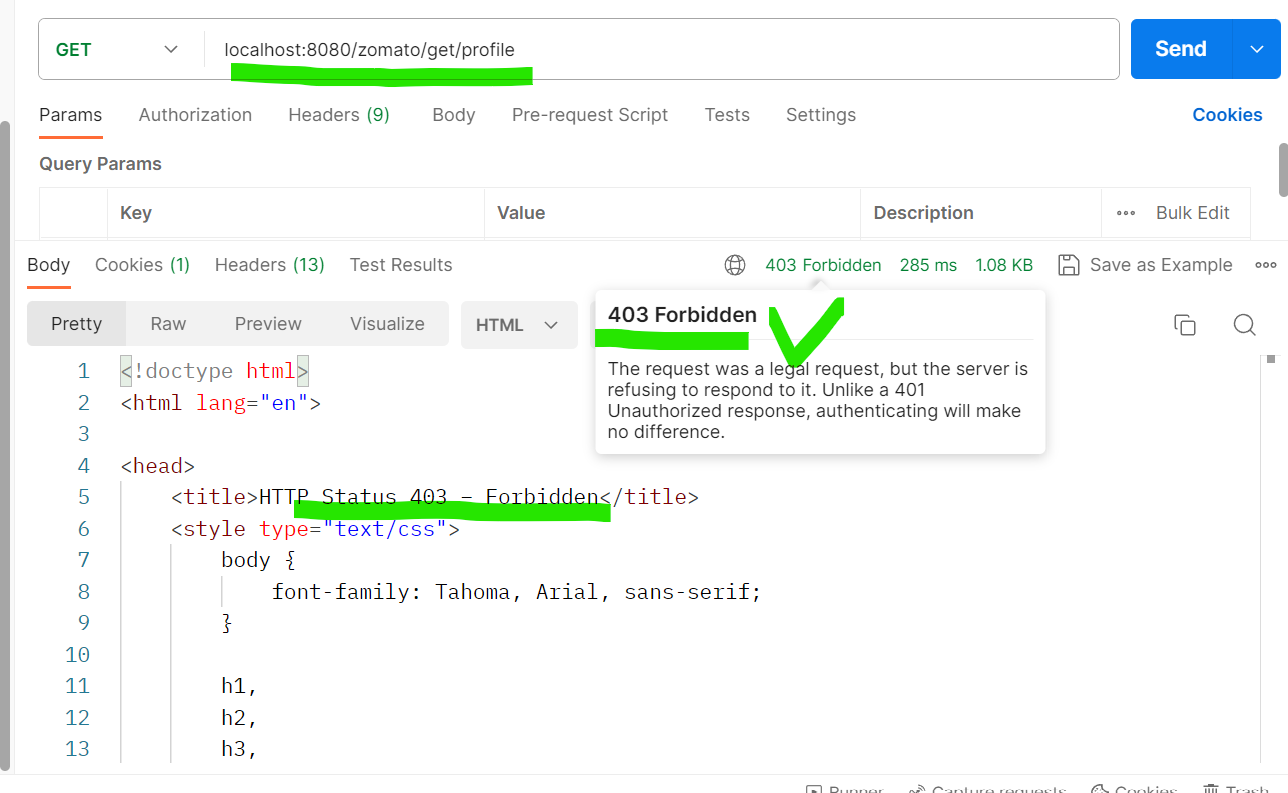
**Now Access Other REST services of Application.**

**Adding 2 Endpoints in Controller: /get/profile & /get/orders**

|  |
| --- |
| **package** com.zomato.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.security.authentication.AuthenticationManager;  **import** org.springframework.security.authentication.BadCredentialsException;  **import** org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  **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.RestController;  **import** com.zomato.dto.UserLogin;  **import** com.zomato.dto.UserLoginResponse;  **import** com.zomato.dto.UserRegister;  **import** com.zomato.service.JWTTokenUtil;  **import** com.zomato.service.ZomatoUserService;  @RestController  **public** **class** ZomatoController {  @Autowired  ZomatoUserService service;    @Autowired  JWTTokenUtil jwtTokenUtil;    @Autowired  AuthenticationManager authenticationManager;  @PostMapping("/create/user")  **public** String registerUser(@RequestBody UserRegister request) {  **return** service.registerUser(request);  }  @PostMapping("/login/user")  **public** UserLoginResponse loginUser(@RequestBody UserLogin request) {  **this**.doAuthenticate(request.getEmailId(), request.getPassword());  String token = **this**.jwtTokenUtil.generateToken(request.getEmailId());  **return** **new** UserLoginResponse(request.getEmailId(), token);  }    **private** **void** doAuthenticate(String emailId, String password) {  UsernamePasswordAuthenticationToken authentication =  **new** UsernamePasswordAuthenticationToken(emailId, password);  **try** {  authenticationManager.authenticate(authentication);  } **catch** (BadCredentialsException e) {  **throw** **new** RuntimeException("Invalid UserName and Password");  }  }    **@GetMapping("/get/profile")**  **public String getProfile() {**  **return "Welcome Dilip , Please find Your Profile Details";**  **}**  **@GetMapping("/get/orders")**  **public String getOrders(@RequestHeader String token) {**  **return "Welcome Dilip , Please find Your Order Details";**  **}**  **}** |

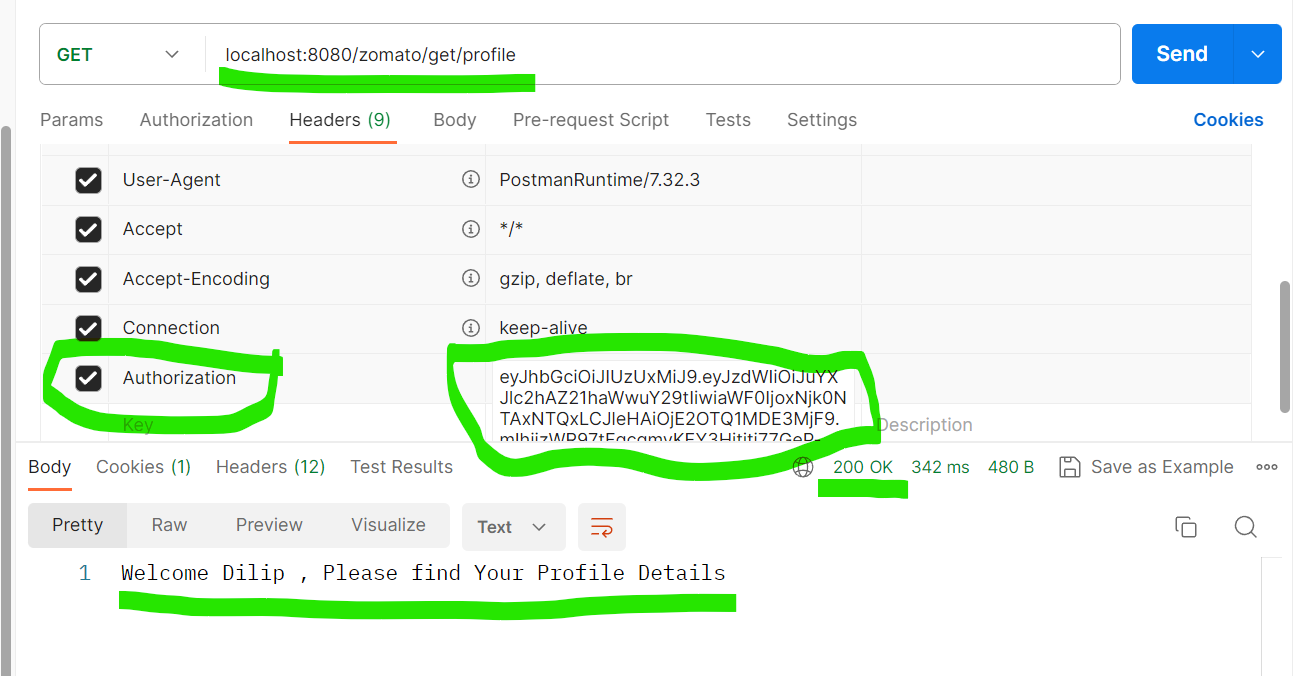
Now We have to access /get/profile and /get/orders. Let’s Do it.

**Testing:** /get/profile



We got forbidden Access, means Unauthorized. Because our application security is allowing only **/create/user** and **/login/user** without **JWT token.** Other than those two, any URI should be authenticated with JWT token. So Let’s Trigger same request again with JWT token.

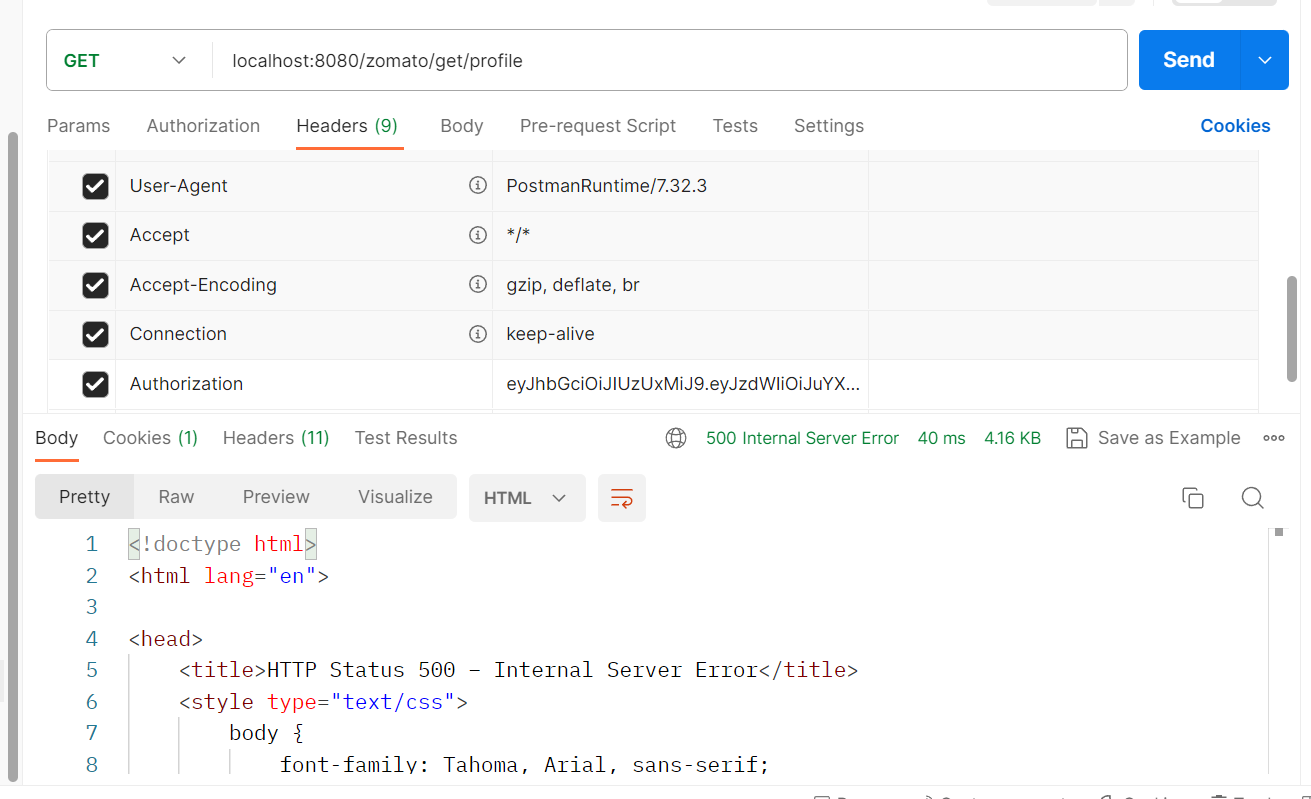
Add token in Headers : **Authorization <token>**

****

Now Token Validated and then Security Layer allowed to access out endpoint method and got expected Response.

In case If we pass an expired token, then our application will not allow to execute Endpoint logic, send response as Invalid token.

**Send Invalid/expired Token: We got Error Response, See Server Side Console Logs.**

****

**Server Logs:** We got Exception as **ExpiredJwtException** while validating token by JWT framework. So we should handle these exceptions as well to process meaningful messages to client.

