Why to Secure Rest Api's

So far, you have learned how to develop REST API. But these API's are accessible to everyone i.e. anyone can use these API's to perform CRUD operations on the data. For example, the API's we have developed for our TrainingBank application can be used by anyone get the details of any customer and in the worst case anyone can delete or update the details of a customer. But this is not correct as only customers and employees of the bank can access these API's. In other words, we need **authentication** of users of the application and only valid users should be allowed to access it.

After authentication of user you need to identify the kind of operations that the user can perform. For example, in our TrainingBank application only employees having admin access should be able to add, update and delete the customer details. This means that the access to certain API's has to be restricted based on the user's role. This calls for adding **authorization** to the application so that only users having right permission can access certain API's.

So proper **authentication** and **authorization** is required for securing REST API's. Spring Security is the most popular framework used for securing REST API's. It provides authentication and authorization for REST API's.

Before going into Spring Security, we need to understand the most common and critical vulnerabilities of a web application. For that we need to take help from some of the de-facto standards to be considered for securing applications at design, coding, and testing.

OWASP (Open Web Application Security Project)

It is a non-profitable organization which works in the area of web application and mobile security. Its objective is to make people aware of common and critical security vulnerabilities and measures to avoid those vulnerabilities. As of this course creation, OWASP Top 10 2017 is the latest vulnerabilities list is as follows:

1. Injection

Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

2. Broken Authentication

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities (temporarily or permanently).

3. Sensitive Data Exposure

Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.

4. XML External Entities (XXE)

Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal SMB file shares on unpatched Windows servers, internal port scanning, remote code execution, and denial of service attacks, such as the Billion Laughs attack.

5. Broken Access control

Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

6. Security misconfigurations

Security misconfiguration is the most common issue in the data, which is due in part to manual or ad hoc configuration (or not configuring at all), insecure default configurations, open S3 buckets, misconfigured HTTP headers, error messages containing sensitive information, not patching or upgrading systems, frameworks, dependencies, and components in a timely fashion (or at all).

7. Cross Site Scripting (XSS)

XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user supplied data using a browser API that can create JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

8. Insecure Deserialization

Insecure deserialization flaws occur when an application receives hostile serialized objects. Insecure deserialization leads to remote code execution. Even if deserialization flaws do not result in remote code execution, serialized objects can be replayed, tampered or deleted to spoof users, conduct injection attacks, and elevate privileges.

9. Using Components with known vulnerabilities

Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

10. Insufficient logging and monitoring

Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

CWE/SANS Top 25 Most Dangerous Programming Errors

This Top 25 programming errors list is the most widespread and critical errors that can lead to serious vulnerabilities in the application, which are easy to find and exploit.

The Top 25 list is a tool for educating and creating awareness to help the developer to prevent the most dangerous programming errors thereby helping them to create a more secure application.

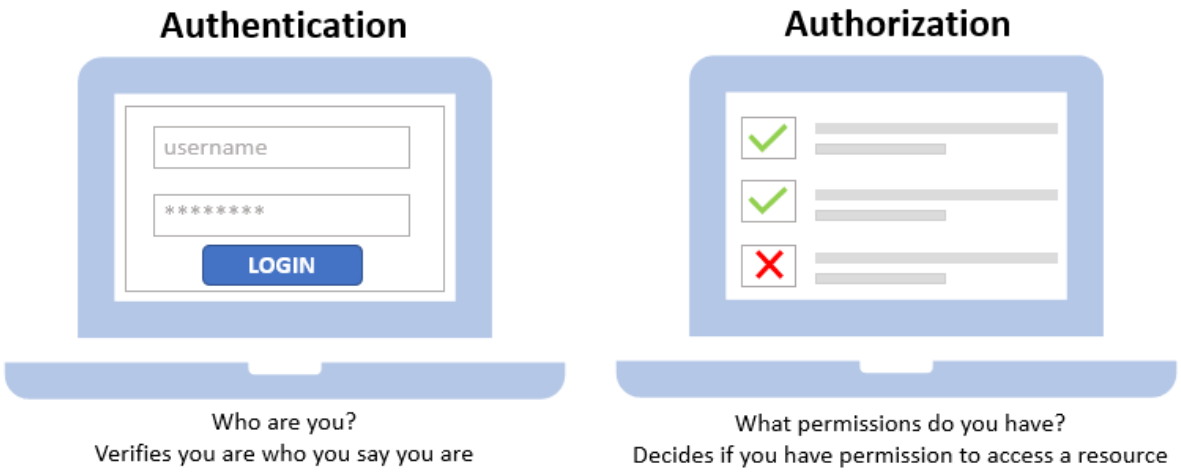
In this course we going to learn how to use Spring Security to secure Java based enterprise applications. It mainly focuses on **authentication and authorization** for securing applications. So let us learn more about authentication and authorization.

Authentication

It is the process of verifying the identity of the user i.e verify whether the user is the intended user or not. For example, swipe in/out system at turnstiles at the company entrances is the best example for authentication.

Authorization

It is the process of restricting the access to resources for authenticated users so that the person/system who is authorized to access the resource can access the same. For example, a company has many employees and all the employees can enter the company premises but entry to the server room is allowed only to few people.



Now we will learn what is Spring Security and how it is used for authentication and authorization.

Introduction to Spring Security

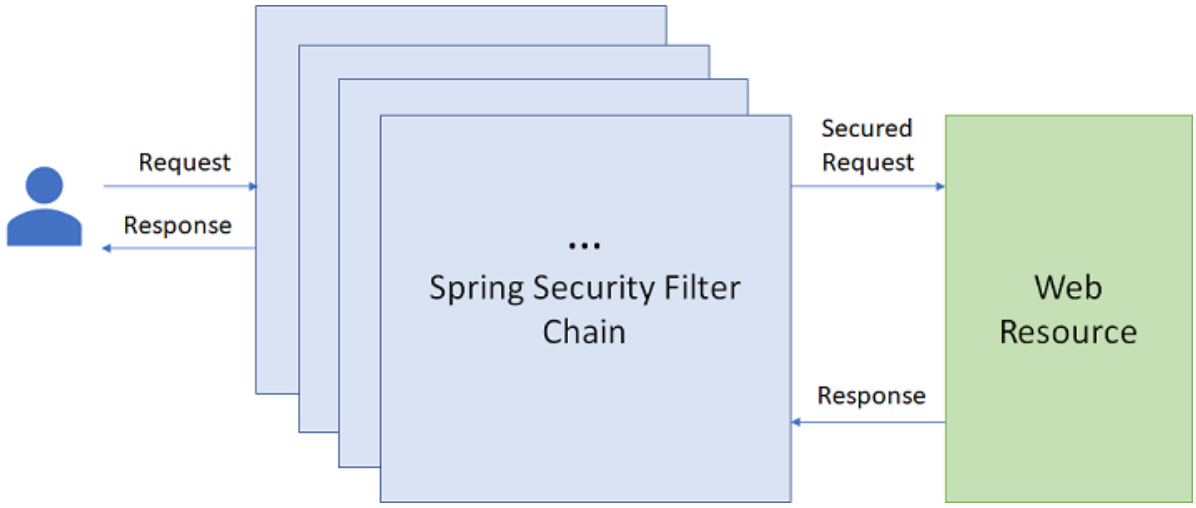
It is an open-source framework that is used by many for securing their applications in a platform-independent way. Security is applied in a simpler way using declarative programming approach with annotation-based configuration. It provides following core security services to your application

- Authentication
    - Basic authentication with default login/Http basic form
    - Authentication against database
    - Secure Password Storage
    - Authentication against LDAP

- Authorization
    - Role-based access
    - Restricting URL access
    - Method level security
    - Page-level security
- Session management
- Https channel security
- Remember me service

Spring Security also provides many sub-projects under its umbrella to support popular security standards/protocols such as OAuth, SAML,Kerberos

Spring Security is entirely based on chain of filters for authentication and authorization. Each filter is applied to every request for authentication and authorization. Any request which does not have proper credential for authentication and authorization, the request would be rejected, and an an exception will be thrown.

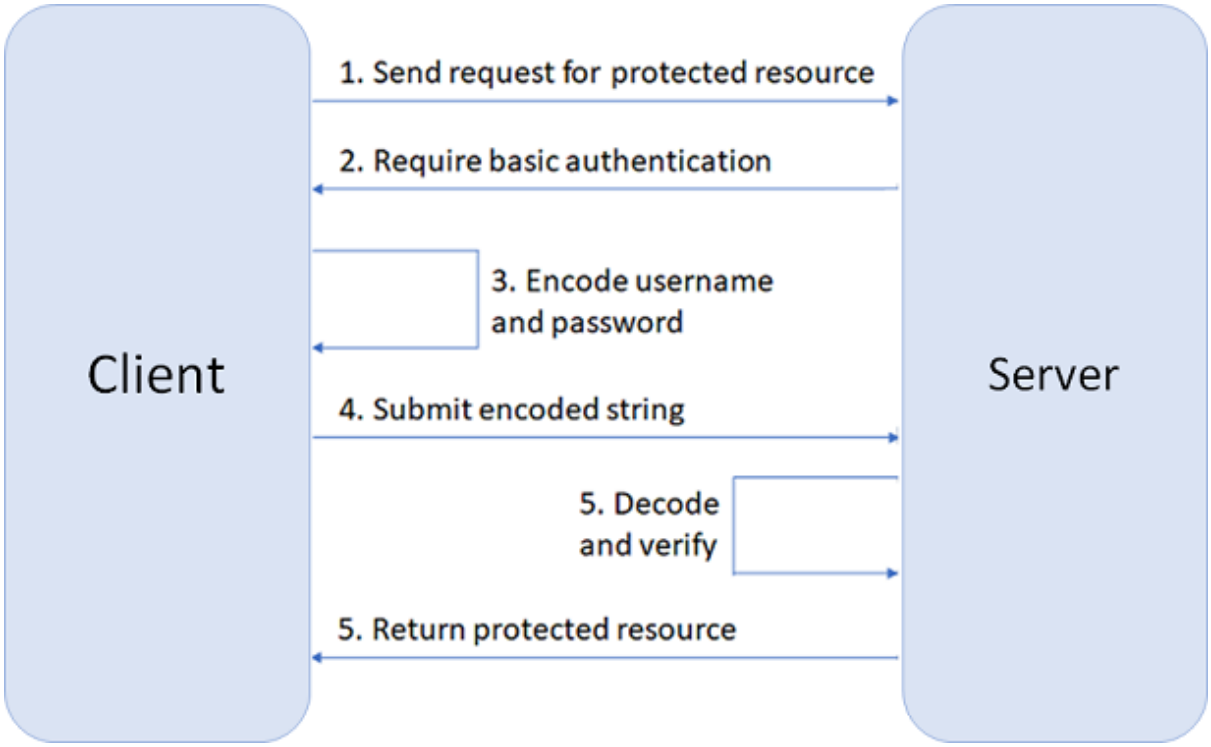


Some important filters of Spring Security filter chain are as follows :

- **UsernamePasswordAuthenticationFilter**: It performs the task of authentication.
- **AnonymousAuthenticationFilter:** It performs anonymous authentication. If authentication credentials are not present in request it creates an anonymous users. These users are allowed to access API"s which does not require authentication.
- **ExceptionTranslationFilter:** It translates security exceptions to a proper HTTP response.
- **FilterSecurityInterceptor:** It performs authorization of resources.

Spring Security provides multiples approaches for authentication and authorization. One of the approach is HTTP basic authentication. Now let us learn about this approach and then we will how it is used with Spring Security to secure REST API.

In this approach, when client requests for a protected resource the server asks the client the authentication credentials. The client send their Base64 encoded authentication credentials along with request using HTTP authorization header. On receiving the credentials, the server decodes and validates them. On successful validation the client gets the requested resource. The following figure shows steps of basic authentication:



This type of authentication does not mandate us to have any login page which is the very standard way of authenticating the client requests in any web application. So, this approach is useful when we have web services interacting with each other. In addition, there is no need to store the credentials or any related information in the session. Each request here, is treated as an independent request which in turn promotes scalability. In a Spring Boot application when Spring Security is enabled basic authentication is default authentication approach which is auto configured.

Now let us see how to use Spring Security in Spring Boot project to secure REST API.

Using Spring Security in Spring Boot application

To use Spring Security in a Spring Boot application add the following dependency in pom.xml file:

```
1. <dependency>
2.       <groupId>org.springframework.boot</groupId>
3.       <artifactId>spring-boot-starter-security</artifactId>
4. </dependency>
```

On seeing this dependency in classpath, Spring Boot configures default security configuration for whole application. It provides following security features:

- All requests will be authenticated using HTTP basic authentication.
- The credentials for accessing the application will be automatically generated with only one user. There is no specific roles or authorities assigned to user. The default user name is user, and the default password is generated in a random fashion during the application start. You can get this password from the console log of the application as shown below:

```
Using generated security password: 48382fb7-72b3-476e-8e51-56d717ace9e5
```

You can also define user name, password and role using following properties in application.properties file:

- spring.security.user.name - This property defines the default user name.
- spring.security.user.password - This property defines password of default user name.
- spring.security.user.roles - This property defines defines granted role for default user name.

In the previous demo, we created REST API's for TrainingBank application. Now let us secure these API's using Spring Security.

Objective :

To understand to secure REST API using Spring Security default configuration.

Steps :

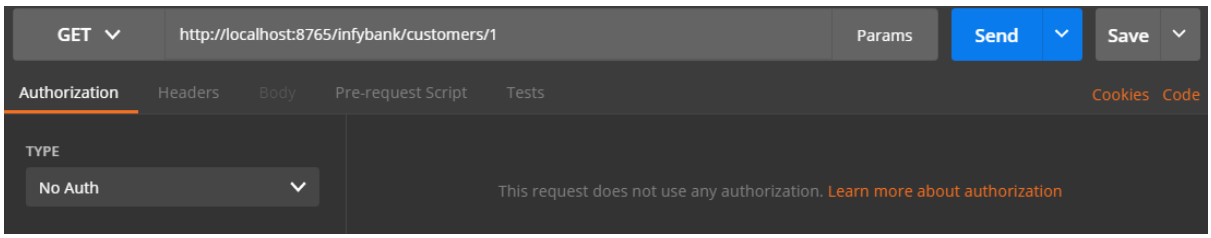
Step 1 : Import the Spring REST Validation demo in Eclipse.

Step 2 : Add the following dependency in pom.xml file :

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

Step 3 : Execute the application to deploy the API.

Step 4: Open Postman and send GET request to URL http://localhost:8000/bank/customers/1 as shown below:

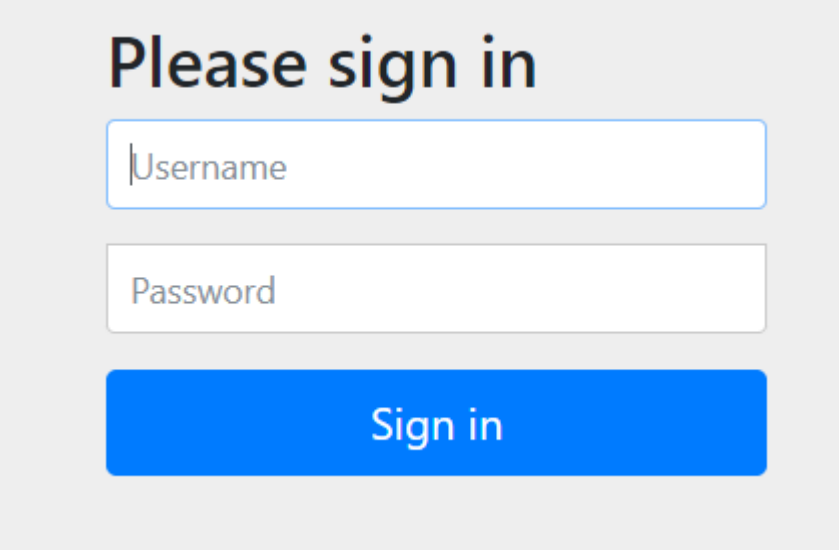


You will get the following response:

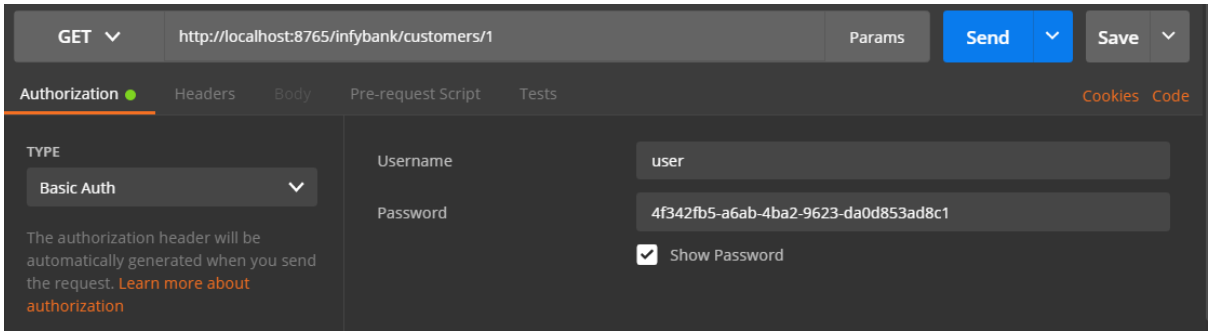
```
{
    "timestamp": "2020-08-04T10:08:17.580+00:00",
    "status": 401,
    "error": "Unauthorized",
    "message": "Unauthorized",
    "path": "/bank/customers/1"
}
```

The response status is 401 - Unauthorized. So you have seen that access to API is blocked as user is not authenticated.

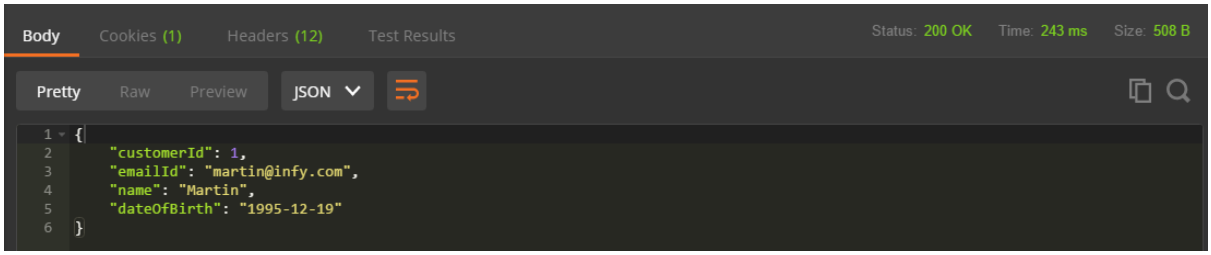
Note : If you are using browser to test the application, you will get following page to enter the credentials:



Step 5 : Now send the request with credentials for authentication. For this click on authorization tab and select Type as Basic Auth. Enter username as **user** and password as printed in Eclipse console and send GET request to URL http://localhost:8765/trainingbank/customers/1 as shown below:



Step 6 : Now after authentication you will get following response:



Securing REST API using HTTP Basic Authentication

You have seen that how to secure REST API's using Spring Boot default configuration. But it is not powerful enough to secure real world applications. So, you have to create your own custom configuration to override the default configuration. To do this the first you have to decide how you want security to behave. For example, we want to have following configurations for CustomerAPI of our TrainingBank application:

- The application should be accessible to following two users :

Username	Password	Role
tim	tim123	GUEST
smith	smith123	ADMIN

These credentials can be present in different data sources such as a database or an active directory like LDAP etc. Spring Security provides support for authentication using different data sources. In this course we will use in-memory data source where username, password and role of a user will be stored in memory where application is running.

- All the requests should be authenticated using HTTP basic authentication.
- The /customers API should be accessed by only users having ADMIN role.

This can be done in following 3 steps:

1. Create custom configuration class
2. Implement authentication
3. Implement authorization

To create custom security configuration class you have to extend WebSecurityConfigurerAdapter class and annotate it with @EnableWebSecurity annotation as follows:

```
1. @EnableWebSecurity
2. public class SecurityConfig extends WebSecurityConfigurerAdapter
   {
3. //rest of the code
4. }
```

In above code,

- **@EnableWebSecurity** enables web security support provided by Spring Security.
- **WebSecurityConfigurerAdapter** class provides methods which has to overridden to implement custom security requirements.

After creating custom security configuration class you have to specify data source where user credentials are stored for authentication. For this you have to override **protected void configure(AuthenticationManagerBuilder auth)** method WebSecurityConfigurerAdapter class in SecurityConfig class. In this method you have to configure the data source of user credentials using **AuthenticationManagerBuilder**. For using in-memory data source and to add smith and tim as users of the application you have to override this method as follows:

```
1. @Override
2. protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
3.    auth.inMemoryAuthentication()
4.          .withUser("smith")
5.          .password("{noop}smith123").roles("ADMIN")
6.          .and()
7.          .withUser("tim")
8.          .password("{noop}tim123").roles("USER");
9. }
```

In Spring Security 5.x onwards the password has to be encrypted before being stored. It does not allow plain text password without specifying the password encryption algorithm. So you have to

specify the password encryption algorithm before begin stored. But If you dont want to encode the password then it has to be preceeded with **{noop}** which uses plain text NoOpPasswordEncoder encoder. This encoder does not encrypt the password and is useful for testing where encryption of password is not necessary. But in real life applications it is good to encrypt the password as unencrypted passwords are insecure. For this Spring Security provides encoders which implements **PasswordEncoder** interface. One of the encoder is **BCryptPasswordEncoder** which we will use. To use this configure following bean in SecurityConfig class:

```
1. PasswordEncoder passwordEncoder() {
2.   return new BCryptPasswordEncoder();
3. }
```

Once password encoder is configured you can use it while configuring user store as follows:

```
1. @Override
2. protected void configure(AuthenticationManagerBuilder auth)
   throws Exception {
3.    auth.inMemoryAuthentication()
4.          .withUser("smith")
5.
    .password(passwordEncoder().encode("smith123")).roles("ADMIN")
6.          .and()
7.          .withUser("tim")
8.
    .password(passwordEncoder().encode("tim123")).roles("USER");
9. }
```

In above code,

- encode() method is used to encode the password.

After authentication you have to implement authorization. For this you have to override **configure(HttpSecurity http)** method to specify URLs which should be secured and which should not. Using object of HTTPSecurity you can define how to handle security at the web level. To authorize requests to all users you can override this method as follows:

```
1. protected void configure(HttpSecurity http) throws Exception{
2.        http
3.           .authorizeRequests()
4.             .anyRequest()
5.             .authenticated()
6.             .and()
7.             .httpBasic();
8.        http.csrf().disable();
9. }
```

In above code,

- authorizeRequests() method returns an object of ExpressionInterceptUrlRegistry using which URL paths and their security requirements are specified.
- The Spring security by default enables CSRF(Cross Site Request Forgery) protection. Because of this you can access only GET endpoints. For accessing PUT, POST and

DELETE endpoints, you have to disable the CSRF protection. So this is disabled usign the below code:

```
1. http.csrf().disable();
```

Note: Instead of disbaling CSRF you can also send CSRF token along with the request which is beyond the scope of this course. CSRF is disabled usually if REST API is consumed by non-browser based REST clients.

To authorize requests to URI trainingbank/customers only to users having ADMIN role you can override this method as follows:

```
1. protected void configure(HttpSecurity http) throws Exception {
2.    http
3.      .authorizeRequests()
4.       .antMatchers("/trainingbank/customers/**")
5.       .hasRole("ADMIN")
6.     .anyRequest()
7.     .authenticated()
8.     .and()
9.     .httpBasic();
10.    http.csrf().disable();
11. }
```

In above code,

- The URI to be protected is mentioned using antMatchers() method. In URI "/trainingbank/customers/**", ** means any value after /trainingbank/customers.
- the hasRole() method specifies the role of the user.
- anyRequest().authenticated() means any request mapped to this URI will be authenticated.
- httpBasic() specifies that HTTP basic authentication has to be used.

You can also specify the HTTP method along with the URI as follows:

```
1. protected void configure(HttpSecurity http) throws Exception {
2.    http
3.      .authorizeRequests()
4.       .antMatchers(HttpMethod.POST, "/trainingbank/customers/**")
5.       .hasRole("CUSTOMER")
6.     .anyRequest()
7.     .authenticated()
8.     .and()
9.     .httpBasic();
10.    http.csrf().disable();
11. }
```

In above code,

- The users with role CUSTOMER can access only the POST endpoint with URI "/trainingbank/customers/**".

Securing REST API using HTTP Basic Authentication – Demo

Objective :

To understand how to secure REST API using Spring Security custom configuration and HTTP basic authentication.

Steps:

Step 1 : Import the previous demo in Eclipse.

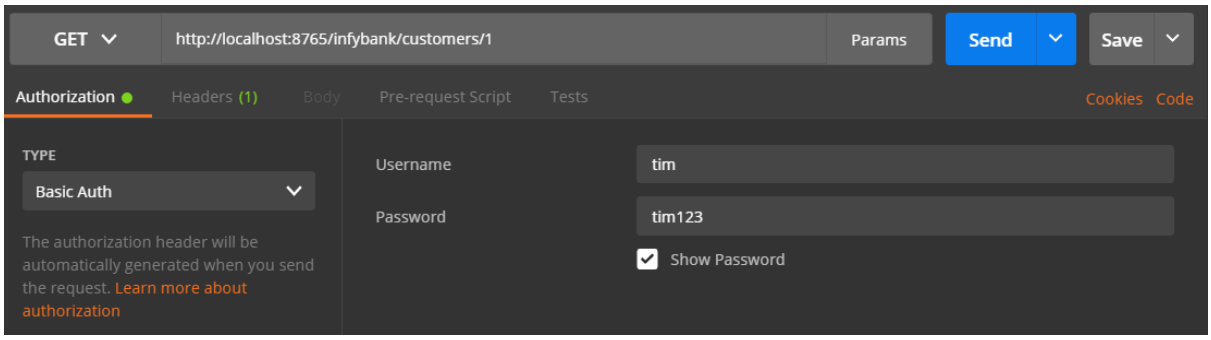
Step 2 : Add the following class in com.training.security package :

```
1. package com.training.security;
2.
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5. import
   org.springframework.security.config.annotation.authentication.bui
   lders.AuthenticationManagerBuilder;
6. import
   org.springframework.security.config.annotation.web.builders.HttpS
   ecurity;
7. import
   org.springframework.security.config.annotation.web.configuration.
   EnableWebSecurity;
8. import
   org.springframework.security.config.annotation.web.configuration.
   WebSecurityConfigurerAdapter;
9.
10.  import
   org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
11.  import
   org.springframework.security.crypto.password.PasswordEncoder;
12.
13.  @EnableWebSecurity
14.  public class SecurityConfig extends WebSecurityConfigurerAdapter
   {
15.
16.          @Override
17.          protected void configure(AuthenticationManagerBuilder
   auth) throws Exception {
18.
     auth.inMemoryAuthentication().withUser("smith").password(passwordE
   ncoder().encode("smith123")).roles("ADMIN")
19.
     .and().withUser("tim").password(passwordEncoder().encode("tim123"))
   .roles("USER");
20.          }
21.
22.          @Override
23.          protected void configure(HttpSecurity http) throws
   Exception {
24.
     http.authorizeRequests().antMatchers("/trainingbank/customers/**").hasRo
   le("ADMIN").anyRequest().authenticated().and().httpBasic();
25.                   http.csrf().disable();
26.          }
```

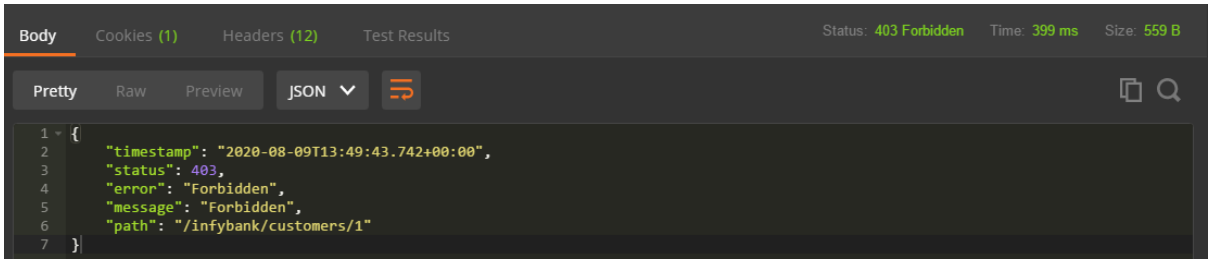
```
27.
28.         @Bean
29.         PasswordEncoder passwordEncoder() {
30.                 return new BCryptPasswordEncoder();
31.         }
32. }
```

Step 3 : Execute the application to deploy the API.

Step 4 : Open Postman and send GET request to URL http://localhost:8765/trainingbank/customers/1 with username as tim and password tim123 as shown below:



Step 5 : You will get following response as tim is not authorized to access the API:



Step 6 : Now send GET request to URL http://localhost:8765/trainingbank/customers/1 with username as smith and password smith123. You will get following response: