Importance of Security

- **Ensuring Robust Authentication and Authorization**: Implementing strong security measures helps safeguard applications from unauthorized access.
- **Comprehensive Security Measures**: Applications should be protected at multiple levels to prevent vulnerabilities.
- **Security as a Growing Concern**: Modern applications face increasing risks from various attack vectors, making it essential to stay proactive.

TELUSKO

OWASP Top Ten (2021)

The **Open Web Application Security Project (OWASP)** highlights the most critical security risks for web applications. The 2021 Top 10 list includes:

1. **Broken Access Control**
2. **Cryptographic Failures** (e.g., use of deprecated algorithms like MD5, SHA1)
3. **Injection Attacks**:
    - Example: Using unvalidated input in SQL queries, such as: SELECT * FROM users WHERE username = 'userInput';
    - Risk: Queries that allow username = 'navin' OR 1=1 can expose all user data.
    - **Prevention**: Use prepared statements to prevent SQL injection.
4. **Insecure Design**
5. **Security Misconfiguration** (e.g., default configurations in use)
6. **Vulnerable and Outdated Components**
7. **Identification and Authentication Failures**
8. **Software and Data Integrity Failures**
9. **Security Logging and Monitoring Failures**
10. **Server-Side Request Forgery (SSRF)**

Note:

You can see the latest top 10 OWASP attacks: **OWASP Top 10 Attacks**

TELUSKO

Creating a Spring Security Project

To start a basic Spring Security project:

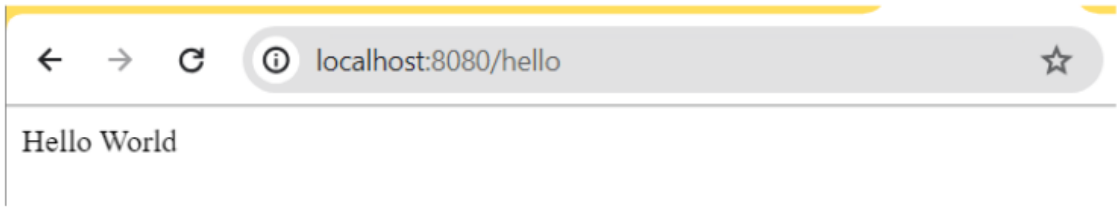
Controller Setup:

```
package com.spring.security.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
    @GetMapping("/hello")
    public String greet() {
        return "Hello World";
    }
}
```

Initial Behavior:

- By default, any user can access this endpoint without restrictions.

Illustrate the default access with an image showing a successful "/hello" endpoint call.



- **Next Step**: To secure this endpoint so that only authenticated users can access it, enable Spring Security.

TELUSKO

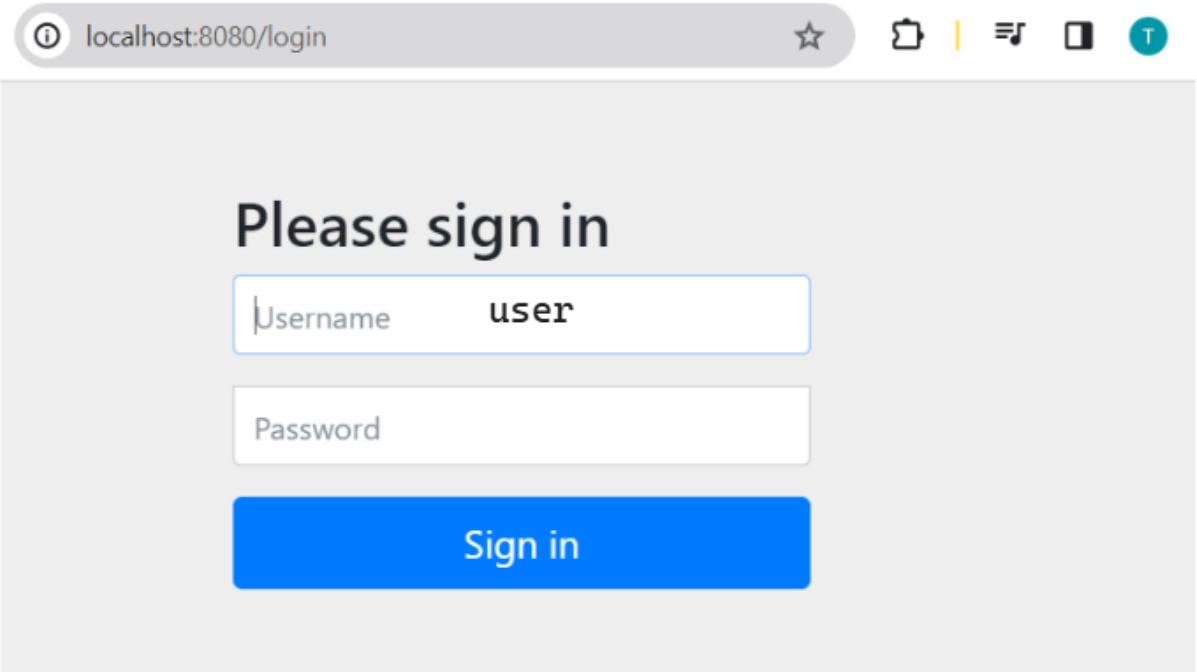
Default Login Form in Spring Security

Add spring-security dependency into your application:

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

When you add Spring Security as a dependency:

- **Automatic Features**:
    1. All APIs become private.
    2. A default login form is generated for user authentication.
- **Initial Password**: Spring Security generates a default password on application startup (e.g., fed7bbd5-e48d-4f2a-8cff-d2672d6324ca).

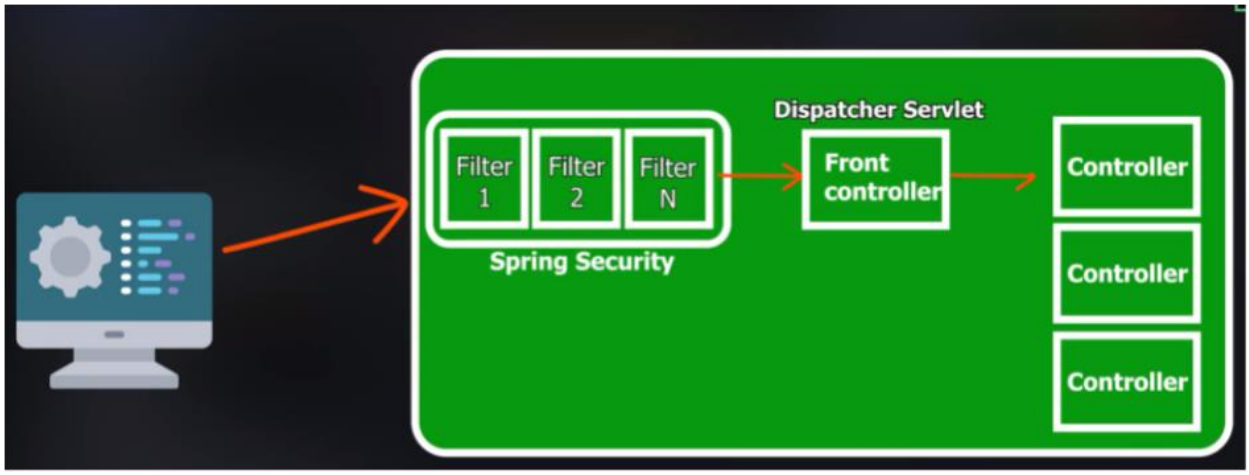


Visualizing the Security Flow

Spring Security integrates seamlessly with the application's front controller and intercepts requests through filters:

TELUSKO

- **Filter Chain**: Requests pass through multiple security filters before reaching the controller.



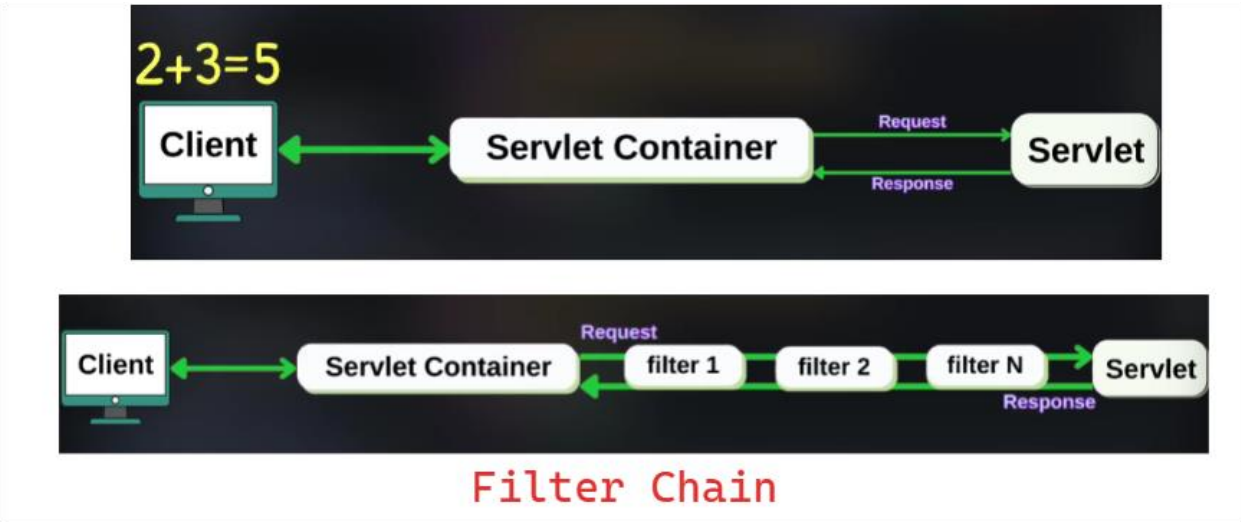
By implementing these concepts, your Spring Boot application will have robust security controls that protect against common vulnerabilities.

TELUSKO

Spring Security Filter Chain

- **Overview**: The filter chain is a sequence of filters that handle requests before they reach the controller. These filters process authentication, authorization, and other security-related tasks.
- **Flow**:

1. Client sends a request.
2. The request passes through various security filters (Filter 1, Filter 2, etc.).
3. The servlet container processes it, and then it reaches the servlet/controller.



- Spring security used multiple filters like **DefaultSecurityFilterChain**, **WebAsyncManagerIntegrationFilter**, **SecurityContextHolderFilter**, **LogoutFilter**, etc.

TELUSKO

Session ID

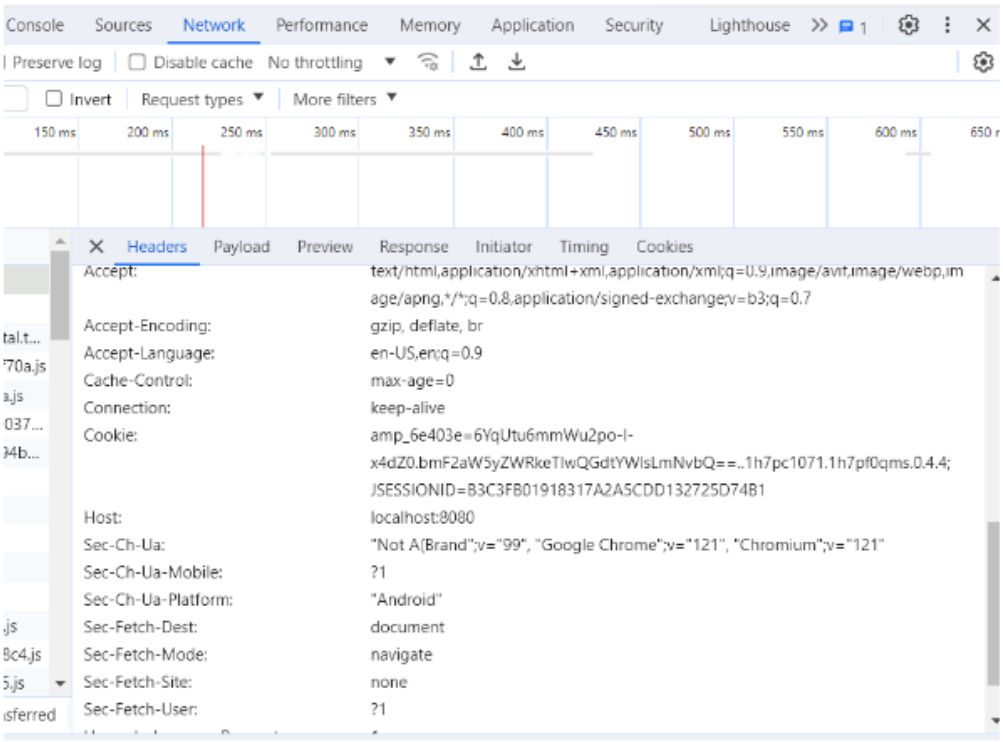
What is a session ID ?: A session ID is a unique identifier generated by the server for each user session. It helps manage user state across multiple requests.

Example Code:

```
package com.spring.security.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import jakarta.servlet.http.HttpServletRequest;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String greet(HttpServletRequest request) {

        return "Hello World " + request.getSession().getId();\

    }

    @GetMapping("/about")
    public String about(HttpServletRequest request) {

        return "About Page Session ID: " + request.getSession().getId();

    }
}
```

- **Note**: Each time a user logs in, a different session ID is generated to maintain unique sessions.

TELUSKO



Everytime we login we get different Session ID

Telusko B3C3FB01918317A2A5CDD132725D74B1

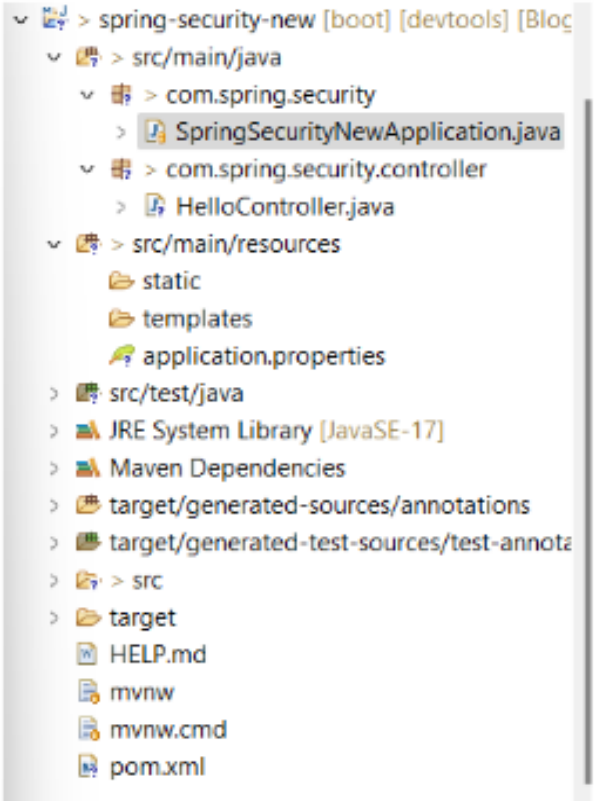
TELUSKO

Setting Username and Password

Configuration: You can set a basic username and password in **application.properties.**

```
spring.security.user.name=shiva
spring.security.user.password=root
```

- **Important**: Hardcoding passwords is discouraged in production.Use environment variables or a secure vault.
- **Scenario**: For REST APIs, consider implementing a more dynamic form of authentication rather than hardcoded credentials.



TELUSKO

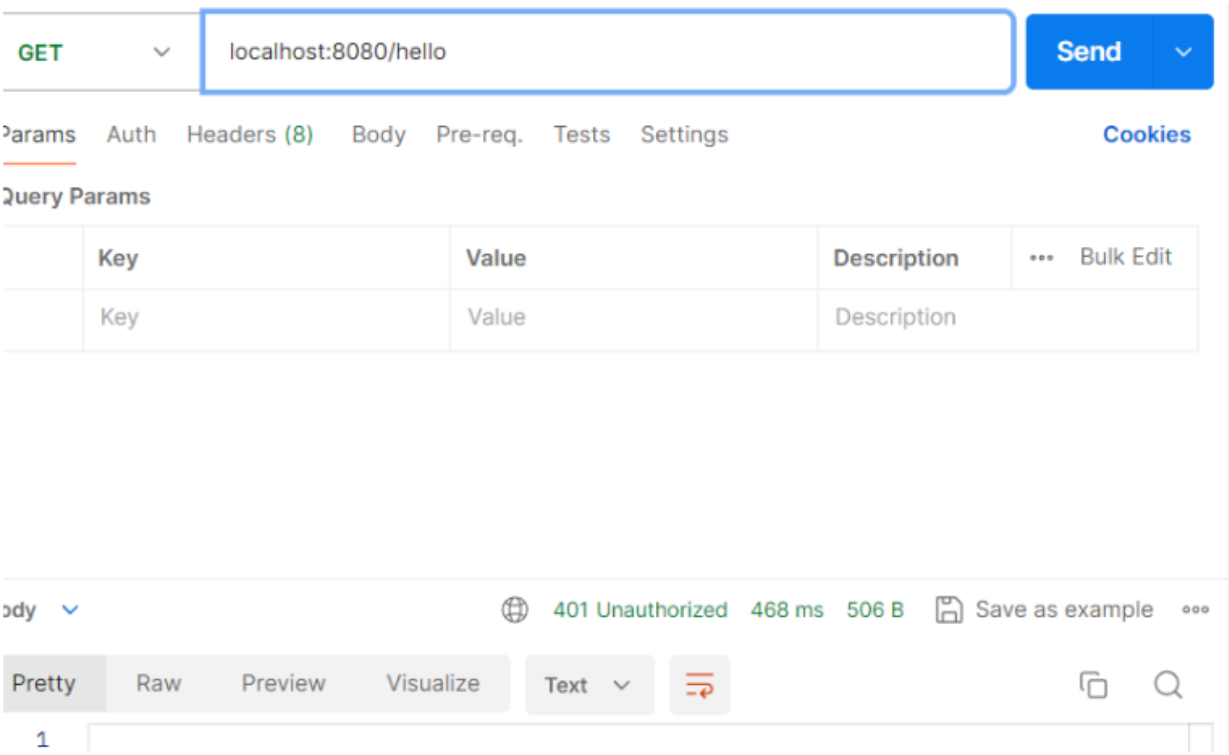
Basic Auth Using Postman

Testing with Postman:

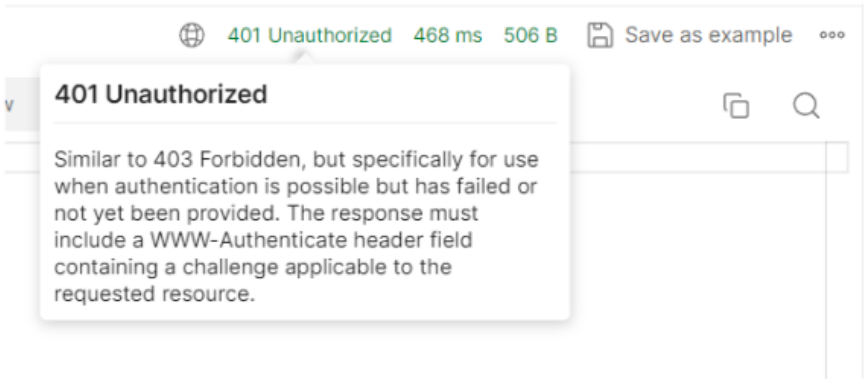
- When accessing secured endpoints without authentication, you receive a 401 Unauthorized error.
- To send authenticated requests, use Postman's Basic Authorization feature:
    1. Go to the Authorization tab.
    2. Select **Basic Auth** and provide the username and password.

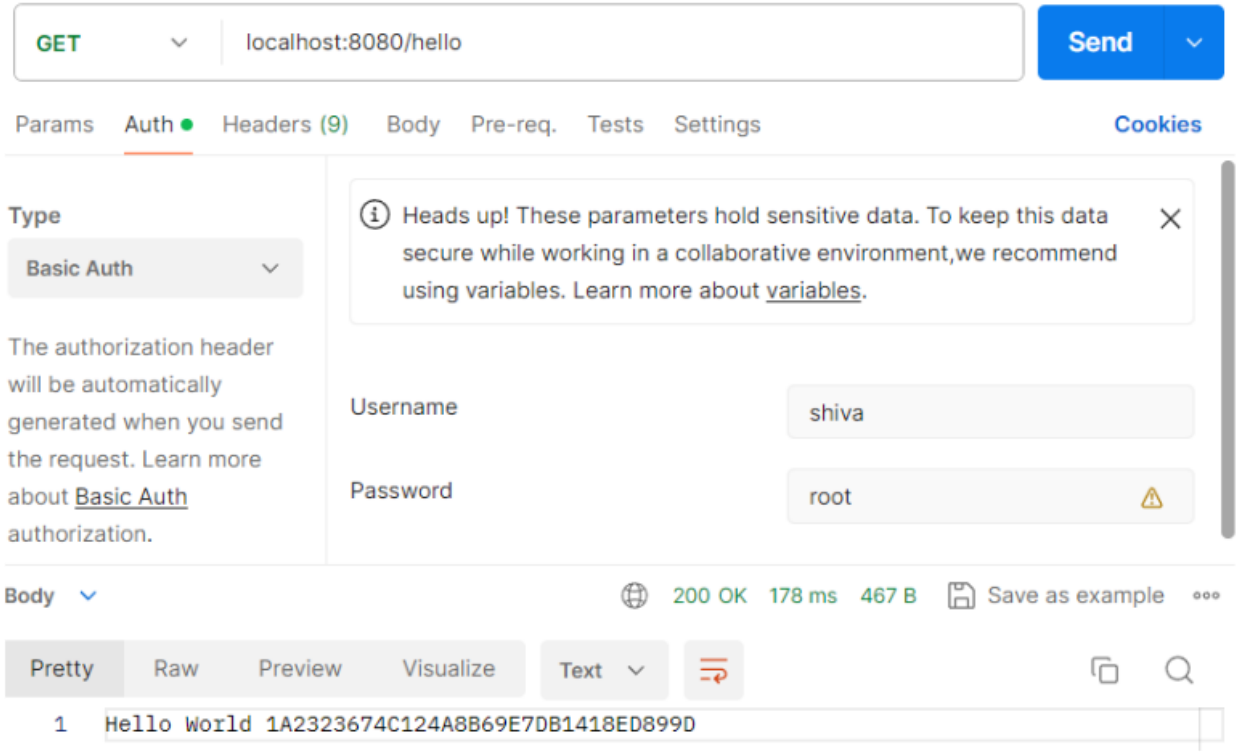
Example Request:

- Without authentication: 401 Unauthorized
- With authentication: Success response with data.



TELUSKO


401 Unauthorized 468 ms 506 B
Save as example
401 Unauthorized
Similar to 403 Forbidden, but specifically for use when authentication is possible but has failed or not yet been provided. The response must include a WWW-Authenticate header field containing a challenge applicable to the requested resource.


GET
localhost:8080/hello
Send
Params
Auth
Headers (9)
Body
Pre-req.
Tests
Settings
Cookies
Type
Basic Auth
The authorization header will be automatically generated when you send the request. Learn more about Basic Auth authorization.
Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment,we recommend using variables. Learn more about variables.
Username
shiva
Password
root
Body
200 OK 178 ms 467 B
Save as example
Pretty
Raw
Preview
Visualize
Text
1 Hello World 1A2323674C124A8B69E7DB1418ED899D

TELUSKO

What is CSRF?

CSRF (Cross-Site Request Forgery) is a web security vulnerability that tricks a user into performing an unwanted action on a website they are currently authenticated to.

How it works:

1. **Malicious Website:** A malicious website (e.g., a phishing site or an ad) contains a hidden link or form that targets the victim's legitimate website.
2. **User Action:** The user is tricked into visiting the malicious website or clicking on a link/button.
3. **Unauthorized Request:** The malicious website sends a request (e.g., transferring funds, changing passwords) to the victim's website in the background, using their active session.
4. **Action Performed:** Since the victim is already logged in, the website authenticates the request and performs the action, without the victim's knowledge or consent.

Example:
Imagine you're logged into your bank's website. A malicious website might contain a hidden form that attempts to transfer money from your account to the attacker's account. If you visit the malicious website, the form is submitted in the background, and your bank might unknowingly process the transaction.

Prevention Techniques:

- **Same-Site Cookies:** Browsers now support the "SameSite" attribute for cookies, which restricts cookies to only be sent with requests from the same origin as the one that set the cookie.
- **CSRF Tokens:** Implementing CSRF tokens adds a unique, unpredictable value to every request. The server checks this token to ensure the request originated from the expected source.
- **HTTP Strict Transport Security (HSTS):** Enforces HTTPS connections, making it harder for attackers to intercept and modify requests.

TELUSKO

Key Takeaways:

- CSRF is a serious security threat that can have significant financial and data loss implications.
- Implementing robust security measures like Same-Site cookies and CSRF tokens is essential for protecting web applications from this vulnerability.

- By default spring security will implement CSRF for POST, PUT & DELETE request types but not for GET because, it not perform any changes.

TELUSKO

Sending CSRF Token

Purpose of the getCsrfToken Method:

- This method is used to fetch the Cross-Site Request Forgery (CSRF) token from the HttpServletRequest object.
- It retrieves the token stored under the attribute "_csrf" in the request.

CSRF Token Retrieval:

- CSRF tokens are a security mechanism to prevent unauthorized or malicious actions on behalf of authenticated users.

In this implementation, the token is extracted using:

```
return (CsrfToken) request.getAttribute("_csrf");
```

Endpoint:

- The endpoint "/csrf-token" is mapped to this method, which allows clients (e.g., frontend applications) to retrieve the CSRF token as needed for secure communications.

Practical Use Case:

- The retrieved token can be sent to the client (e.g., in the response body or headers) and is expected to be included in subsequent requests requiring CSRF protection.

Spring Security Integration:

- This method assumes that Spring Security is configured to handle CSRF tokens automatically, where tokens are generated and managed for each session or request.
- The attribute "_csrf" is a standard key used by Spring Security to store the CSRF token.

TELUSKO

Usage in Client Requests:

- When performing actions like POST, PUT, or DELETE, the client must include this token in the request headers (e.g., X-CSRF-Token) or as a hidden form field.

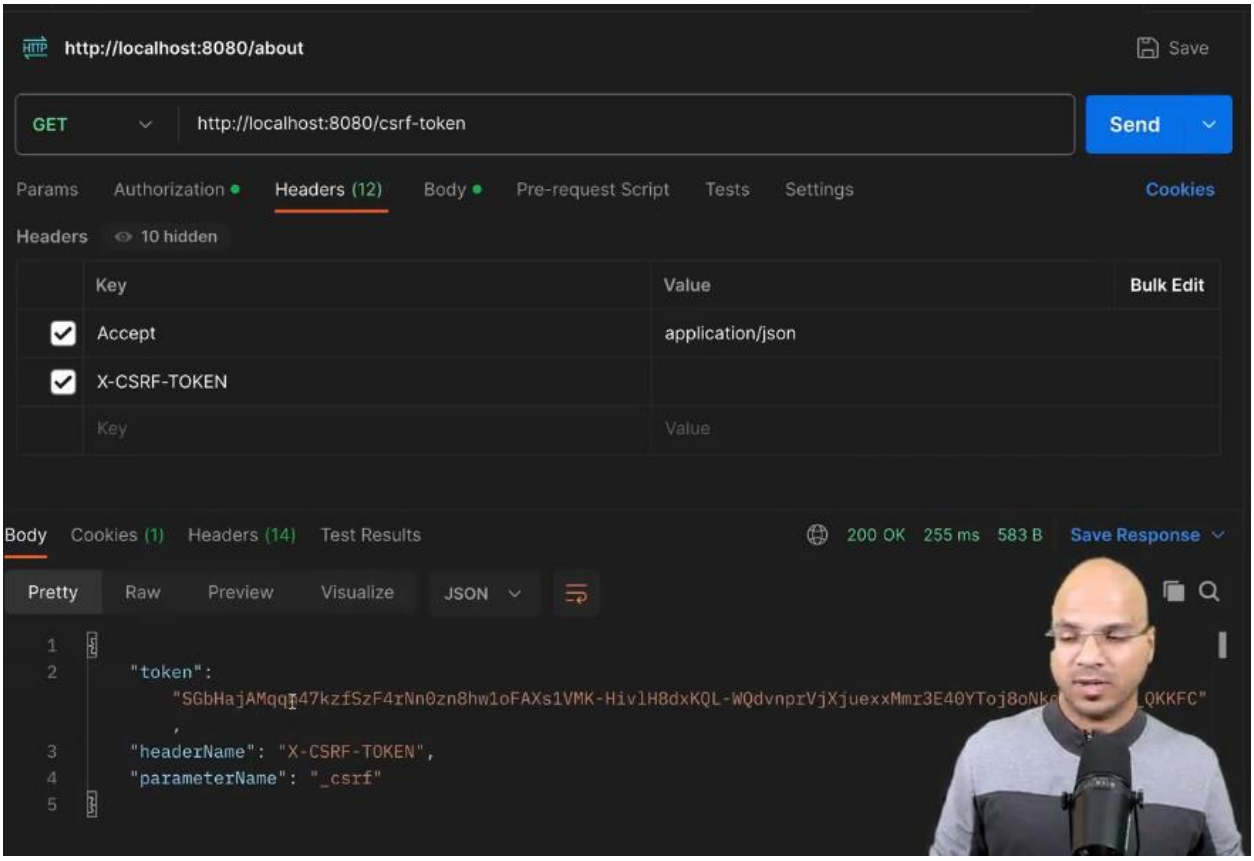
Security Best Practices:

- Ensure that this endpoint ("/csrf-token") is not exposed to unauthorized access.
- Use HTTPS to protect the transmission of the CSRF token.
- Avoid caching responses from this endpoint, as CSRF tokens are session-specific.

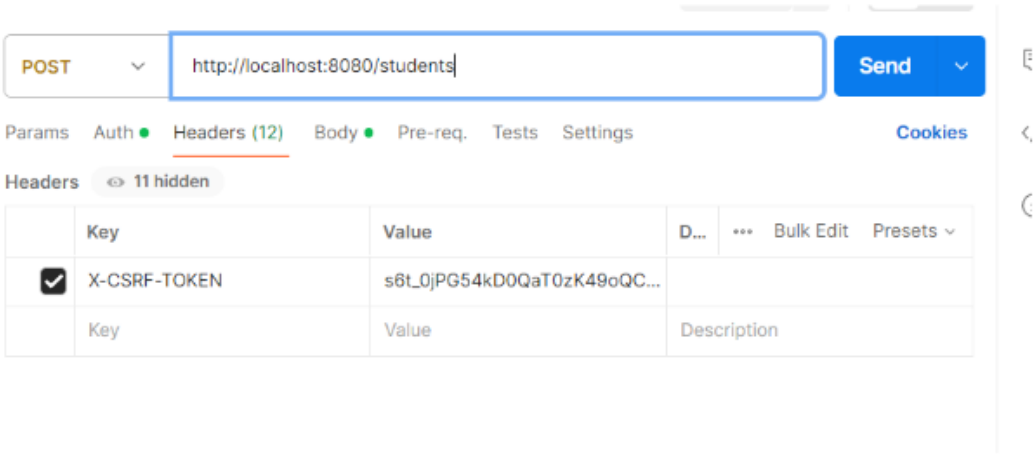
StudentsController.java

```
@RestController
public class StudentController {

    @Autowired
    private StudentService studentService;

    @GetMapping("/csrf-token")
    public CsrfToken getCsrfToken(HttpServletRequest request) {
        return (CsrfToken) request.getAttribute("_csrf");
    }

    @GetMapping("/students")
    public ResponseEntity<List<Student>> getStudents(){
        return ResponseEntity.ok(studentService.getStudents());
    }

    @PostMapping("/students")
    public ResponseEntity<Student> addStudent(@RequestBody Student student ){
        if(studentService.addStudent(student)) {
            return ResponseEntity.ok(student);
        }else {
            return ResponseEntity.internalServerError().build();
        }
    }

}
```

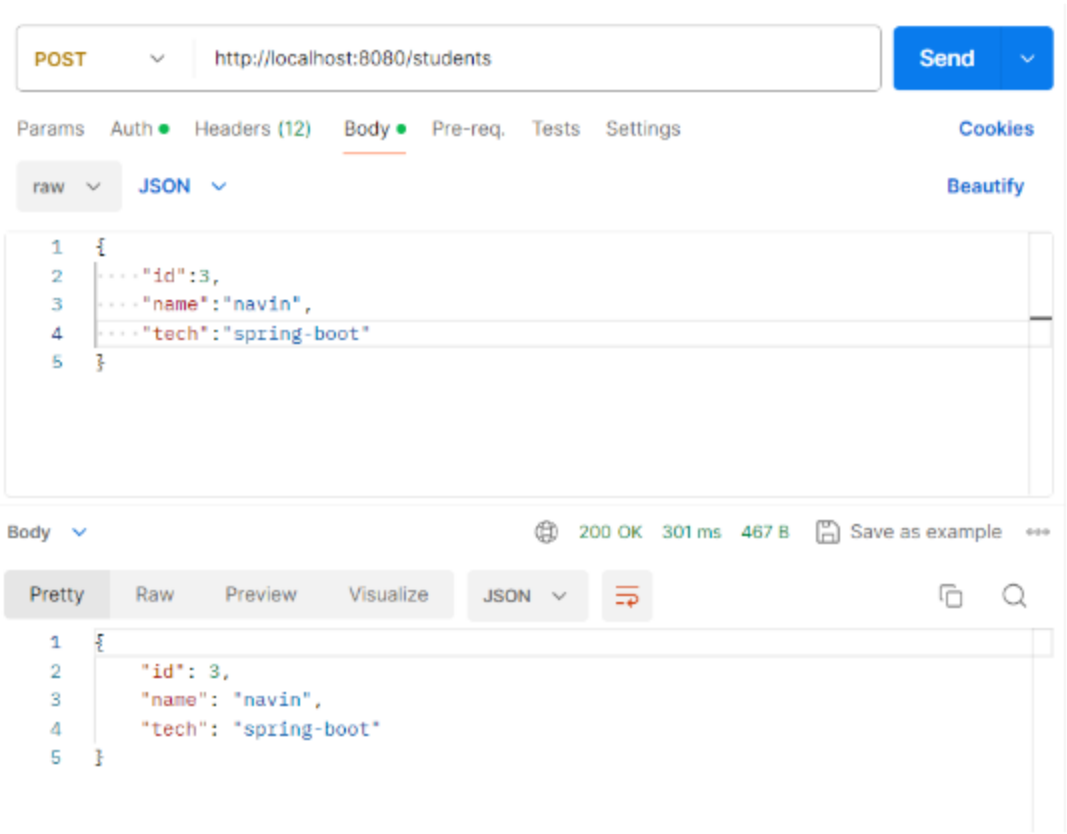
TELUSKO



Now after adding csrf token:



TELUSKO


POST
http://localhost:8080/students
Send
Params
Auth
Headers (12)
Body
Pre-req.
Tests
Settings
Cookies
raw
JSON
Beautify
{
"id":3,
"name":"navin",
"tech":"spring-boot"
}
Body
200 OK
301 ms
467 B
Save as example
Pretty
Raw
Preview
Visualize
JSON
{
"id": 3,
"name": "navin",
"tech": "spring-boot"
}

TELUSKO

Same Site Strict

In application.properties

server.servlet.session.cookie.same-site=strict

Setting the **server.servlet.session.cookie.same-site** property to **"strict"** provides an important security benefit by enhancing protection against Cross-Site Request Forgery (CSRF) attacks.

When a cookie is set with the **"SameSite"** attribute set to **"strict"**, the browser will only include the cookie in a request if the request originated from the same site as the one that set the cookie. This helps prevent malicious websites from making requests to your server on behalf of the user, thereby reducing the risk of CSRF attacks.

By setting the **"SameSite"** attribute to **"strict"**, you're ensuring that cookies containing session information are only sent in requests that originate from your own site. This helps protect sensitive user data and prevents unauthorized access to user sessions. Overall, it strengthens the security posture of your web application.

Rest Application: Most of time Stateless

TELUSKO

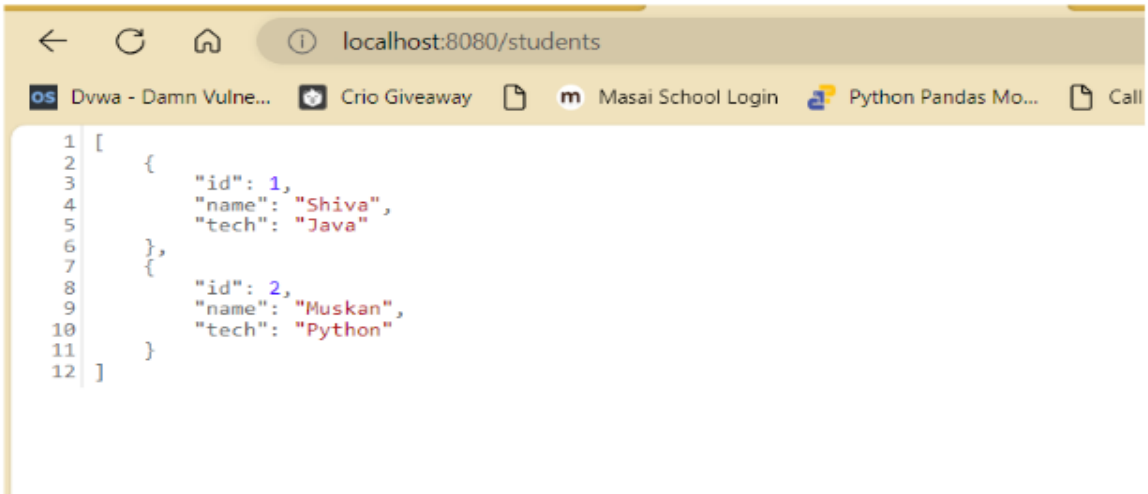
Security Configuration

The provided SecurityConfig class demonstrates how to configure security for a Spring Boot application using a custom implementation of the SecurityFilterChain. This approach allows for fine-grained control over HTTP security configurations.

SecurityConfig.java

```
package com.spring.security.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        return http.build();
    }
}
```

Note: After creating an object of SecurityFilterChain on your own. And writing my own logic. I found that they do not ask for authentication.



TELUSKO

Disabling CSRF Token

Approaches to Configure Security

Your images show two approaches to configuring security and disabling CSRF:

1. **Lambda Way**:
    - Configuration uses functional programming with lambdas for concise and readable syntax.
2. **Imperative Way**:
    - Configuration uses traditional imperative-style method calls.

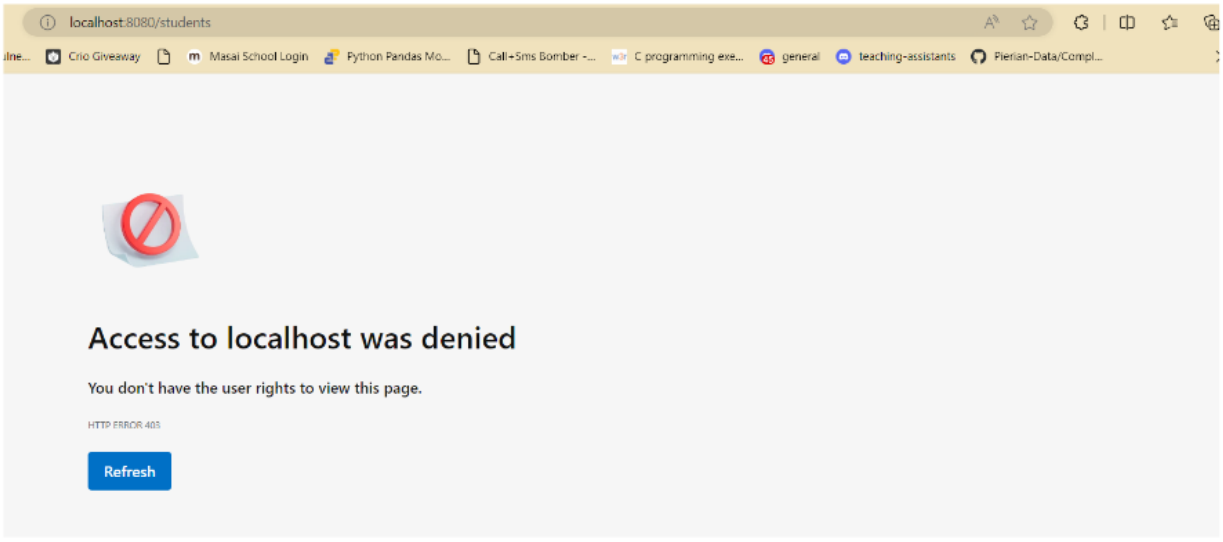
Both achieve the same result but differ in style.

Lambda Way Configuration

Example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
      http
         .csrf(csrf -> csrf.disable())  // Disabling CSRF
         .authorizeHttpRequests(request -> request.anyRequest().authenticated());
               //Allowing only authenticated requests

      return http.build();
   }
}
```

TELUSKO



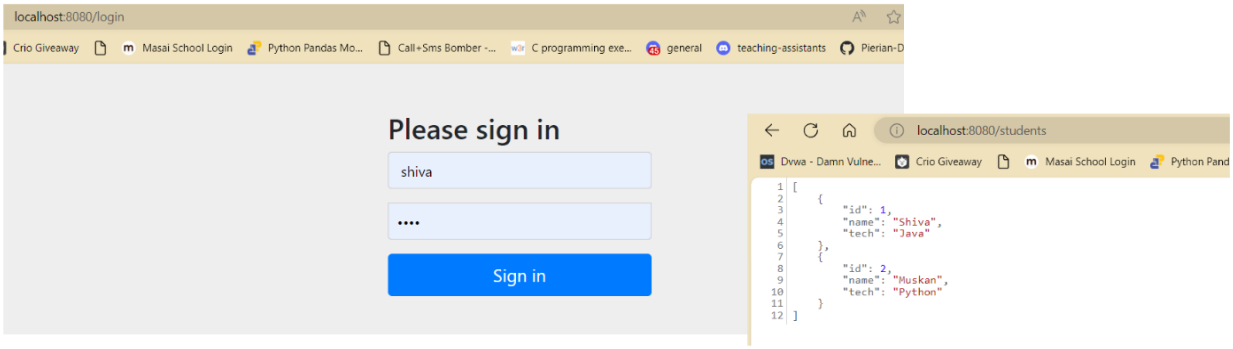
Behavior:

- **CSRF Disabled**:
  - No CSRF token is required for POST, PUT, or DELETE requests.
- **Authentication Required**:
  - All endpoints require authentication.
- **Output**:
  - If accessed without proper credentials, an "Access Denied" error is shown.

Example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

	@Bean
	public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
		http.csrf(customizer->customizer.disable());
		http.authorizeHttpRequests(request->request.anyRequest().authenticated());
		http.formLogin(Customizer.withDefaults());

		return http.build();
	}
}
```

TELUSKO



Behavior:

- **CSRF Disabled**:
  - Like the lambda approach, CSRF tokens are not needed.
- **Form-Based Authentication**:
  - Users are prompted with a login form for authentication.
- **Output**:
  - Upon accessing secured endpoints without credentials, users see a login page instead of "Access Denied."

Stateless Configuration (Sessionless)

Example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
      http
         .csrf(csrf -> csrf.disable()) // Disable CSRF protection

         // Configure authentication
         .authorizeHttpRequests(request -> request.anyRequest().authenticated())

         // Add Basic Authentication
         .httpBasic()
```

TELUSKO

```
        // Stateless sessions
        .and()
        .sessionManagement(session -> session
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    return http.build();
  }
}
```

Behavior:

- **Stateless Sessions**:
  - The server does not maintain session information.
  - Common for REST APIs, where each request is authenticated independently.
- **Output**:
  - REST APIs become simpler and more compatible with token-based authentication systems like JWT.

Key Takeaways:

1. **CSRF Disabled**:
   - Removing CSRF makes APIs easier to use for stateless services.
   - Ensure other forms of authentication (e.g., token-based) are in place.
2. **Lambda vs Imperative**:
   - Both styles work; choose based on readability and team preferences.
3. **Stateless Configuration**:
   - Suitable for REST APIs.
   - Requires enabling SessionCreationPolicy.STATELESS to avoid session tracking.
4. **Form Login vs Basic Auth**:
   - **Form Login**: Preferred for user-facing web applications.
   - **Basic Auth**: Best for APIs or development/testing.

TELUSKO

Without Lambda

1. **Lambda-Based Configuration:**
    - The first block of code (commented out) demonstrates configuring the HttpSecurity object using lambdas.
    - The lambda expressions simplify the code by allowing concise, in-line customization.
    - Code like http.csrf(csrfCustomizer -> csrfCustomizer.disable()); and http.authorizeHttpRequests(request -> request.anyRequest().authenticated()); show how easy it is to express configurations using lambda expressions.
2. **Without Lambda Configuration:**
    - The second block of code (also commented out) shows an equivalent configuration without using lambdas.
    - A Customizer implementation is explicitly created and passed to methods like http.csrf() and http.authorizeHttpRequests().
    - This approach is more verbose and may be preferable for those who need clearer readability or are working in environments without lambda support.
3. **Functional Configuration (Lambda-based):**
    - The last block of code is an active configuration using lambda expressions for better readability and conciseness.
    - Each configuration aspect (e.g., disabling CSRF, configuring HTTP basic authentication, and session management) is compactly expressed.

TELUSKO

SecurityConfig.java

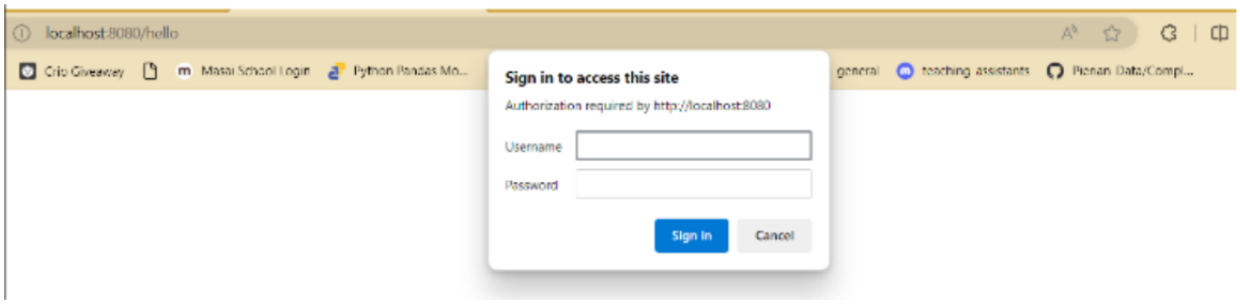
```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

//	@Bean
//	public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
//		http.csrf(customizer->customizer.disable());
//		http.authorizeHttpRequests(request->request.anyRequest().authenticated());
////		http.formLogin(Customizer.withDefaults());
//		http.httpBasic(Customizer.withDefaults());
//		http.sessionManagement(session->session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
//
//		return http.build();
//	}



//without lambda
//	@Bean
//	public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
//
//		Customizer<CsrfConfigurer<HttpSecurity>> custCsrf= new Customizer<CsrfConfigurer<HttpSecurity>>() {
//			@Override
//			public void customize(CsrfConfigurer<HttpSecurity> configure) {
//			 configure.disable();
//			}
//		};
//		http.csrf(custCsrf);
//
//
	Customizer<AuthorizeHttpRequestsConfigurer<HttpSecurity>.AuthorizationManagerRequestMatcherRegistry> custHttp= new Customizer<AuthorizeHttpRequestsConfigurer<HttpSecurity>.AuthorizationManagerRequestMatcherRegistry>() {
//			@Override
//			public void customize(AuthorizeHttpRequestsConfigurer<HttpSecurity>.AuthorizationManagerRequestMatcherRegistry request) {
//				request.anyRequest().authenticated();
//			}
//		};
//		http.authorizeHttpRequests(custHttp);
//
//		return http.build();
//	}
```

TELUSKO

```
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(customizer->customizer.disable())
          .authorizeHttpRequests(request->request.anyRequest().authenticated())
          .httpBasic(Customizer.withDefaults())
          .sessionManagement(session- >
              session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

        return http.build();
    }

}
```

TELUSKO

Getting Ready for User Database:



- For every request, a new session ID is created. It means a stateless way of sending a request.
- Now the second problem We do not have more than one user, but we need multiple users with different permissions.
- Now, for that we need a database in memory or a MySQL database.

TELUSKO

Working with Multiple Users

Overview:

When using Spring Security, if no specific UserDetailsService is defined, the framework uses a default configuration. However, you can define your own UserDetailsService to manage custom users with different roles and authorities.

Code Example for Custom In-Memory UserDetailsService:

```
@Bean
public UserDetailsService userDetailsService() {
    UserDetails user1 = User.withDefaultPasswordEncoder()
            .username("muskan")
            .password("root")
            .roles("USER")
            .build();

    UserDetails user2 = User.withDefaultPasswordEncoder()
            .username("admin")
            .password("root")
            .roles("ADMIN")
            .build();

    return new InMemoryUserDetailsManager(user1, user2);
}
```

Explanation:

- **UserDetailsService Bean**:
  - A UserDetailsService bean is created to define custom users.
  - The User class is a convenient way to create UserDetails instances.
- **User.withDefaultPasswordEncoder()**:
  - Creates a user with a password encoder that stores passwords in plain text (not recommended for production).
- **User Definitions**:
  - Two users are defined:
    - muskan with the role USER.
    - admin with the role ADMIN.

TELUSKO

- **InMemoryUserDetailsManager**:
    - An in-memory implementation of UserDetailsService that takes the UserDetails objects (user1 and user2) and stores them.

Complete SecurityConfig.java Example:

```
package com.spring.security.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.configurers.CsrfConfigurer;
import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

   @Bean
   public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
      http.csrf(Customizer.withDefaults())
         .authorizeHttpRequests(request -> request.anyRequest().authenticated())
         .httpBasic(Customizer.withDefaults())
         .sessionManagement(session ->
            session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
      return http.build();
   }

   @Bean
   public UserDetailsService userDetailsService() {
      UserDetails user1 = User.withDefaultPasswordEncoder()
            .username("muskan")
            .password("root")
            .roles("USER")
            .build();
```

TELUSKO

```
        UserDetails user2 = User.withDefaultPasswordEncoder()
                .username("admin")
                .password("root")
                .roles("ADMIN")
                .build();

        return new InMemoryUserDetailsManager(user1, user2);
    }
}
```

Notes:

- **@EnableWebSecurity**: Enables web security configuration.
- **UserDetailsService**: Interface to provide user details to Spring Security.
- **Security Filter Chain**:
    - Configures HTTP security settings like CSRF, basic authentication, and session management.
- **In-Memory Users**:
    - Ideal for quick setups, testing, or demos. For production, consider using a database-backed UserDetailsService.

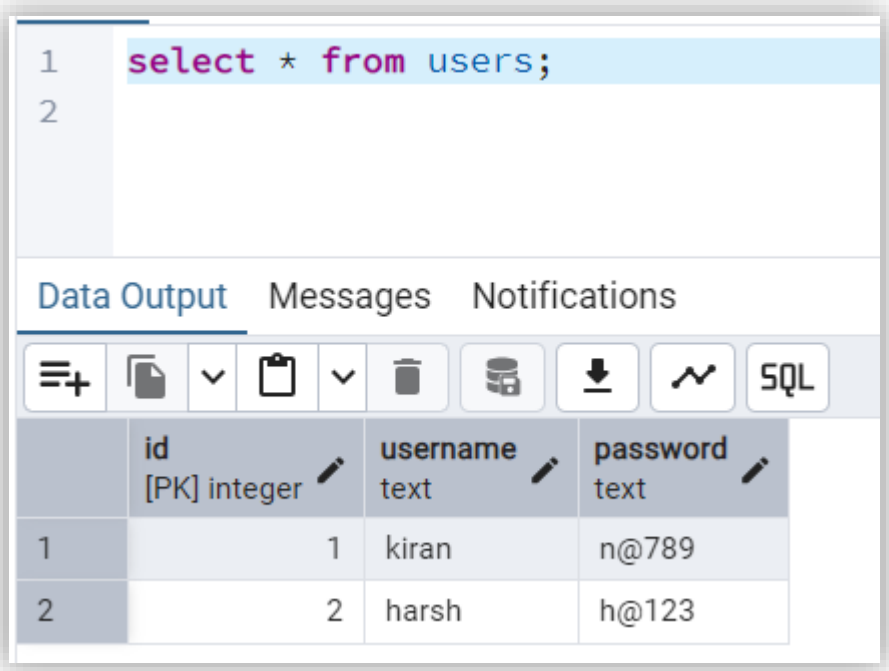
TELUSKO

Creating User Table And DB Properties

Let's see how to set up Spring Security with database authentication

- **First, we need a database table to store user information. We'll create a users table with columns for:**
  - id (primary key)
  - username
  - password

- **We'll add records in users table**



- **Database Configuration in application.properties**

  We need to tell our Spring application how to connect to the PostgreSQL database:

```
# PostgreSQL Database Configuration
spring.datasource.url=jdbc:postgresql://localhost:5432/telusko
spring.datasource.username=postgres
spring.datasource.password=0000
spring.datasource.driver-class-name=org.postgresql.Driver
```

TELUSKO

➢ Adding Required Dependencies

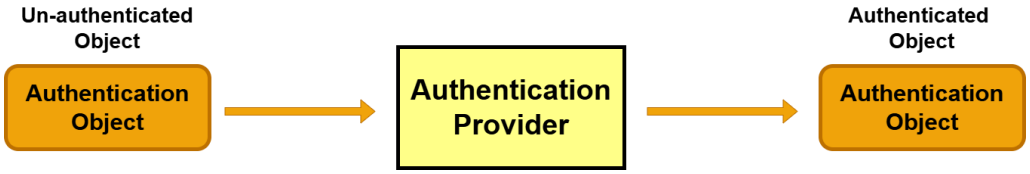
To connect our Java application to the database using JPA, we need to add these dependencies to our pom.xml file:

```
# JPA dependency
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

# PostgreSQL dependency
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

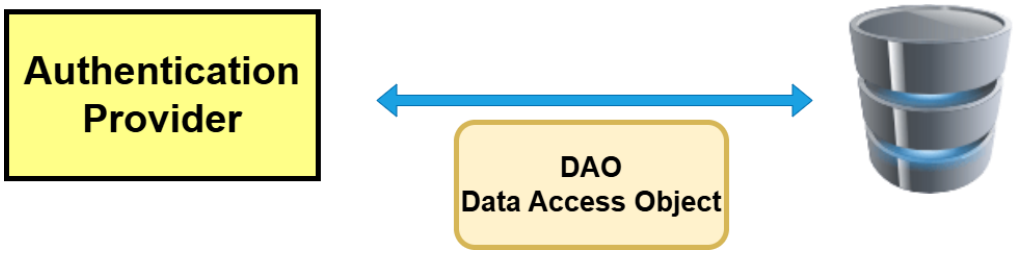
➢ Configuring Spring Security to Use Database Authentication

By default, Spring Security uses hardcoded values for authentication, but we want to use our database instead. We need to:

- Change the default Authentication Provider By default, Spring Security uses a built-in Authentication Provider that works with hardcoded values



- Switch to a database-aware Authentication Provider We need to configure a new Authentication Provider that connects to our database



- Create a Users entity class Finally, we need to specify how our application maps to the database table using a Users entity class. This class will represent our users table in Java code.

TELUSKO

- This setup allows our Spring Security to authenticate users against the database instead of using hardcoded credentials. When a user attempts to log in, Spring Security will:
    - Take the provided username and password
    - Use the Authentication Provider to check these credentials against our database
    - Grant or deny access based on whether the credentials match

TELUSKO

AuthenticationProvider

When implementing Spring Security, the framework automatically provides a default Authentication Provider behind the scenes. However, for database authentication, we need to create our own custom Authentication Provider.

👉 Creating a Custom Authentication Provider

To connect our Spring Security with a database, we need to:

- Create a bean inside SecurityConfig class that returns an AuthenticationProvider
- Use DaoAuthenticationProvider which implements the AuthenticationProvider interface
- Configure this provider to work with our database
- Understanding the Class Hierarchy
  - AuthenticationProvider is an interface
  - DaoAuthenticationProvider is an implementation class for AuthenticationProvider interface
  - DaoAuthenticationProvider extends AbstractUserDetailsAuthenticationProvider
  - AbstractUserDetailsAuthenticationProvider implements AuthenticationProvider

👉 DaoAuthenticationProvider

Creating a DaoAuthenticationProvider object alone is not enough. It needs to know:

- Which database we're working with
- How to represent user data
- What the user table name is

👉 UserDetailsService

To provide this information, we need a UserDetailsService:

- The default UserDetailsService works with static values
- We need a custom implementation to work with database values
- We connect it to our provider using setUserDetailsService()

👉 PasswordEncoder

We also need to specify a password encoder:

- Use setPasswordEncoder() method

TELUSKO

- For simple testing, we can use NoOpPasswordEncoder.getInstance()
- In production, you should use a secure encoder like BCrypt

Example:

```java
package com.telusko.springsecdemo.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.NoOpPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public AuthenticationProvider authProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(NoOpPasswordEncoder.getInstance());
        return provider;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(customizer -> customizer.disable())
            .authorizeHttpRequests(request -> request.anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

        return http.build();
    }

    // The commented code below shows the old in-memory authentication approach
    // that we're replacing with database authentication
    /*
     * @Bean public UserDetailsService userDetailsService() {
     *
     * UserDetails user=User
     *     .withDefaultPasswordEncoder()
     *     .username("navin")
     *     .password("n@123")
     *     .roles("USER")
     *     .build();
     *
     * UserDetails admin=User
     *     .withDefaultPasswordEncoder()
     *     .username("admin")
     *     .password("admin@789")
     *     .roles("ADMIN")
     *     .build();
     *
     * return new InMemoryUserDetailsManager(user,admin);
     * }
     */
}
```

TELUSKO

- The @Autowired private UserDetailsService userDetailsService; needs to be implemented
- We need to create a custom class that implements UserDetailsService to connect to our database
- The Authentication Provider depends on UserDetailsService to know how to authenticate users
- For production use, always use a secure password encoder (not NoOpPasswordEncoder)

➢ **After setting up this configuration, you'll need to**

- Create a class that implements UserDetailsService
- Define how to fetch user details from your database
- Map database user records to Spring Security's UserDetails objects

TELUSKO

Creating A UserDetailsService

UserDetailsService is a core interface in Spring Security that loads user-specific data for authentication. When we want to authenticate users against a database instead of hardcoded values, we need to create our own implementation of this interface.

Create a class called MyUserDetailsService in the service layer that implements the UserDetailsService interface. This class will be responsible for fetching user information from our database when a user attempts to log in.

Example:

```java
package com.telusko.springsecdemo.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.stereotype.Service;

@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    private UserRepo repo;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException
{
        return null; // Will be implemented to fetch user from database
    }
}
```

- **The @Service Annotation**
  Add the @Service annotation to class so Spring can detect it as a bean during component scanning. This allows it to be automatically injected into our AuthenticationProvider in the SecurityConfig class.

- **UserRepo Dependency**
  To access database, autowire a UserRepo interface. This repository will handle all database operations needed to fetch user information.

- **The loadUserByUsername Method**
  The most important part of this service is the loadUserByUsername() method:

  - It takes a username as input
  - It's called by Spring Security during authentication
  - It should query the database for the user with the matching username
  - It must return a UserDetails object containing the user's credentials and authorities
  - If no user is found, it should throw a UsernameNotFoundException

TELUSKO

User Repository

➢ Creating the User Repository

When working with JPA in Spring, we create a repository interface to interact with our database:

- Create an interface called UserRepo in the DAO (Data Access Object) layer
- It extends JpaRepository which provides built-in database operations
- The first type parameter (User) is the entity class that maps to our database table
- The second type parameter (Integer) is the type of our primary key
- Add the @Repository annotation to mark it as a Spring repository component

Example:

```
package com.telusko.springsecdemo.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepo extends JpaRepository<User, Integer> {

}
```

➢ Creating the User Entity Class

For our repository to work, we need a User entity class that maps to our database table:

- Create this class in the model layer
- It has the same properties as our database table (id, username, password)
- @Entity marks this class as a JPA entity
- @Table(name = "users") specifies which database table this class maps to
- @Id marks the primary key field
- @Data is a Lombok annotation that automatically generates getters, setters, equals, hashCode, and toString methods

Example:

```java
package com.telusko.springsecdemo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Table(name = "users")
@Entity
public class User {

    @Id
    private int id;
    private String username;
    private String password;

}
```

➢ **Adding a Custom Query Method**

To find a user by username, we add a custom query method to our repository:

- findByUsername is a custom method we define
- Spring Data JPA automatically implements this method based on the naming convention
- It will generate a query like SELECT * FROM users WHERE username = ?
- The method returns a single User object (or null if not found)

```java
package com.telusko.springsecdemo.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.telusko.springsecdemo.model.User;

@Repository
public interface UserRepo extends JpaRepository<User, Integer> {
    User findByUsername(String username);
}
```

➢ **Using the Repository in UserDetailsService**

Now we can use our repository in the loadUserByUsername method:

- Use the repository to find a user by their username
- Check if the user exists in the database
- If the user doesn't exist, we throw a UsernameNotFoundException

Example:

```java
package com.telusko.springsecdemo.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.stereotype.Service;
import com.telusko.springsecdemo.dao.UserRepo;
import com.telusko.springsecdemo.model.User;
import com.telusko.springsecdemo.model.UserPrincipal;

@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    private UserRepo repo;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = repo.findByUsername(username);

        if (user == null) {
            System.out.println("User 404");
            throw new UsernameNotFoundException("User 404");
        }

        return null
    }
}
```

TELUSKO

UserDetails And UserPrincipal

In Spring Security, UserDetails is an interface that provides core user information. Spring Security needs this information to authenticate users and manage their access to resources. When we implement database authentication, we need to create a class that implements UserDetails to bridge our database User model with Spring Security's authentication system.

- **Creating UserPrincipal Class**
  Create a class called UserPrincipal in the model package that implements the UserDetails interface:

Example:

```
package com.telusko.springsecdemo.model;

import java.util.Collection;
import java.util.Collections;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

public class UserPrincipal implements UserDetails {

    private static final long serialVersionUID = 1L;

    private User user;

    public UserPrincipal(User user) {
        this.user = user;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Collections.singleton(new SimpleGrantedAuthority("USER"));
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

    @Override
    public String getUsername() {
        return user.getUsername();
    }

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

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

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

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

TELUSKO

1. **Constructor**

```
public UserPrincipal(User user) {
    this.user = user;
}
```

- Takes a User object (our database entity) and stores it
- This allows us to access the user's information from our database

2. **getAuthorities()**

```
@Override public Collection<? extends GrantedAuthority> getAuthorities() {
    return Collections.singleton(new SimpleGrantedAuthority("USER"));
}
```

- Returns the roles/permissions assigned to the user
- In this example, we're assigning a simple "USER" role to all users
- Collections.singleton() creates a collection with just one element
- For multiple roles, we would return a list of authorities

3. **getPassword()**

```
@Override public String getPassword() {
  return user.getPassword();
}
```

- Returns the user's password from our database entity
- Spring Security uses this to verify the password during authentication

4. **getUsername()**

```
@Override public String getUsername() {
    return user.getUsername();
}
```

- Returns the user's username from our database entity
- Used as the identifier for the user in Spring Security

5. **Account Status Methods**

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

TELUSKO

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

- These methods control various aspects of account status
- Returning true means the account is valid in all aspects
- In a more complex application, these might check database flags for account status

➢ **Using UserPrincipal in UserDetailsService**

In MyUserDetailsService, use the UserPrincipal class to wrap our database user:

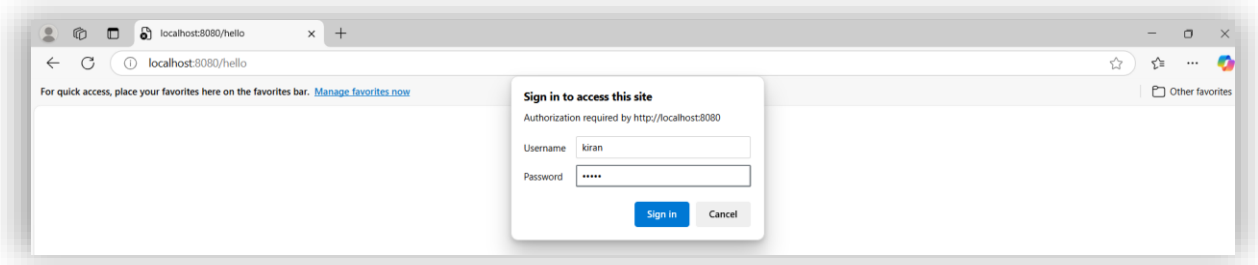
Example:

```
package com.telusko.springsecdemo.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.stereotype.Service;
import com.telusko.springsecdemo.dao.UserRepo;
import com.telusko.springsecdemo.model.User;
import com.telusko.springsecdemo.model.UserPrincipal;

@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    private UserRepo repo;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = repo.findByUsername(username);

        if (user == null) {
            System.out.println("User 404");
            throw new UsernameNotFoundException("User 404");
        }

        return new UserPrincipal(user);
    }
}
```

- We fetch the User from the database using the repository
- If the user doesn't exist, we throw a UsernameNotFoundException
- If the user exists, we create a new UserPrincipal with the user data
- We return this UserPrincipal object, which Spring Security will use for authentication

- The loadUserByUsername() method is called during authentication
- It retrieves our database User entity
- We wrap this entity in a UserPrincipal object
- Spring Security uses the information from UserPrincipal to:
  - Verify the password
  - Check account status
  - Determine user authorities/roles

Output:



TELUSKO

Summary Till Now

👉 Spring Security with Database Authentication: Complete Summary

➢ The Security Configuration

The heart of our Spring Security setup is the SecurityConfig class:

Example:

```java
package com.telusko.springsecdemo.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.NoOpPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public AuthenticationProvider authProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(NoOpPasswordEncoder.getInstance());
        return provider;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(customizer -> customizer.disable())
            .authorizeHttpRequests(request -> request.anyRequest().authenticated())
            .httpBasic(Customizer.withDefaults())
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

        return http.build();
    }
}
```

- We use @Configuration and @EnableWebSecurity to mark this as a Spring Security configuration
- We create a custom AuthenticationProvider bean that uses database authentication
- We configure security settings like CSRF protection, authentication requirements, and session management

TELUSKO

➢ **The Custom UserDetailsService**

To connect with our database, we implement the UserDetailsService interface:

Example:

```java
package com.telusko.springsecdemo.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.stereotype.Service;

import com.telusko.springsecdemo.dao.UserRepo;
import com.telusko.springsecdemo.model.User;
import com.telusko.springsecdemo.model.UserPrincipal;

@Service
public class MyUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepo repo;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = repo.findByUsername(username);

        if (user == null) {
            System.out.println("User 404");
            throw new UsernameNotFoundException("User 404");
        }
        return new UserPrincipal(user);
    }
}
```

- The @Service annotation makes this a Spring-managed bean that can be autowired
- We inject the UserRepo to access our database
- The loadUserByUsername() method fetches a user by username from the database
- If the user exists, we wrap it in a UserPrincipal object and return it
- If the user doesn't exist, we throw a UsernameNotFoundException

TELUSKO

➢ **The Repository Layer**

We create a UserRepo interface to interact with our database:

Example:

```java
package com.telusko.springsecdemo.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.telusko.springsecdemo.model.User;

@Repository
public interface UserRepo extends JpaRepository<User, Integer> {
    User findByUsername(String username);
}
```

- It extends JpaRepository to get basic CRUD operations for free
- We add a custom findByUsername() method to search by username
- Spring Data JPA automatically implements this method based on the name

➢ **The User Entity**

Our User class maps to the database table:

Example:

```java
package com.telusko.springsecdemo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Table(name = "users")
@Entity
public class User {

    @Id
    private int id;
    private String username;
    private String password;
}
```

TELUSKO

- @Entity and @Table annotations mark this as a JPA entity mapping to the "users" table
- It has fields corresponding to the database columns: id, username, and password
- @Data from Lombok generates getters, setters, equals, hashCode, and toString methods

➢ **The UserPrincipal Class**

This class adapts our database User to Spring Security's UserDetails interface:

Example:

```java
package com.telusko.springsecdemo.model;

import java.util.Collection;
import java.util.Collections;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

public class UserPrincipal implements UserDetails {

    private static final long serialVersionUID = 1L;

    private User user;

    public UserPrincipal(User user) {
        this.user = user;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Collections.singleton(new SimpleGrantedAuthority("USER"));
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

    @Override
    public String getUsername() {
        return user.getUsername();
    }

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

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

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

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

TELUSKO

- It implements the UserDetails interface required by Spring Security
- It wraps our database User object to access username and password
- The getAuthorities() method returns the user's roles/permissions (here, a simple "USER" role)
- The account status methods all return true, indicating the account is valid

➢ **Database Configuration**

In application.properties, we configure the database connection:

Example:

```
# PostgreSQL Database Configuration
spring.datasource.url=jdbc:postgresql://localhost:5432/telusko
spring.datasource.username=postgres
spring.datasource.password=0000
spring.datasource.driver-class-name=org.postgresql.Driver

server.servlet.session.cookie.same-site=strict
```

- These properties tell Spring how to connect to our PostgreSQL database
- The cookie setting adds extra security for the session cookie

👉 **Explanation**

➢ When a user tries to access a protected resource, Spring Security intercepts the request

➢ Spring Security calls our custom AuthenticationProvider

➢ The provider uses MyUserDetailsService to load the user from the database

➢ MyUserDetailsService uses UserRepo to fetch the user and wraps it in a UserPrincipal

➢ The provider compares the submitted password with the one from the database

➢ If they match, authentication succeeds and the user can access the resource

➢ If they don't match, or the user doesn't exist, authentication fails

TELUSKO

What is Bcrypt?

👉 The Problem with Plain Text Passwords

When we store passwords in a database or transfer them over networks in plain text, it creates a security risk. Anyone who gains access to the database or intercepts the transfer can immediately see all passwords.

👉 Encryption (Two-way)\

- Encryption is like using a lock and key for your data
- You encrypt (lock) a password with a key when storing it
- You decrypt (unlock) it with the same key when verifying
- **Problem**: If someone steals the encryption key, they can decrypt all passwords

👉 Hashing (One-way)

- Hashing converts data into a fixed-length string of characters
- **Key difference**: Hashing is one-way - once hashed, you cannot convert back to the original
- To verify passwords: you hash what the user enters and compare the hashes
- Popular hashing algorithms include MD5 and SHA256, but they have vulnerabilities

TELUSKO

➢ The Secure Solution

- **Bcrypt** is a specialized password-hashing function designed for security
- It's intentionally slow to prevent brute-force attacks
- It automatically incorporates "salt" to protect against rainbow table attacks

How Bcrypt Works

- When a user creates a password, Bcrypt hashes it
- The hashed result is stored in the database
- When the user tries to log in, Bcrypt hashes their entered password
- The system compares this hash with the stored hash
- If they match, the password is correct

Bcrypt Example

You can see Bcrypt in action at https://www.browsrling.com

TELUSKO

- A Bcrypt hash looks like this:

  $2a$10$8i.dYW9Uk/g88KVLSTkmf.fWuYBdPsTNB.QEUxy9q3ArGUJvaKI4m

- $2a indicates the Bcrypt version

- $10 indicates the "cost factor" (number of rounds)
  - This means 2^10 (1,024) iterations, not just 10
  - More rounds = more secure but slower processing

- The rest is the salt and the hashed password combined

👉 **Key features of Bcrypt**

- Slow by design to prevent brute-force attacks

- Built-in salt protection

- Adjustable work factor to adapt to faster computers over time

- Industry standard for password security

TELUSKO

User Registration

To implement secure user registration with password encoding, we need to create a proper controller-service architecture. This will handle the HTTP requests and save user data to the database.

👉 Creating the UserController

- Create a controller class called UserController
- Annotate it with @RestController to handle HTTP requests
- Define a registration endpoint with @PostMapping("register")
- Accept user data in JSON format using @RequestBody
- Inject the UserService into the UserController using @Autowired
- Call the service method from the controller to process the registration

Example:

```
package com.telusko.springsecdemo.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.telusko.springsecdemo.model.User;
import com.telusko.springsecdemo.service.UserService;

@RestController
public class UserController {
    @Autowired
    private UserService service;

    @PostMapping("register")
    public User register(@RequestBody User user) {
      return service.saveUser(user);
    }
}
```

TELUSKO

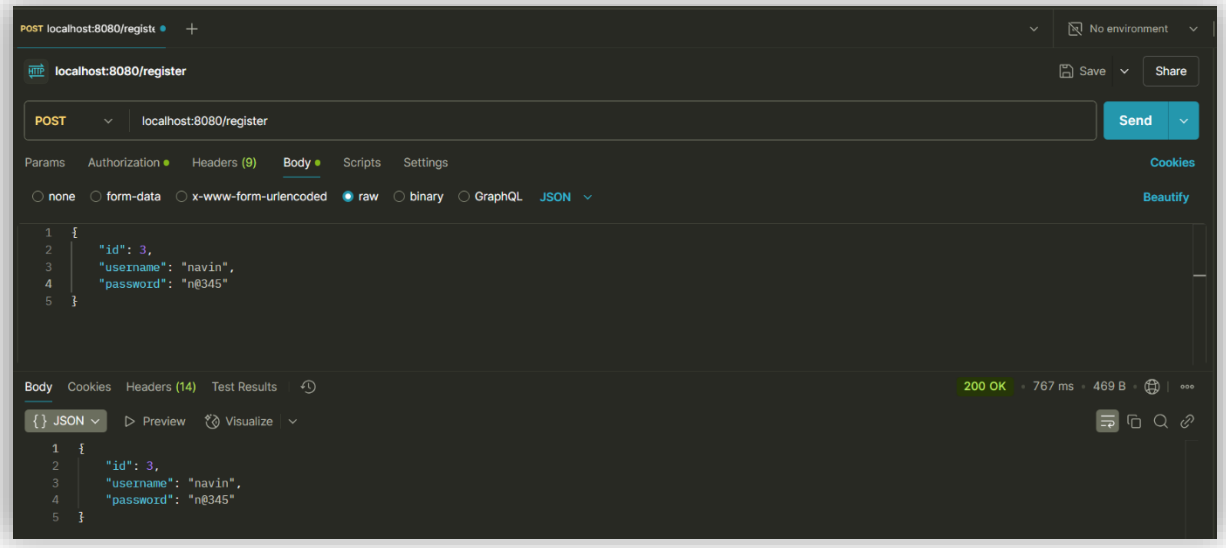
👉 Setting Up the UserService

- Create a service class called UserService
- Annotate it with @Service to mark it as a Spring service component
- Implement a saveUser() method that takes a User object and returns the saved User
- Inject the UserRepo repository to interact with the database

Example:

```java
package com.telusko.springsecdemo.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.telusko.springsecdemo.dao.UserRepo;
import com.telusko.springsecdemo.model.User;

@Service
public class UserService {
    @Autowired
    private UserRepo repo;

    public User saveUser(User user) {
        return repo.save(user);
    }
}
```

Output:



TELUSKO

Query | Query History

```
1  SELECT * FROM users
2
```

Data Output | Messages | Notifications

	id [PK] integer	username text	password text
1	1	kiran	n@789
2	2	harsh	h@123
3	3	navin	n@345

TELUSKO

Bcrypt Encoding For User Registration

➢ **When we implement Bcrypt in an existing application, we face a challenge:**

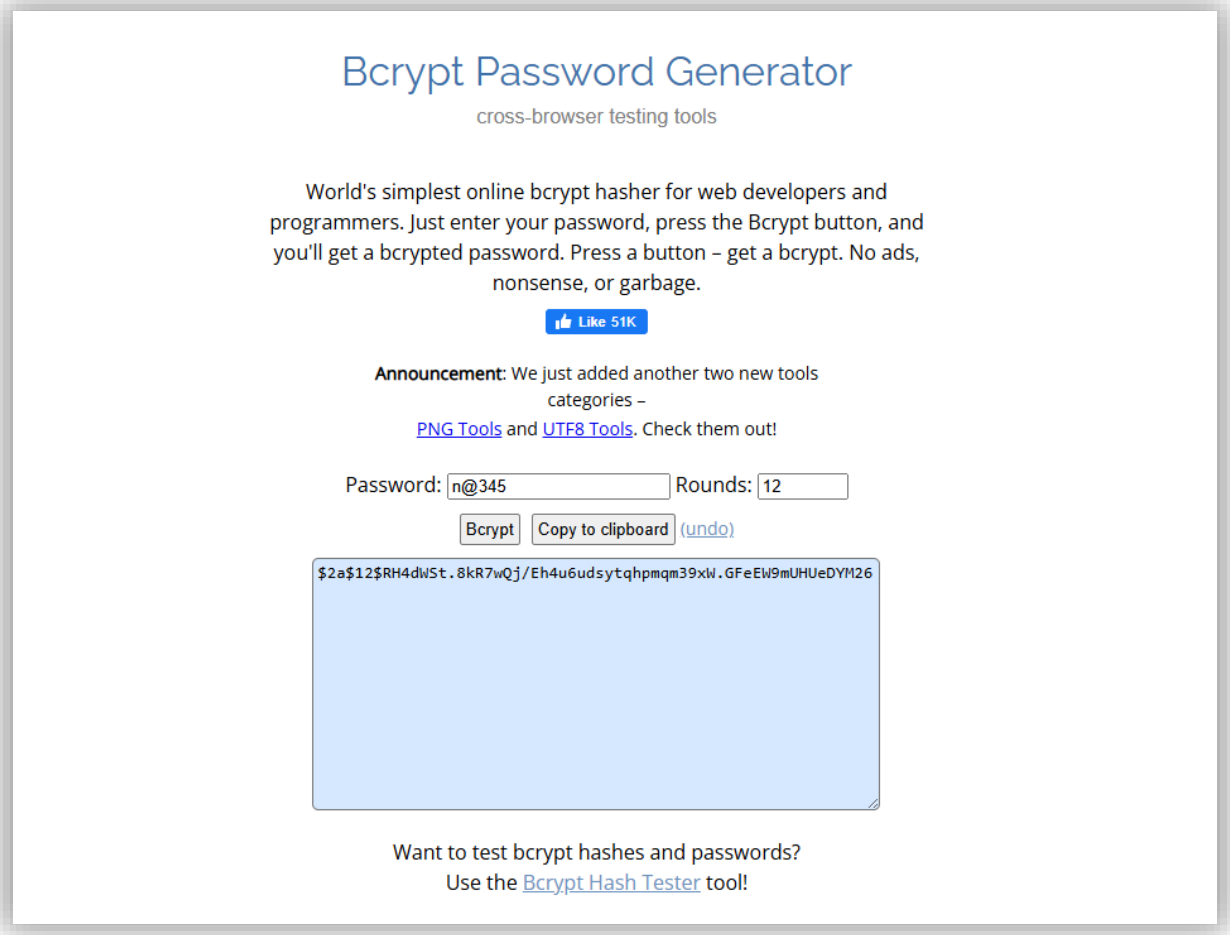
- Existing users already have plain text passwords in the database
- Our authentication system expects plain text passwords
- Once Bcrypt is implemented, the system will try to compare Bcrypt hashes with plain text

➢ **Update Existing Password Data**

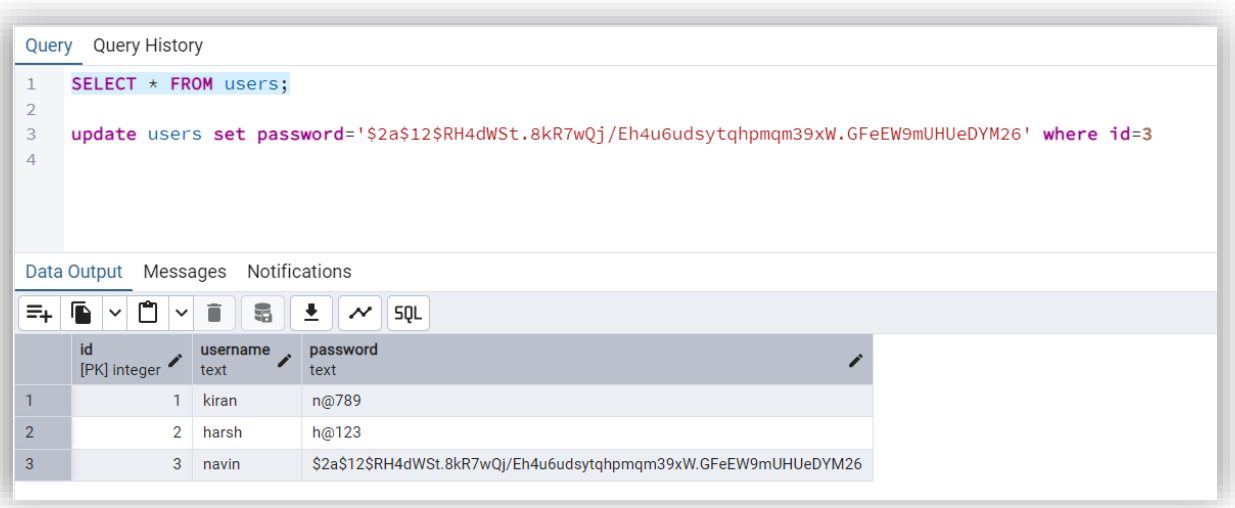
Before implementing Bcrypt, we need to:

- Update existing passwords in the database
- Convert plain text passwords to Bcrypt-encoded versions
- This can be done using an update query after generating encoded passwords

You can generate encoded passwords using an online tool like https://www.browsrling.com:



TELUSKO



➢ **Implementing Bcrypt in the Service Layer**

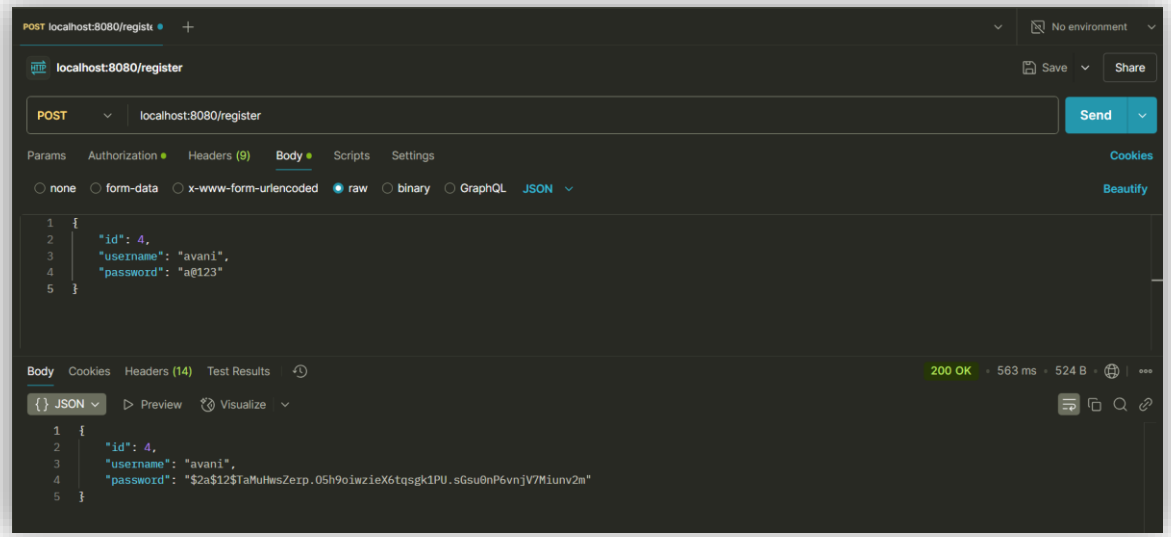
- In the UserService class, create a BCryptPasswordEncoder object
- Set the strength parameter (rounds) to 12 for good security
- Before saving the user, encode the password using the encoder
- Save the user with the encoded password to the database

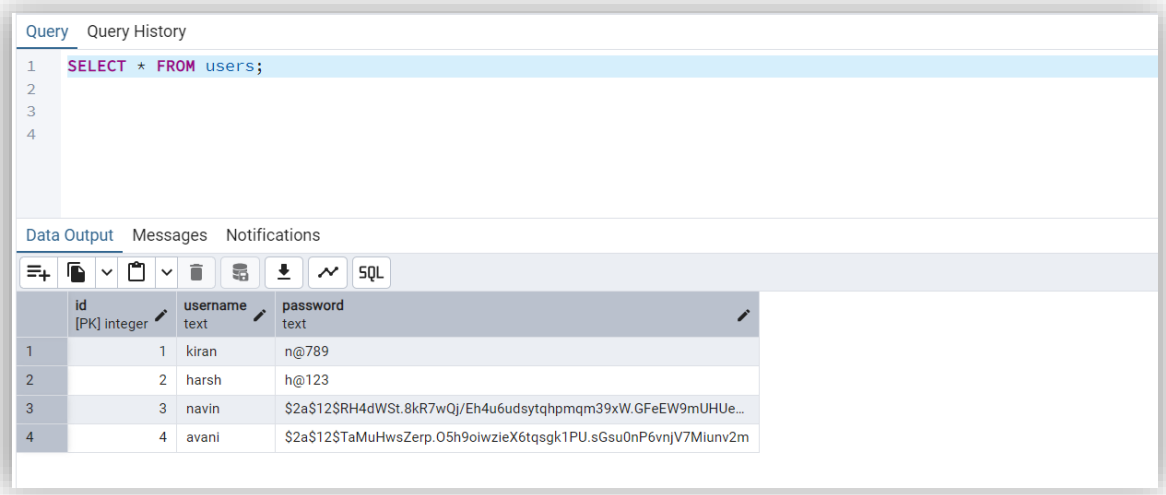
Example:

```
package com.telusko.springsecdemo.service;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;
import com.telusko.springsecdemo.dao.UserRepo;
import com.telusko.springsecdemo.model.User;

@Service
public class UserService {

    @Autowired
    private UserRepo repo;
    private BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(12);

    public User saveUser(User user) {
        user.setPassword(encoder.encode(user.getPassword()));
        System.out.println(user.getPassword());
        return repo.save(user);
    }
}
```

TELUSKO

Output:





👉 Remember

- BCryptPasswordEncoder is part of Spring Security framework
- The parameter 12 represents the work factor (2^12 = 4,096 rounds)
- Higher work factor means better security but slower performance
- encode() method transforms the plain text password into a Bcrypt hash
- We're printing the encoded password to console for demonstration (not recommended in production)
- After this implementation, all new users will have encoded passwords
- The authentication system must also be updated to verify with Bcrypt (covered in next section)

TELUSKO

Setting Password Encoder

After implementing password encoding during registration, we need to update our authentication system to verify encoded passwords. The key points:

- Previously, we were using NoOpPasswordEncoder which expects plain text passwords
- Now that our passwords are encoded with BCrypt, we need to use BCryptPasswordEncoder for authentication
- This ensures the system can properly verify passwords during login

👉 Updating the Security Configuration

In our SecurityConfig class, we need to make the following changes:

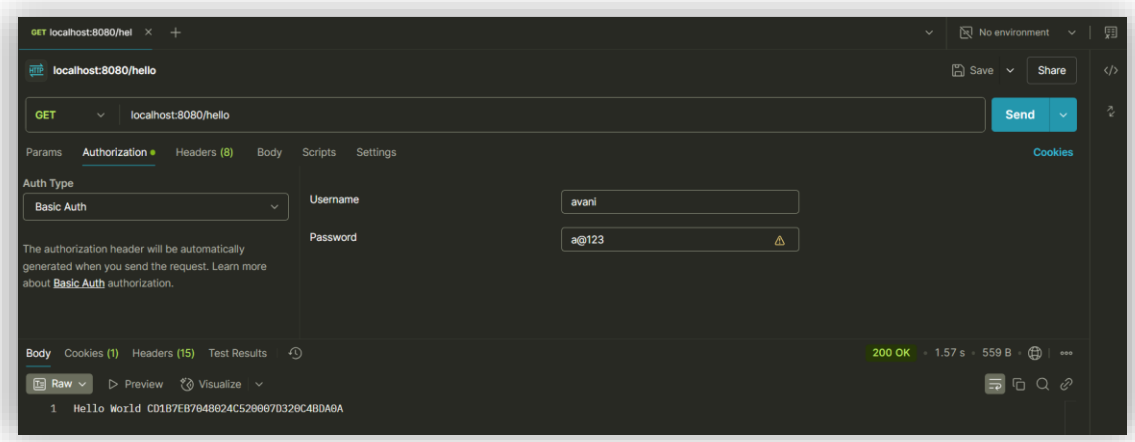
- Keep the @Configuration and @EnableWebSecurity annotations that mark this as a Spring Security configuration
- The AuthenticationProvider bean must now use BCryptPasswordEncoder instead of NoOpPasswordEncoder
- Set the same strength parameter (12) as we used in the registration process

Example:

```java
package com.telusko.springsecdemo.config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public AuthenticationProvider authProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(new BCryptPasswordEncoder(12));
        return provider;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(customizer -> customizer.disable())
                .authorizeHttpRequests(request -> request.anyRequest().authenticated())
                .httpBasic(Customizer.withDefaults())
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
        return http.build();
    }
}
```

TELUSKO

Output:



- User provides credentials (username and password)
- Spring Security retrieves the user details using UserDetailsService
- The BCryptPasswordEncoder takes the plain text password from the login attempt and applies the same hashing algorithm
- It then compares this hash with the stored hash from the database
- If they match, authentication succeeds

TELUSKO

Setting Password Encoder

After implementing password encoding during registration, we need to update our authentication system to verify encoded passwords. The key points:

- Previously, we were using NoOpPasswordEncoder which expects plain text passwords
- Now that our passwords are encoded with BCrypt, we need to use BCryptPasswordEncoder for authentication
- This ensures the system can properly verify passwords during login

👉 Updating the Security Configuration

In our SecurityConfig class, we need to make the following changes:

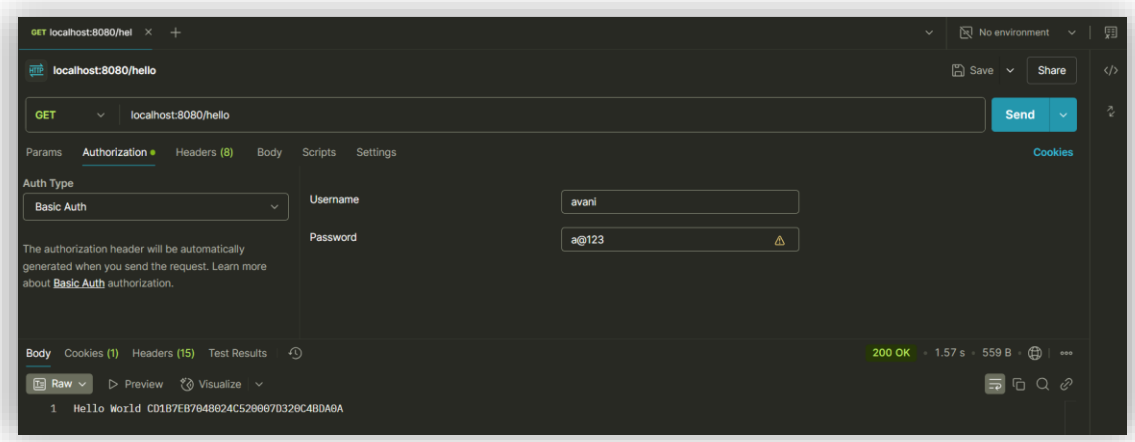
- Keep the @Configuration and @EnableWebSecurity annotations that mark this as a Spring Security configuration
- The AuthenticationProvider bean must now use BCryptPasswordEncoder instead of NoOpPasswordEncoder
- Set the same strength parameter (12) as we used in the registration process

Example:

```java
package com.telusko.springsecdemo.config;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public AuthenticationProvider authProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(new BCryptPasswordEncoder(12));
        return provider;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(customizer -> customizer.disable())
                .authorizeHttpRequests(request -> request.anyRequest().authenticated())
                .httpBasic(Customizer.withDefaults())
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
        return http.build();
    }
}
```

TELUSKO

Output:



- User provides credentials (username and password)
- Spring Security retrieves the user details using UserDetailsService
- The BCryptPasswordEncoder takes the plain text password from the login attempt and applies the same hashing algorithm
- It then compares this hash with the stored hash from the database
- If they match, authentication succeeds

TELUSKO