Secure an Endpoint with Spring Security

Estimated time needed: 20 minutes

Overview

In this lab, you will use `Spring Initializr` to generate a project to create a Library Spring Boot web application with REST API endpoints some of which require to be authenticated. You will use the required dependencies to enable web services and security services. You will use java Collections to maintain a database of the users.

Learning objectives

After completing this lab, you will be able to:

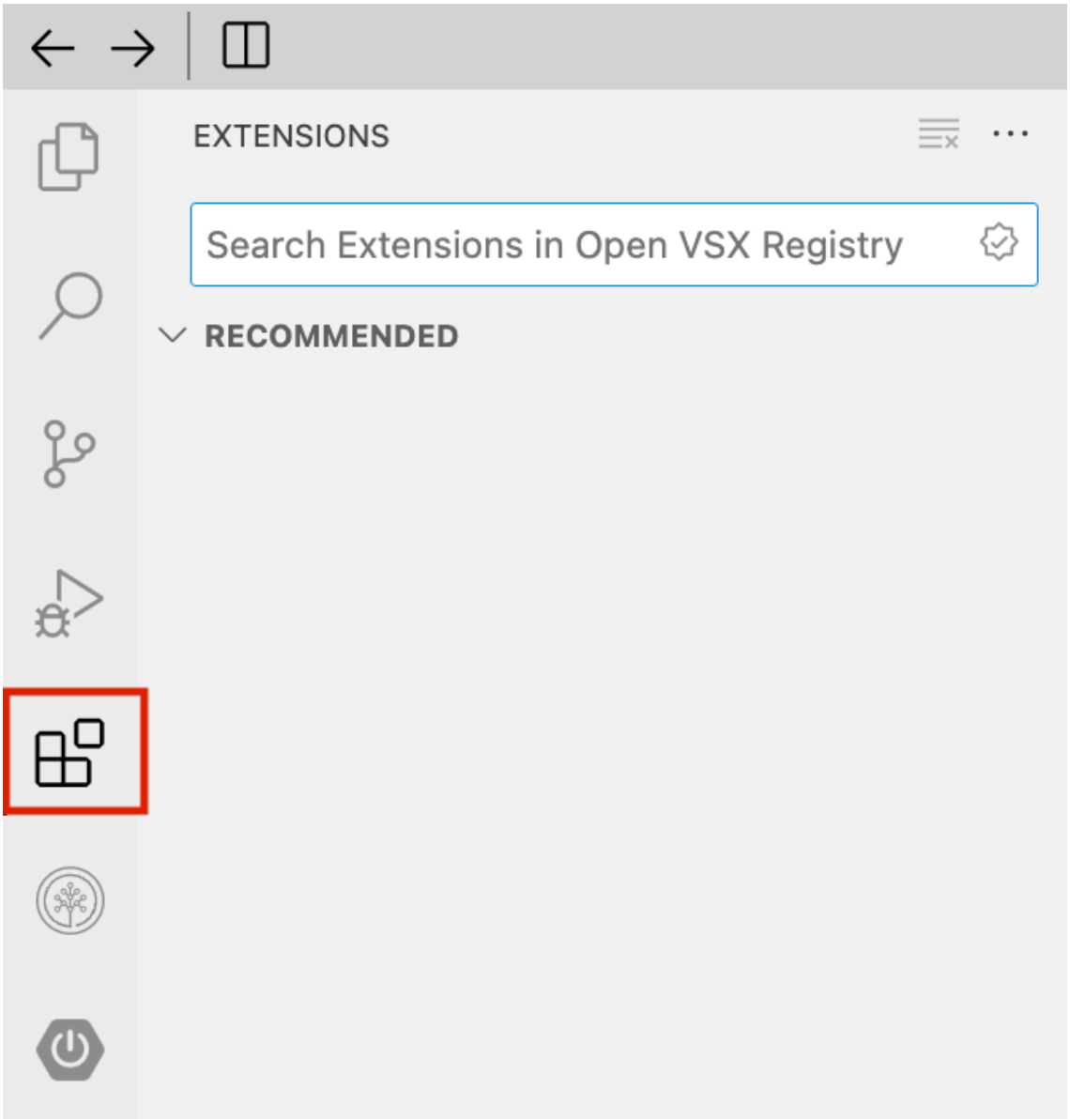
- Use Spring Initializr to generate the project
- Set up a Spring boot project structure
- Create the required model with getters and setters
- Create a controller which uses the model and handles http requests for user registration and login
- Build and run the spring boot application
- Test the endpoints on the browser

Prerequisites (optional)

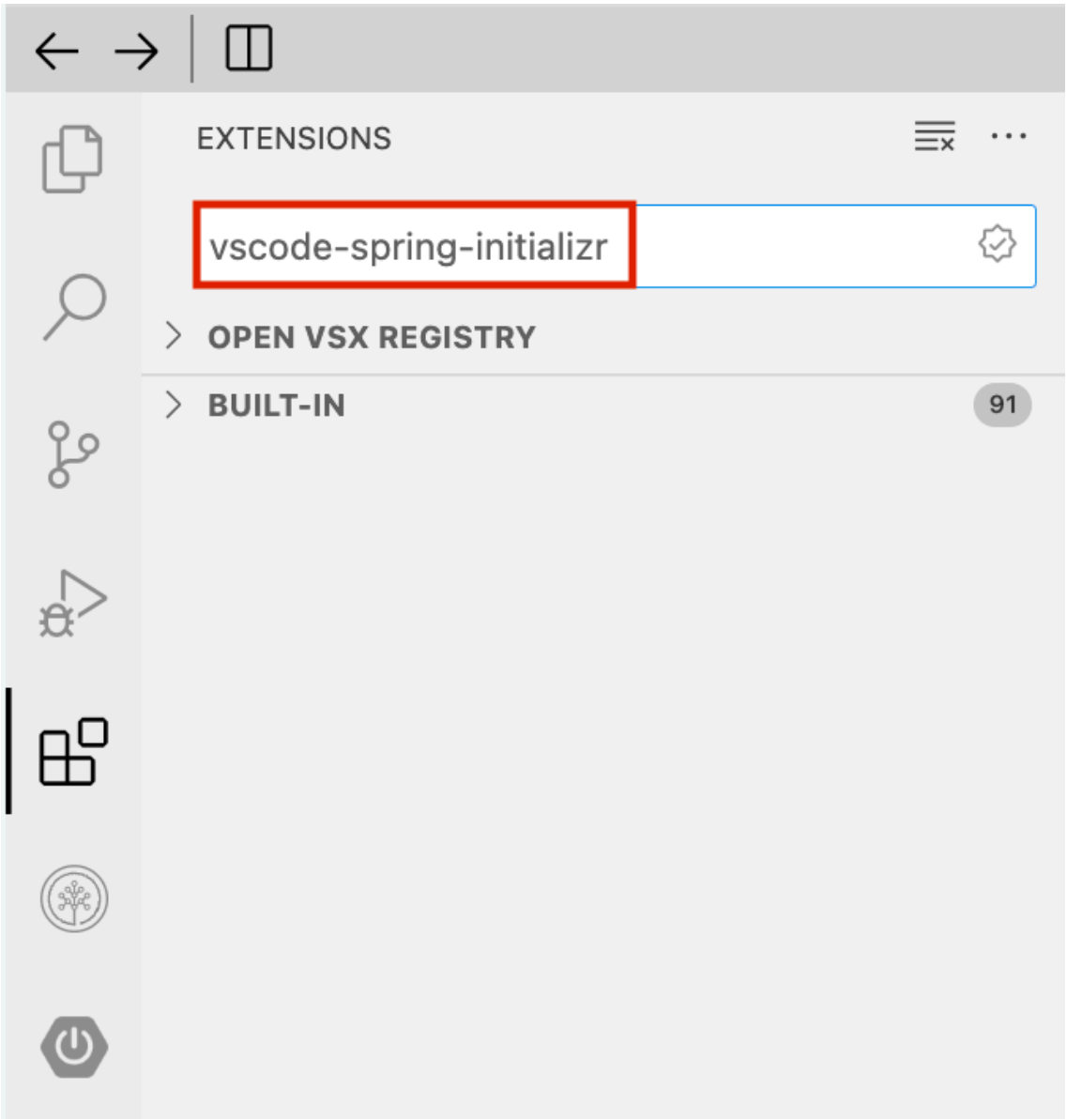
You should know basic Java programming before you get started with this lab. Please ensure that you complete the labs sequentially as they are constructed progressively.

Set up Cloud IDE for Maven Project

1. Click the extensions icon to start installing the extensions.



2. In the search bar for the extensions, type `vscode-spring-initializr`.

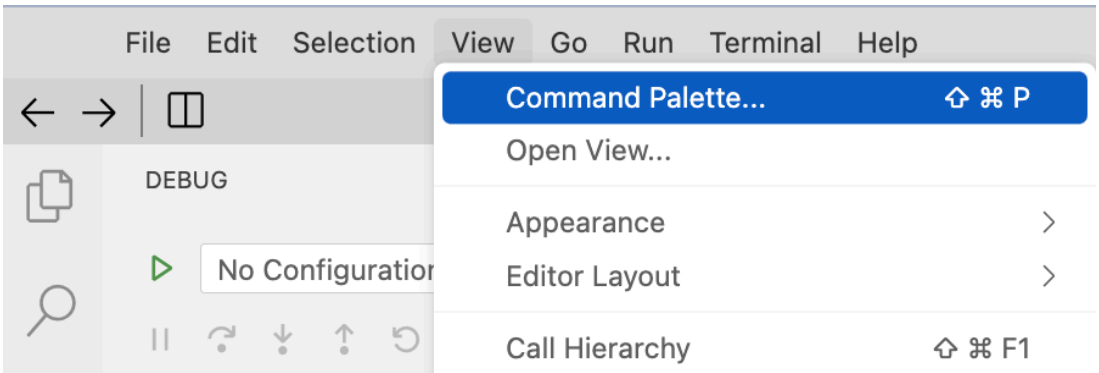


This will list all the extensions you will need for Spring Initializr.

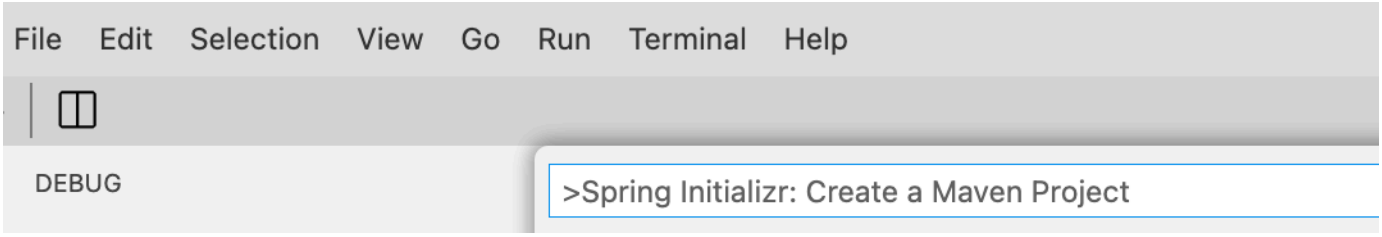
3. Install the `Spring Initializr Java Support` extension.

Create Maven Project

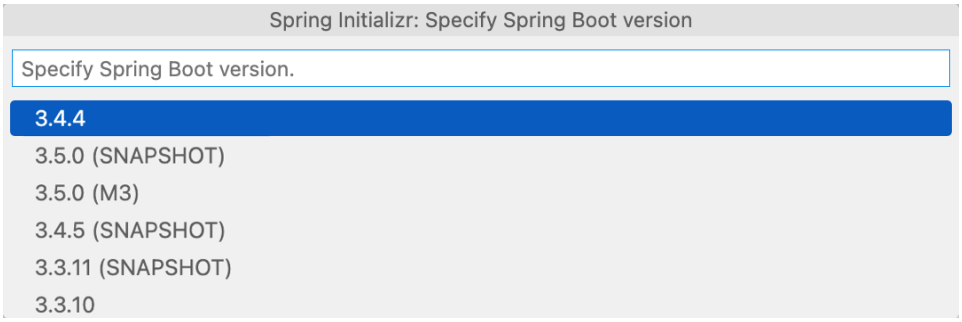
1. Open command palette.



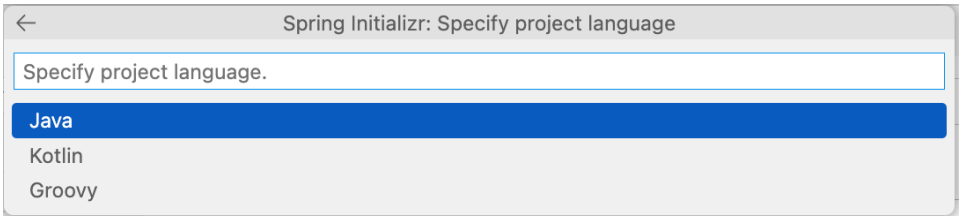
2. In the command paletter text entry box, type `Spring Initializr: Create a Maven Project`.



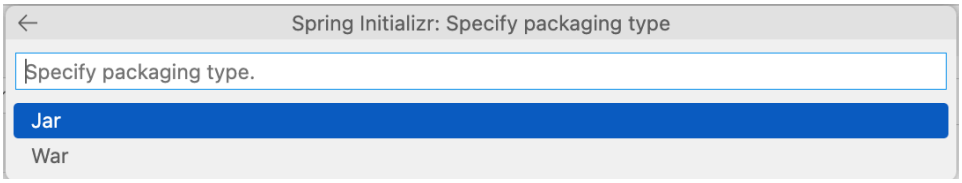
3. Specify the Spring Boot version `3.4.4`.



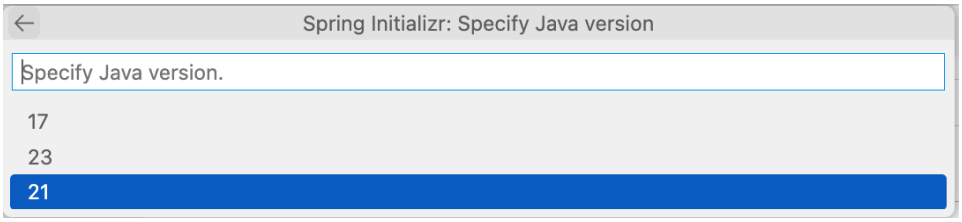
4. Select the language you want to use, in your case`Java`.



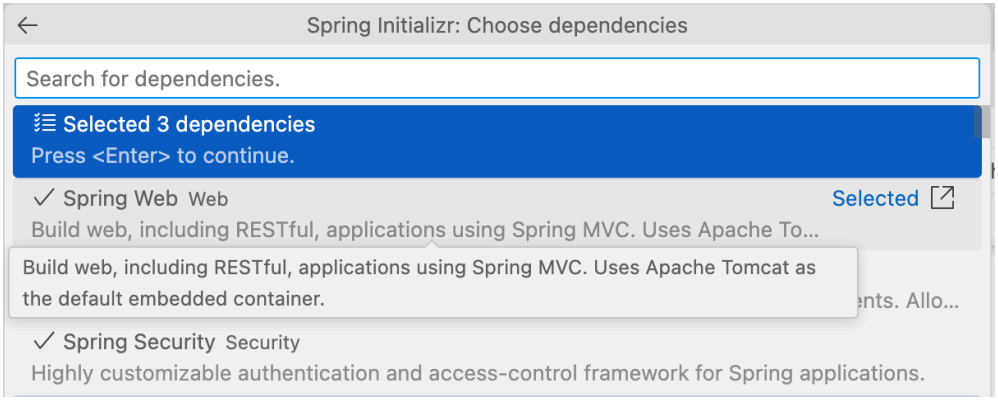
5. Specify the group id such as `com.example`.
6. Specify the artifact id id such as `secureapp`.
7. Specify the packaging you want to use as `Jar` (for Java archives).



8. Specify the Java version you want to use, `21` in the case of the IDE.



9. You will be using `Spring Web`, `Thymeleaf`, and `Spring Security` dependencies for the project. Press enter to continue.



10. The project is generated by default in the `/home/project` folder. Click `Generate` to generate the project in the desired folder.

11. Add the project to the workspace.

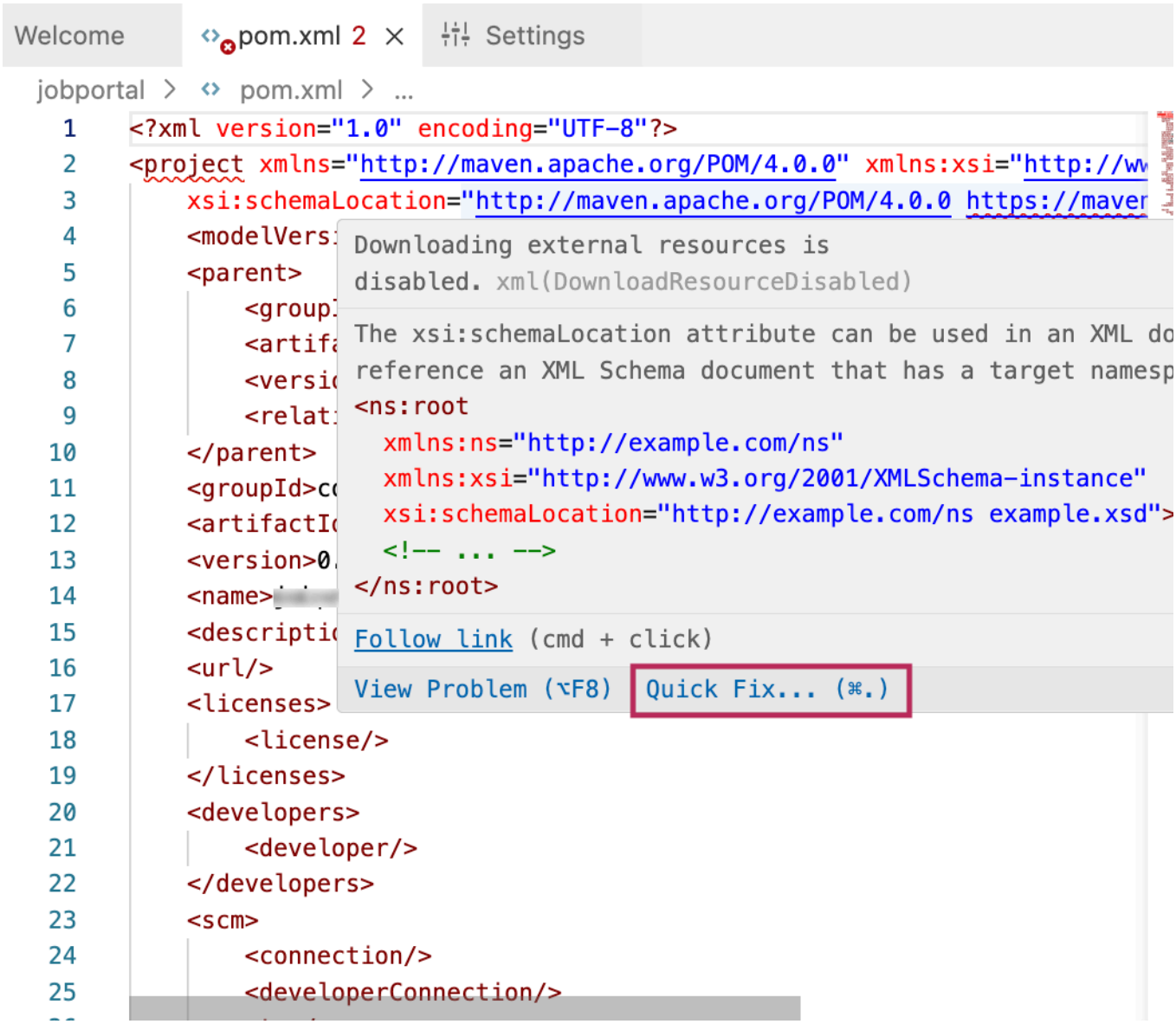
Configure project, create directories and create the User model

1. Open the `pom.xml` and ensure that the dependecies included and packaging is set `jar`.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.thymeleaf.extras</groupId>
        <artifactId>thymeleaf-extras-springsecurity6</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

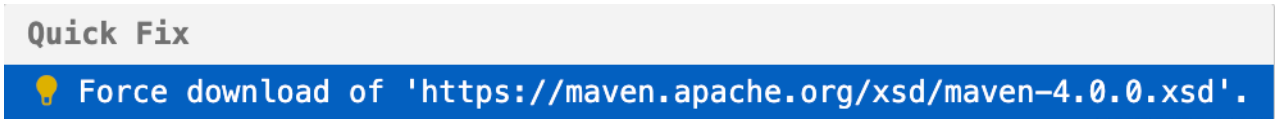
```
<groupId>com.example</groupId>
<artifactId>secureapp</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>secureapp</name>
```

2. Hover over the highlighted links on `pom.xml` and click. A suggestion comes up. Click on `Quick Fix`.



3. Click to download the `xsd` file.

   The xsd (XML Schema Definition) file in a pom.xml serves as a schema validator that defines the structure, syntax, and data types allowed in the Maven Project Object Model (POM) file.



4. Under `com/example/secureapp`, create the following folders.

- `config` to contain class that handles the web application security configuration
- `controller` to contain class that handles the secure and unauthenticated API endpoints
- `model` to contain the `User` class
- `service` to contain the service class that handles `UserDetailsService`

5. Create a new file named `User.java` in the `com/example/secureapp/model` folder.

6. Add the following code in `User.java` to create the `User` class.

```
package com.example.secureapp.model;
public class User {
    private String username; // Username of the user
    private String password; // Password of the user (encoded)
    // Constructor
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    // Getters and Setters
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
```

```
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Create the WebSecurityConfig class

1. Create `WebSecurityConfig.java` file inside `com/example/secureapp/config` folder.
2. Add the following code in `WebSecurityConfig.java` to create the `WebSecurityConfig` class. This class will be used for authenticating the endpoints.

```
package com.example.secureapp.config;
import com.example.secureapp.service.CustomUserDetailsService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {
    private final CustomUserDetailsService userDetailsService;
    public WebSecurityConfig(CustomUserDetailsService userDetailsService) {
        this.userDetailsService = userDetailsService;
    }
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/register", "/login").permitAll() // Allow access to registration and login pages
                .requestMatchers("/greet").authenticated() // Secure the /greet endpoint
                .anyRequest().permitAll() // Allow access to all other endpoints
            )
            .formLogin(form -> form
                .loginPage("/login") // Custom login page
                .defaultSuccessUrl("/greet", true) // Redirect to /greet after successful login
                .permitAll()
            )
            .logout(logout -> logout
                .permitAll()
            );
        return http.build();
    }
    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http) throws Exception {
        AuthenticationManagerBuilder authenticationManagerBuilder =
            http.getSharedObject(AuthenticationManagerBuilder.class);
        authenticationManagerBuilder
            .userDetailsService(userDetailsService) // Use your custom UserDetailsService
            .passwordEncoder(passwordEncoder()); // Use the password encoder
        return authenticationManagerBuilder.build();
    }
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

- `@Configuration`: Marks this class as a Spring configuration class. It tells Spring that this class contains bean definitions and configuration settings.
- `@EnableWebSecurity`: Enables Spring Security's web security support and provides the Spring MVC integration. It allows you to customize security settings for your application.
- `authorizeHttpRequests`: Configures URL-based authorization rules.
    - `.requestMatchers(\"/register\", \"/login\").permitAll()`: Allows unauthenticated access to the `/register` and `/login` endpoints.
    - `.requestMatchers(\"/greet\").authenticated()`: Requires authentication for the `/greet` endpoint. Only authenticated users can access it.
    - `.anyRequest().permitAll()`: Allows access to all other endpoints without authentication.

Create the CustomUserDetailsService class

1. Create `CustomUserDetailsService.java` file inside `com/example/secureapp/service` folder.

2. Add the following code in `CustomUserDetailsService.java` to create the `CustomUserDetailsService` class. This class, CustomUserDetailsService, is a custom implementation of Spring Security's UserDetailsService interface. It is responsible for loading user-specific data (e.g., username, password, roles) during the authentication process. Additionally, it provides a method to register new users by storing their credentials in an in-memory HashMap.

```
package com.example.secureapp.service;
import com.example.secureapp.model.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import java.util.HashMap;
import java.util.Map;
@Service
public class CustomUserDetailsService implements UserDetailsService {
    private final Map<String, User> users = new HashMap<>();
    private final PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = users.get(username);
        if (user == null) {
            throw new UsernameNotFoundException("User not found");
        }
        return org.springframework.security.core.userdetails.User.builder()
            .username(user.getUsername())
            .password(user.getPassword())
            .roles("USER")
            .build();
    }
    public void registerUser(String username, String password) throws Exception {
        if(users.containsKey(username)) {
            throw new Exception("User already exists");
        } else {
            String encodedPassword = passwordEncoder.encode(password);
            users.put(username, new User(username, encodedPassword));
        }
    }
}
```

- @Service: Marks this class as a Spring service bean.
- implements UserDetailsService: Implements the UserDetailsService interface, which is a core interface in Spring Security for loading user-specific data.
- `registerUser` is called when the user registers. The password is encoded using BCryptPasswordEncoder, and the user's credentials are stored in the HashMap.
- `loadUserByUsername` is called when a user attempts to log in. The method retrieves the user's details from the HashMap and returns a UserDetails object.

Create the GreetingController class

1. Create `GreetingController.java` file inside `com/example/secureapp/controller` folder.

2. Add the following code in `GreetingController.java` to create the `GreetingController` class. This class handles requests related to user authentication, registration, and displaying a greeting page.

```
package com.example.secureapp.controller;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import com.example.secureapp.service.CustomUserDetailsService;
@Controller
public class GreetingController {
    private final CustomUserDetailsService userDetailsService;
    private final AuthenticationManager authenticationManager;
    public GreetingController(CustomUserDetailsService userDetailsService, AuthenticationManager authenticationManager) {
        this.userDetailsService = userDetailsService;
        this.authenticationManager = authenticationManager;
    }
    @GetMapping("/greet")
    public String greet(Model model) {
        // Get the authenticated user's username
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
        String username = authentication.getName();
        System.out.println("Username from context "+username);
        // Add the username to the model
        model.addAttribute("username", username);
        // Return the Thymeleaf template name
        return "greet";
    }
    @GetMapping("/login")
    public String login() {
        return "login"; // Returns the login.html template
    }
    @GetMapping("/register")
    public String register() {
```

```
        return "register"; // Returns the register.html template
    }
    // POST endpoint to handle user registration and auto-login
    @PostMapping("/register")
    public String registerUser(
            @RequestParam String username, // Username from the form
            @RequestParam String password // Password from the form
    ) {
        // Register the user by storing their details in the HashMap
        try {
            userDetailsService.registerUser(username, password);
        } catch (Exception userExistsAlready) {
            // Redirect to the /register endpoint
            return "redirect:/register?error";
        }
        // Authenticate the user programmatically
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(username, password)
        );
        // Set the authentication in the SecurityContext
        SecurityContextHolder.getContext().setAuthentication(authentication);
        // Redirect to the /login endpoint
        return "redirect:/login?success";
    }
}
```

- The constructor takes CustomUserDetailsService and AuthenticationManager. CustomUserDetailsService implements Spring Security's UserDetailsService. It is used to load user details during authentication and register new users. AuthenticationManager component responsible for authenticating users. Both dependencies are injected into the controller via constructor injection.

These are the following API endpoints that the controller handles.

- `@GetMapping("/greet")` to show greetings page for logged in user
- `@GetMapping("/login")` to show the login page
- `@GetMapping("/register")` to show the register page
- `@PostMapping("/register")` to register the user

Create the HTML files in the template folder

1. Create `register.html` file inside `resources/templates` folder.
2. Add the following code in `register.html`.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Registration Page</title>
    <style>
        /* Center the body content */
        body {
            font-family: Arial, sans-serif;
            background-color: #f4f4f4;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            margin: 0;
        }
        /* Style the registration container */
        .registration-container {
            background-color: #fff;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
            width: 300px;
            text-align: center;
        }
        /* Style the form elements */
        .registration-container h2 {
            margin-bottom: 20px;
        }
        .registration-container input {
            width: 100%;
            padding: 10px;
            margin: 10px 0;
            border: 1px solid #ccc;
            border-radius: 4px;
        }
        .registration-container button {
            width: 100%;
            padding: 10px;
            background-color: #28a745;
            color: #fff;
            border: none;
            border-radius: 4px;
            cursor: pointer;
        }
        .registration-container button:hover {
```

```
            background-color: #218838;
        }
        /* Flexbox layout for form items */
        .form-group {
            display: flex;
            flex-direction: column;
            align-items: center;
        }
        .form-group input, .form-group button {
            width: 100%;
            max-width: 250px; /* Adjust as needed */
        }
        .error {
            color: red;
            margin-top: 10px;
        }
    </style>
</head>
<body>
    <div class="registration-container">
        <h2>Register</h2>
        <!-- Login Form -->
        <form th:action="@{/register}" method="post" class="form-group">
            <!-- Username -->
            <input type="text" name="username" placeholder="Username" required>
            <!-- Password -->
            <input type="password" name="password" placeholder="Password" required>
            <!-- Submit Button -->
            <button type="submit">Register</button>
        </form>
        <!-- Error Message -->
        <div th:if="${param.error}" class="error">
            Username already exists
        </div>
        <!-- Link to Login Page -->
        <p>Already have an account? <a href="/login">Login here</a></p>
    </div>
</body>
</html>
```

3. Create `login.html` file inside `resources/templates` folder.

4. Add the following code in `login.html`.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Login Page</title>
    <style>
        /* Center the login container vertically and horizontally */
        body {
            font-family: Arial, sans-serif;
            background-color: #f4f4f4;
            display: flex;
            justify-content: center;
            align-items: center;
            height: 100vh;
            margin: 0;
        }
        /* Style the login container */
        .login-container {
            background-color: #fff;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
            width: 300px;
            text-align: center;
        }
        /* Style the form elements */
        .login-container h2 {
            margin-bottom: 20px;
        }
        .login-container input {
            width: 100%;
            padding: 10px;
            margin: 10px 0;
            border: 1px solid #ccc;
            border-radius: 4px;
        }
        .login-container button {
            width: 100%;
            padding: 10px;
            background-color: #28a745;
            color: #fff;
            border: none;
            border-radius: 4px;
            cursor: pointer;
        }
        .login-container button:hover {
            background-color: #218838;
        }
        .error {
            color: red;
            margin-top: 10px;
        }
```

```
        .success {
            color: green;
            margin-top: 10px;
        }
        /* Flexbox layout for form items */
        .form-group {
            display: flex;
            flex-direction: column;
            align-items: center;
        }
        .form-group input, .form-group button {
            width: 100%;
            max-width: 250px; /* Adjust as needed */
        }
    </style>
</head>
<body>
    <div class="login-container">
        <h2>Login</h2>
        <!-- Login Form -->
        <form th:action="@{/login}" method="post" class="form-group">
            <!-- Username -->
            <input type="text" name="username" placeholder="Username" required>
            <!-- Password -->
            <input type="password" name="password" placeholder="Password" required>
            <!-- Submit Button -->
            <button type="submit">Login</button>
        </form>
        <!-- Error Message -->
        <div th:if="${param.error}" class="error">
            Invalid username or password.
        </div>
        <!-- Success Message -->
        <div th:if="${param.success}" class="success">
            User registered successfully!
        </div>
        <!-- Link to Login Page -->
        <p>Don't have an account? <a href="/register">Register here</a></p>
    </div>
</body>
</html>
```

5. Create `greet.html` file inside `resources/templates` folder.
6. Add the following code in `greet.html`.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Greeting Page</title>
</head>
<body>
    <h1>Hello, <span th:text="${username}"></span>!</h1>
    <form th:action="@{/logout}" method="post">
        <button type="submit">Logout</button>
    </form>
</body>
</html>
```

This page uses thyme to get the logged in user's name and display.

Configure properties and run app

1. Open `resources/application.properties` and add the following content in it. You can change the port number here, if you need to.

```
# Server port
server.port=8080
# Thymeleaf configuration
spring.thymeleaf.cache=false
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
```

2. Change to the project directory.

```
cd /home/project/secureapp
```

3. Use `mvn clean` (maven clean) to clean any existing class files and and `mvn install` (maven install) to compile the files in the project directory and generate the runnable jar file.

```
mvn clean install
```

4. Run the following command to start the application from the jar file.

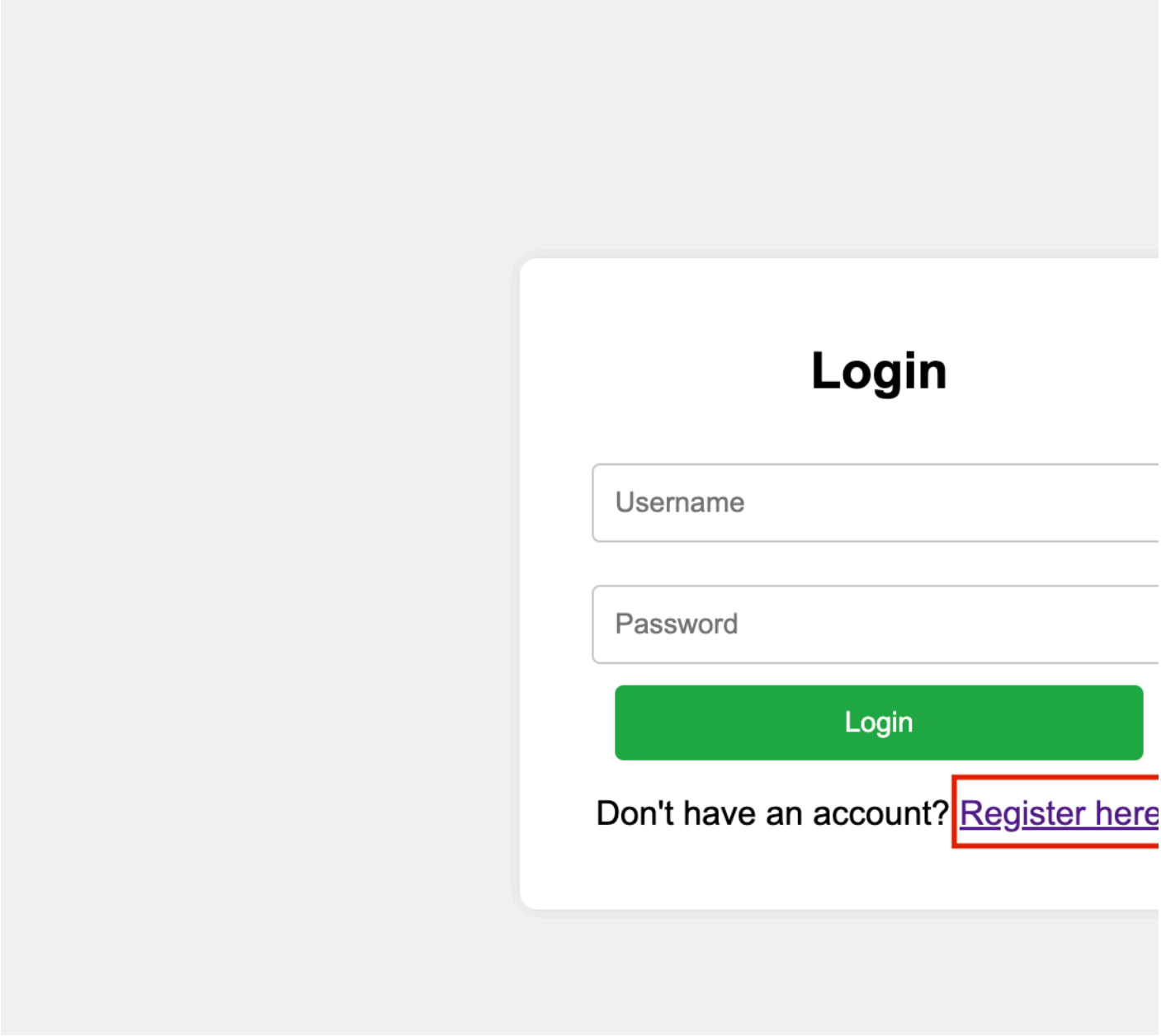
```
java -jar target/secureapp-0.0.1-SNAPSHOT.jar
```

The server will start in port 8080.

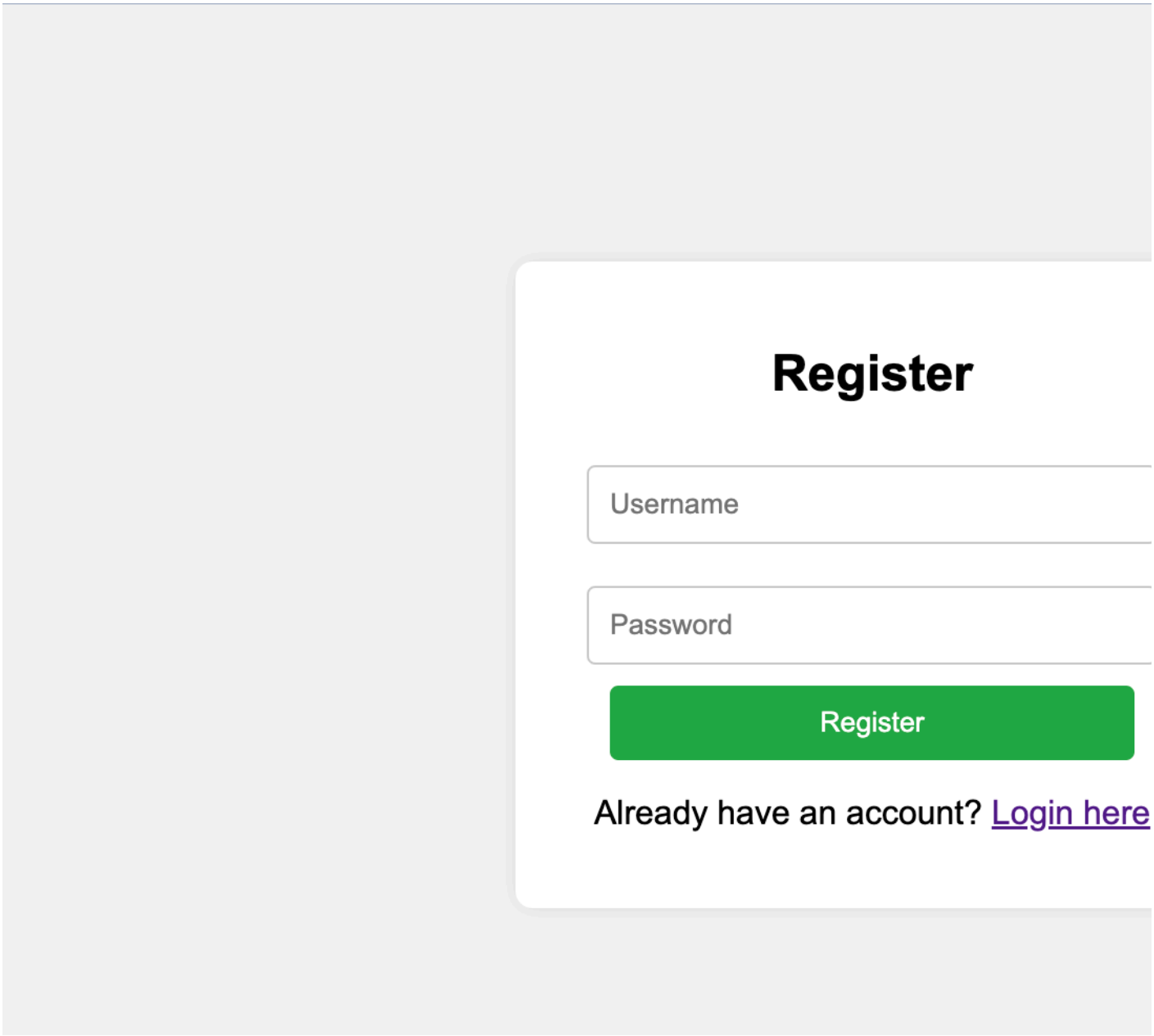
5. Click on the button below to open the browser page to access the end point.

Secure Web Application

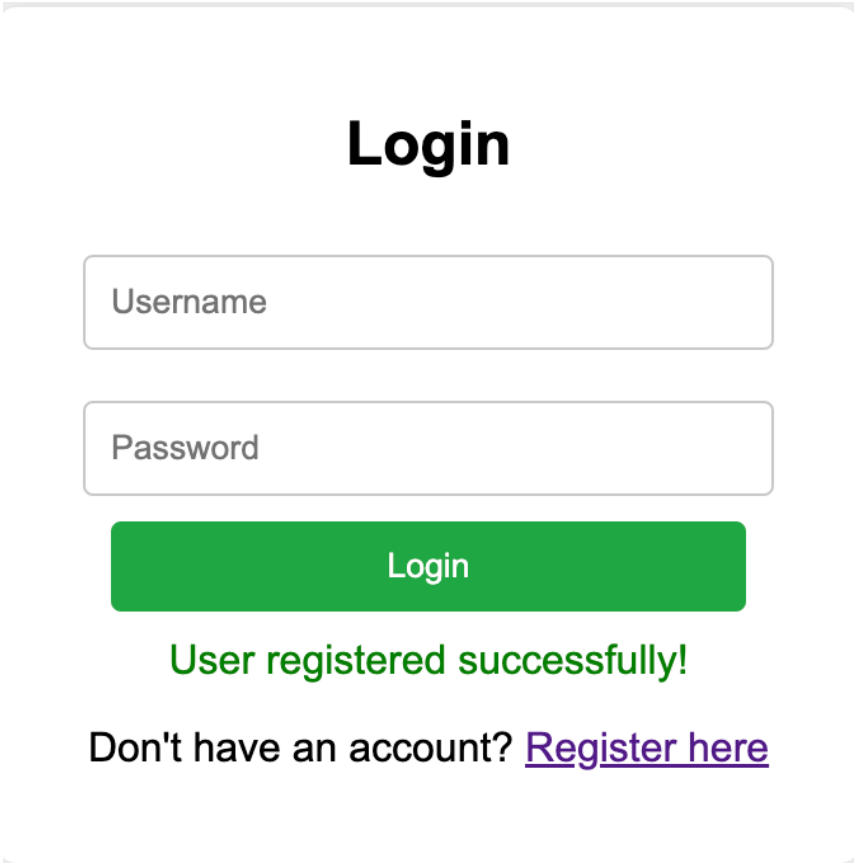
You will see the login page as give below.



6. First you need to register a user to login. Click the `Register Here` link. This will take you to the register page.



7. Enter the user login and password and select `Register`. Once the user registers, the registration is confirmed and login page is displayed.



8. Log in with the username and password you registered with. Once the login is successful, `Hello` greeting is displayed along with the username.

9. When the username already registered is used to register again, an appropriate error message is displayed. When the username or password provided for login is incorrect, an appropriate error message is displayed.

Practice exercise

1. Test all the erroneous data with the login and register end points.
2. Add register and login option to the web page application created in the previous labs.

Conclusion

In this lab, you have:

- Used Spring Initializr to generate the project
- Created the required `User` model with getters and setters
- Create a controller which uses the model and handles http requests for user registration and login
- Used WebSecurity configration using Spring web security
- Build and run the spring boot application
- Test the endpoints on the browser

Author(s)

Lavanya