
CHAPTER 5

SPRING SECURITY

Introduction

- Spring Security is a framework that focuses on providing both authentication and authorization (access control) to java web application and SOAP/REST web services.
- Spring framework supports integration with many technologies
 - HTTP basic authentication
 - LDAP
 - OpenID providers
 - JAAS API
 - ...
 - And customized authentication system (by yourself)

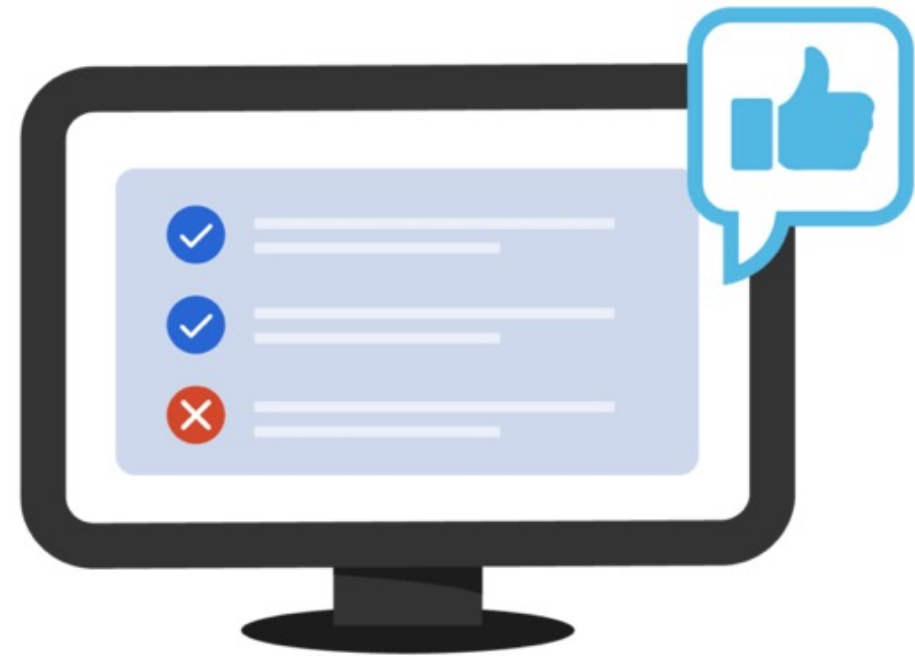
Authentication vs Authorization

Authentication



Confirms users
are who they say they are.

Authorization



Gives users permission
to access a resource.

Authentication vs Authorization

	Authentication	Authorization
What does it do?	Verifies credentials	Grants or denies permissions
How does it work?	Through passwords, biometrics, one-time pins, or apps	Through settings maintained by security teams
Is it visible to the user?	Yes	No
It is changeable by the user?	Partially	No
How does data move?	Through ID tokens	Through access tokens

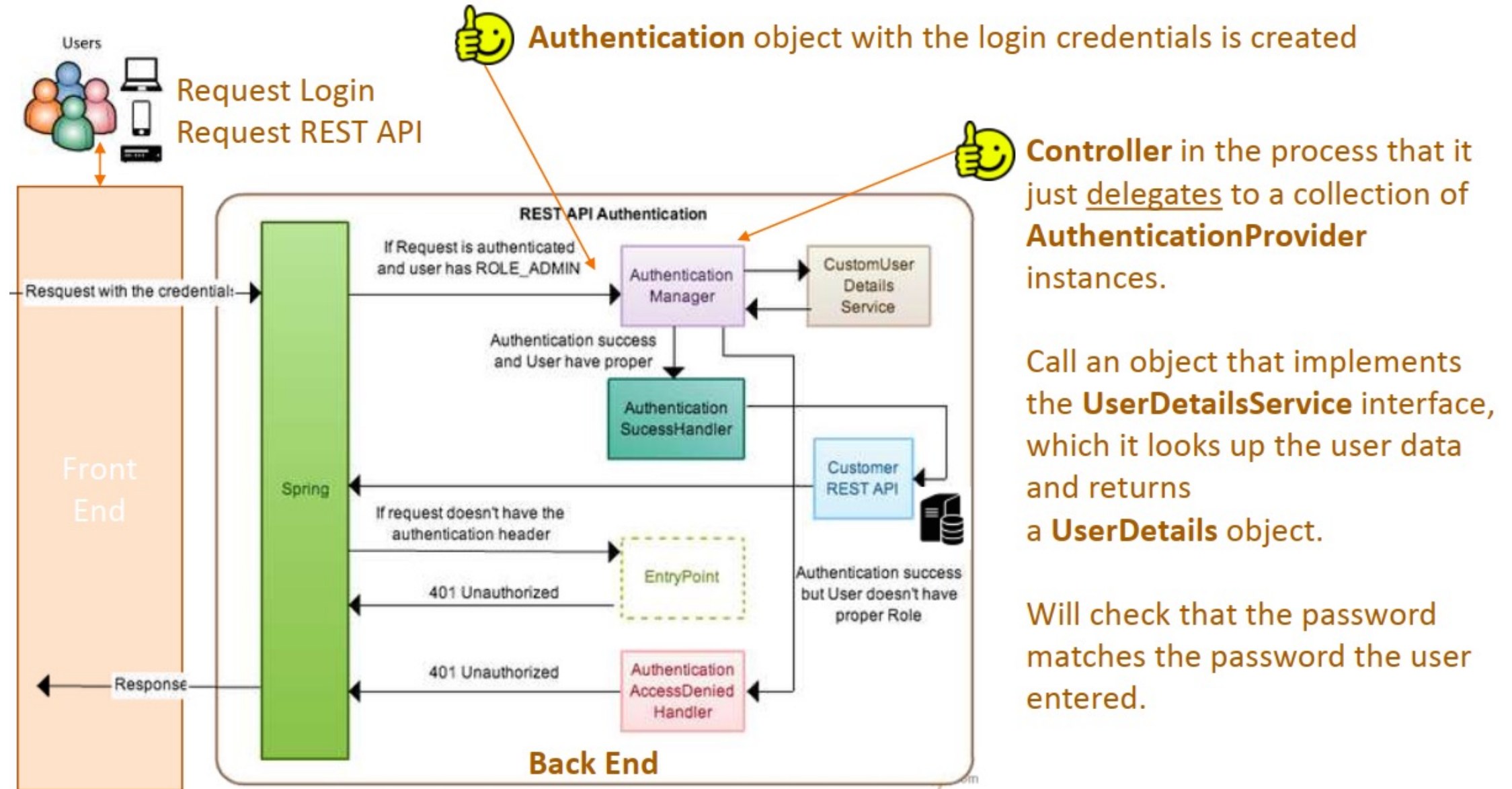
Terminologies

- Principal: User that performs the action
- Authentication: Confirming truth of credentials
- Authorization: Define access policy for principal
- GrantedAuthority: Application permission granted to a principal
- SecurityContext: Hold the authentication and other security information
- SecurityContextHolder: Provides access to SecurityContext

Terminologies

- AuthenticationManager: Controller in the authentication process
- AuthenticationProvider: Interface that maps to a data store which stores your user data.
- Authentication Object: Object is created upon authentication, which holds the login credentials.
- UserDetails: Data object which contains the user credentials, but also the Roles of the user.
- UserDetailsService: Collects the user credentials, authorities(roles) and build an UserDetails object.

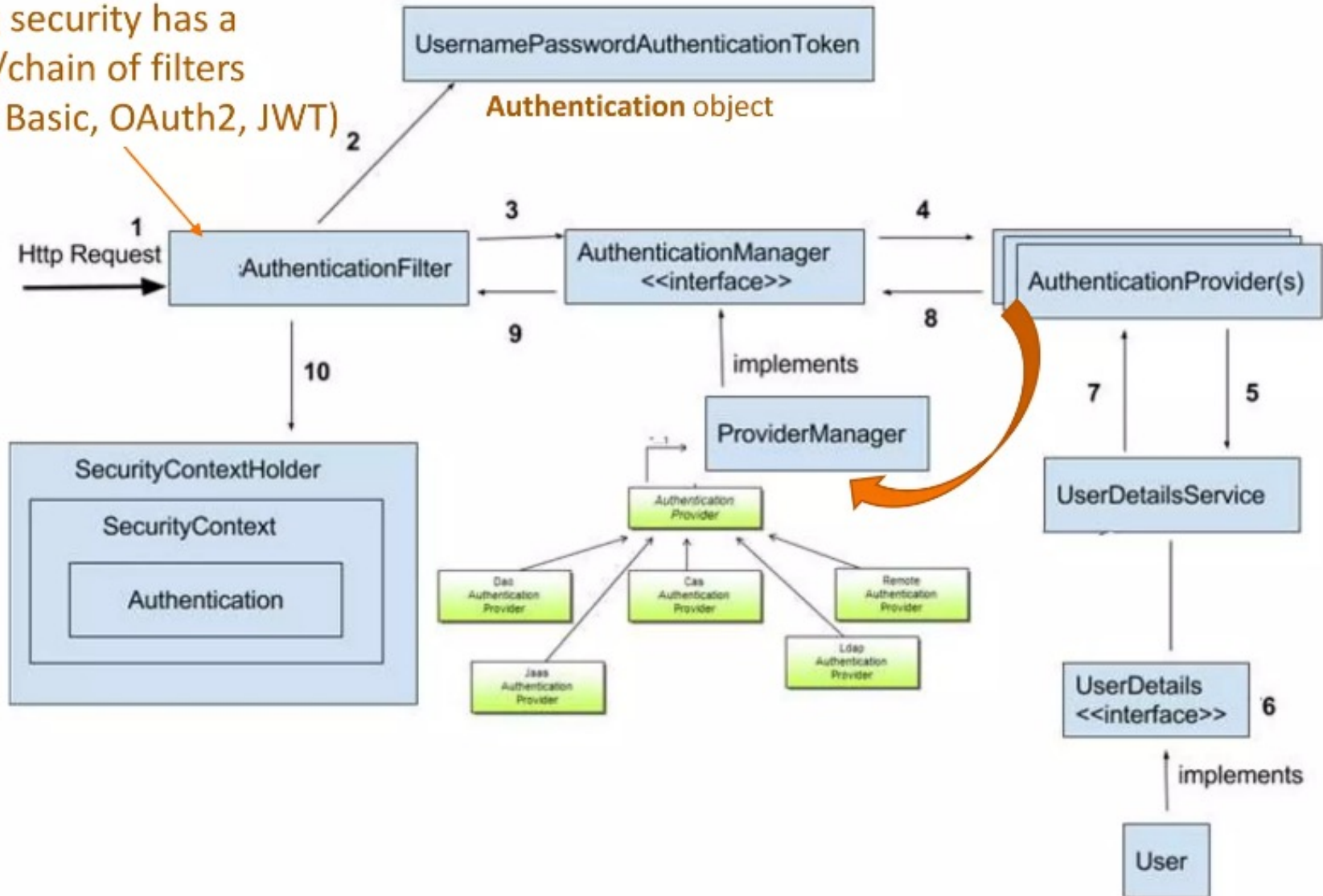
Architecture



Flow



Spring security has a series/chain of filters (HTTP Basic, OAuth2, JWT)



Practice

- In Spring 6.x, by adding the dependency to project, when you access any resources, you should provide the credential.
- To manage the resources grant/deny permission, you should configure them using XML or by code.
- There are many way to implements the security in Spring.
- Dependencies:
 - Spring Web
 - Spring Data JPA
 - Spring Security
 - Spring Boot Devtools
 - MariaDB Driver

Practice (cont.)

Auto Config with SpringBoot

- Default username: **user**
- Default password: the password generate when we run the application

```
2024-11-02T09:18:35.973+07:00 WARN SpringWebMVCDemo
m.s.s.UserDetailsServiceAutoConfiguration
```

```
Using generated security password: 6a7dd9d1-495b-4e0b-
beb7-d181e940b708
```

```
This generated password is for development use only. Your
security configuration must be updated before running
your application in production.
```

Practice (cont.)

Auto Config with SpringBoot

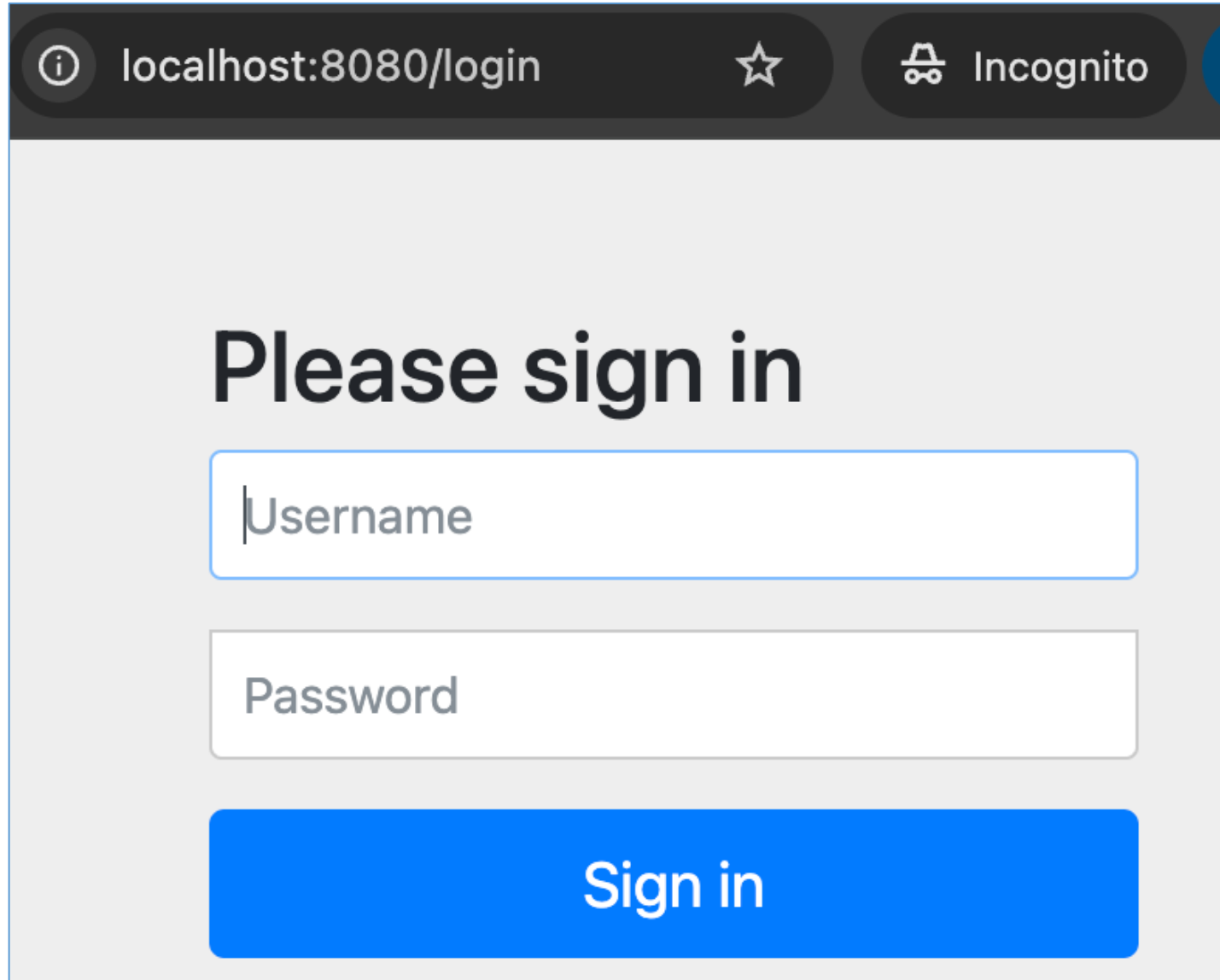
- We can change the default password by specify in `application.properties`
- URL generator password: <https://bcrypt-generator.com>

```
# Security
spring.security.user.name=user01

# 123456
spring.security.user.password=$2a$12$i0bonNQB3xnev8Nzj
9pEJeNCKdrf1fWtrim8VuxhQVUSfniNy1JzK
```

Practice (cont.)

Default Login Form



The image shows a web browser window with a dark header bar. The address bar displays 'localhost:8080/login' with an information icon on the left and a star icon on the right. To the right of the address bar is a tab labeled 'Incognito' with a shield icon. The main content area has a light gray background and contains the text 'Please sign in' in a large, bold, black font. Below this text are two input fields: the first is labeled 'Username' and the second is labeled 'Password'. Both fields are white with a thin blue border. At the bottom of the form is a large blue button with the text 'Sign in' in white.

localhost:8080/login

Incognito

Please sign in

Username

Password

Sign in

Spring Security

Configuration

- Sample steps:
 - Create any class with the `@Configuration` annotation.
 - Add `@EnableWebSecurity`
 - Create a method with a parameter with type `AuthenticationManagerBuilder`
 - Create a bean with return type `SecurityFilterChain` and a parameter with type `HttpSecurity`

Spring Security

Configuration (cont.)

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder
auth) throws Exception {}

    @Bean
    SecurityFilterChain filterChain(HttpSecurity httpSecurity)
throws Exception {}

    @Bean
    PasswordEncoder passwordEncoder() {}
}
```

Spring Security

Resource's grant/deny permission

- You should specify resources for protecting.
 - permitAll()
 - authenticated()
 - denyAll()
 - hasAnyRole()
 - hasRole()
 - ...

Spring Security

Providing credentials

- You can provide credential by many ways:
 - In memory
 - Using JDBC
 - Using OAuth2AuthorizeRequest
 - Using LDAP
 - ...
- In this study, in memory and using JDBC credentials are used.

Credentials

In-memory credential

*Provide role
for users*

*Using
BCryptPasswordEncoder
for encoding
the password*

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private static final String ROLE_ADMIN = "ADMIN";
    private static final String ROLE_USER = "USER";

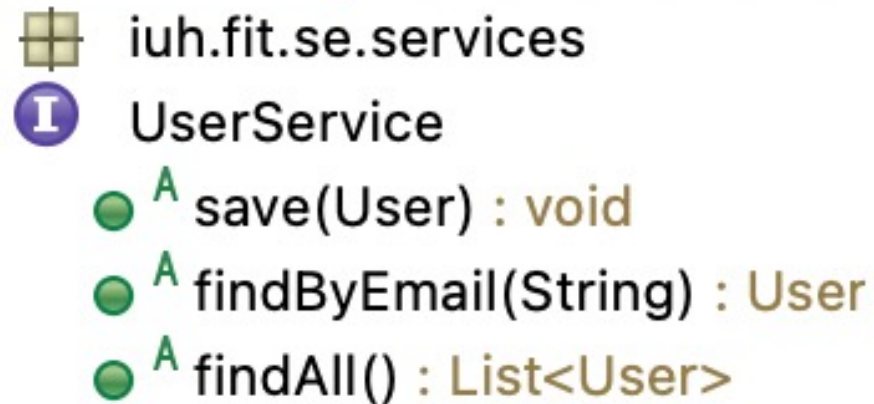
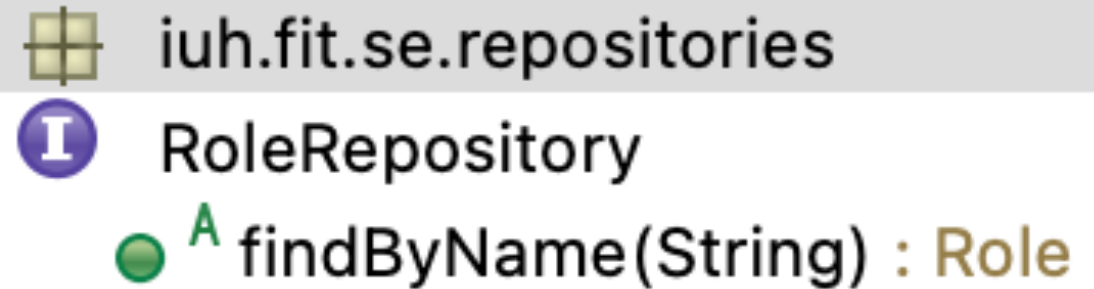
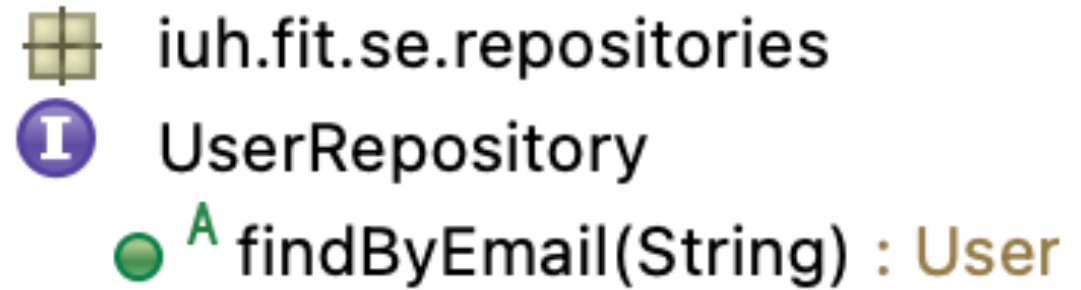
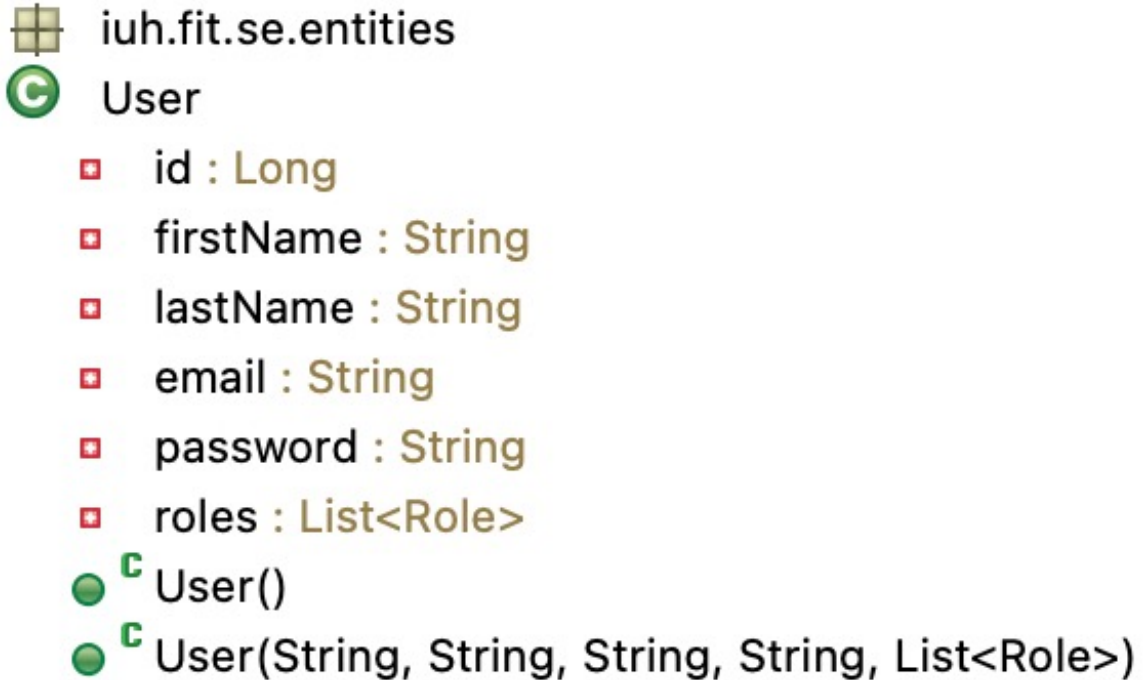
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        UserDetails userDetails01 = User
            .builder()
            .username("admin")
            .password("$2a$12$f2N6Rp5icyQZq6u/YuJUM.ATiTGz/j8pM0w4KzJWP1/uUL4b23xju")
            .roles(ROLE_ADMIN)
            .build();

        UserDetails userDetails02 = User.builder()
            .username("user01")
            .password("$2a$12$f2N6Rp5icyQZq6u/YuJUM.ATiTGz/j8pM0w4KzJWP1/uUL4b23xju")
            .roles(ROLE_USER)
            .build();

        auth.inMemoryAuthentication()
            .withUser(userDetails01)
            .withUser(userDetails02);
    }
}
```


Credentials


Using JDBC credential




Credentials

Using JDBC credential (cont.)


 iuh.fit.se.services.impl

 UserServiceImpl


- userRepository : UserRepository
- roleRepository : RoleRepository
- passwordEncoder : PasswordEncoder


 UserServiceImpl(UserRepository, RoleRepository, PasswordEncoder)

- ▲ save(User) : void
- ▲ findByEmail(String) : User
- ▲ findAll() : List<User>

 iuh.fit.se.controllers

 UserController

- userService : UserService
- showForm(Model) : String
-  ● registration(User, BindingResult, Model) : String
- getUsers(Model) : String

 iuh.fit.se.controllers

▼  HomeController

- home() : String
- login() : String

Credentials

Using JDBC credential (cont.)

```
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
@EnableWebSecurity
public class SpringSecurityConfig {

    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService)
            .passwordEncoder(passwordEncoder());
    }
}
```

Credentials

Using JDBC credential (cont.)

```
@Bean
SecurityFilterChain filterChain(HttpSecurity httpSecurity) throws Exception {
    httpSecurity.csrf(csrf -> {
        csrf.disable();
    })
    // Require authentication
    .authorizeHttpRequests(request -> request
        .requestMatchers("/employees/**").hasRole(ROLE_ADMIN)
        .requestMatchers("/home-role-user").hasAnyRole(ROLE_ADMIN, ROLE_USER)
        .anyRequest()
        .permitAll()
    )
    // Specify the login page and permit all access to it
    .formLogin(form -> {
        form.loginProcessingUrl("/login");
        form.defaultSuccessUrl("/employees");
        form.permitAll();
    })
    // Specify the logout request matcher and permit all access to it
    .logout(logout -> logout.permitAll())
    .httpBasic(Customizer.withDefaults());

    return httpSecurity.build();
}
```

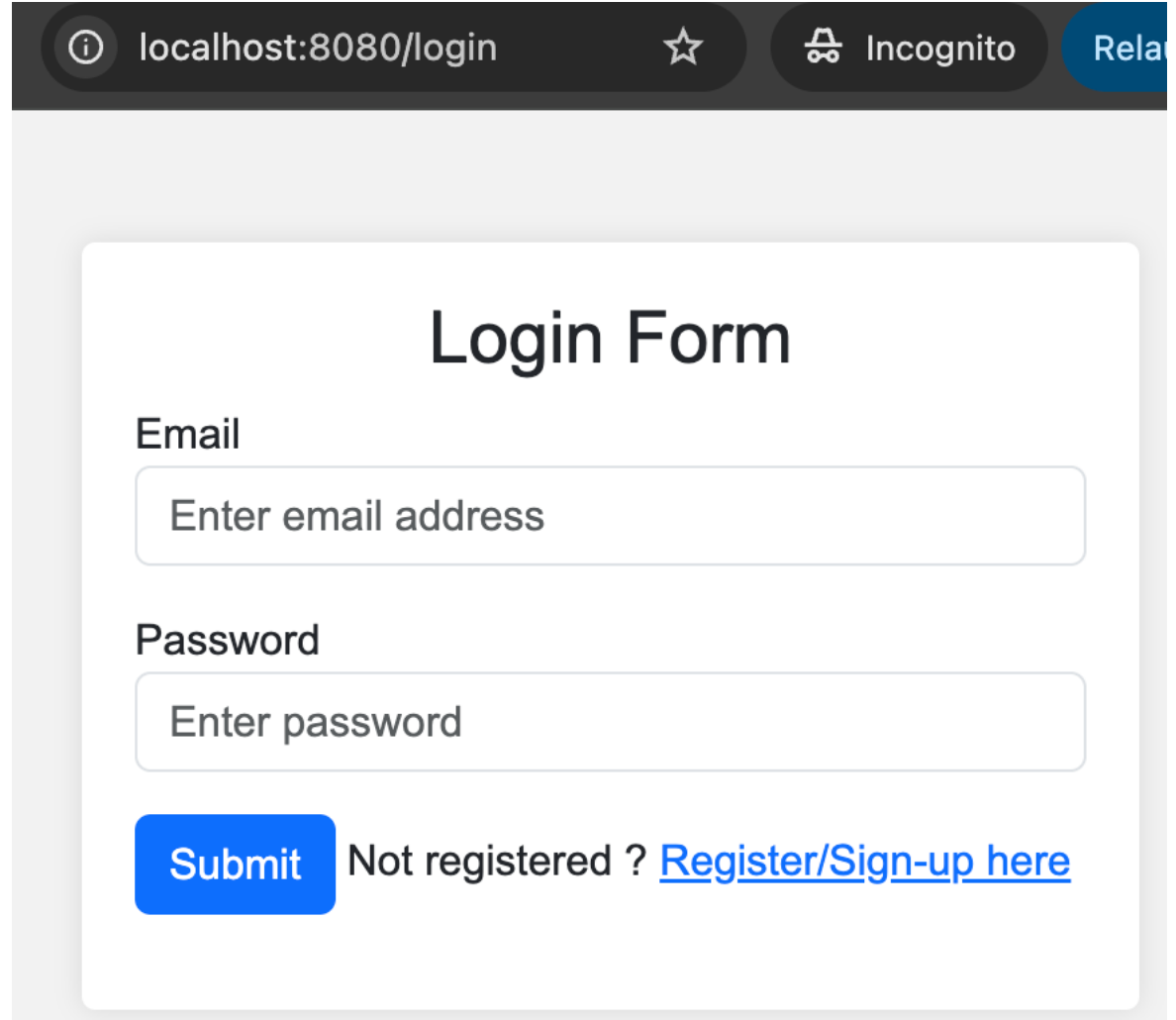
Credentials

Using JDBC credential (cont.)

```
@Controller
public class HomeController {

    @GetMapping
    public String index() {
        return "home";
    }

    @GetMapping("/home-role-user")
    public String homeRoleUser() {
        return "home-role-user";
    }
}
```



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/login'. The browser's Incognito mode is active. The main content area features a 'Login Form' with two input fields: 'Email' and 'Password'. The 'Email' field has a placeholder text 'Enter email address'. The 'Password' field has a placeholder text 'Enter password'. Below the password field is a blue 'Submit' button. To the right of the button is a link that says 'Not registered ? [Register/Sign-up here](#)'.

Credentials

Using JDBC credential (cont.)

```
<h2 class="text-center">Login Form</h2>
<form method="post" role="form" th:action="@{/login}" class="form-horizontal">
  <div class="form-group mb-3">
    <label class="control-label"> Email</label>
    <input type="text" id="username" name="username"
      class="form-control" placeholder="Enter email address" />
  </div>

  <div class="form-group mb-3">
    <label class="control-label"> Password</label>
    <input type="password" id="password" name="password"
      class="form-control" placeholder="Enter password" />
  </div>

  <div class="form-group mb-3">
    <button type="submit" class="btn btn-primary">Submit</button>
    <span> Not registered ?
      <a th:href="@{/users/register}">Register/Sign-up here</a>
    </span>
  </div>
</form>
```

Method level security

Method's security

- In case of specifying roles for accessing methods, you should enable method security.
- Using `@EnableMethodSecurity` annotation on any `@Configuration` class

```
@RestController
@RequestMapping("/api")
public class ProductResources {
    @GetMapping("/test")
    public ResponseEntity<Student> getTest() {
        ResponseEntity<Student> s = ResponseEntity.ok(new Student( id: 1001,
        return s;
    }

    @PreAuthorize("hasRole('ADMIN')")
    @GetMapping("/admin")
    public ResponseEntity<Student> testAdmin() {
        ResponseEntity<Student> s = ResponseEntity.ok(new Student( id: 1011, name: "Administrator"));
        return s;
    }
}
```

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {
```


Spring Security

Other techniques

- OAuth2 (Single sign-On)
- JSON Web Token (JWT)
- OpenID Connect
- ...

Cross-Site Request Forgery (CSRF)

Introduction

- Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.
- With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing.
- If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth.
- If the victim is an administrative account, CSRF can compromise the entire web application.
- There are multiple forms of CSRF attack. Two simple CSRF attacks that most common happen:
 - Using GET method
 - Using POST method

CSRF attacks

Using GET method

- Let's consider the following GET request used by a logged-in user to transfer money to a specific bank account 1234

GET <http://bank.com/transfer?accountNo=1234&amount=100>

- If the attacker wants to transfer money from a victim's account to his own account instead — 5678 — he needs to make the victim trigger the request:

GET <http://bank.com/transfer?accountNo=5678&amount=1000>

CSRF attacks

Using GET method (cont.)

- There are multiple ways to make that happen:

- Link – The attacker can convince the victim to click on this link, for example, to execute the transfer:

```
<a href="http://bank.com/transfer?accountNo=5678&amount=1000">  
  Show Kittens Pictures  
</a>
```

- Image – The attacker may use an `` tag with the target URL as the image source. In other words, the click isn't even necessary. The request will be automatically executed when the page loads:

```

```

CSRF attacks

Using POST method

- Suppose the main request needs to be a POST request:

POST <http://bank.com/transfer?accountNo=1234&amount=100>

- In this case, the attacker needs to have the victim run a similar request:

POST <http://bank.com/transfer?accountNo=5678&amount=1000>

- Neither the `<a>` nor the `` tags will work in this case.
- The attacker will need a `<form>`:

```
<form action="http://bank.com/transfer"
method="POST">
  <input type="hidden" name="accountNo"
value="5678"/>
  <input type="hidden" name="amount" value="1000"/>
  <input type="submit" value="Show Kittens Pictures"/>
</form>
```

The form can be submitted automatically using JavaScript:

```
<body onload="document.forms[0].submit()">
<form>
  ...
</form>
```

Spring MVC Web with CSRF

Client prevent CSRF attack configuration

- We need to include the CSRF token in our requests. The `_csrf` attribute contains the following information:
 - *token* – the CSRF token value
 - *parameterName* – name of the HTML form parameter, which must include the token value
 - *headerName* – name of the HTTP header, which must include the token value
- In case of using HTML forms, the *headerName* and *token* values should be added HTTP header.

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
```

- In case of using JSON view forms, the *headerName* and *token* values should be added HTTP header.

Spring MVC Web with CSRF

Client prevent CSRF attack configuration (cont.)

- In case of using JSON view forms, the *headerName* and *token* values should be added HTTP header.

//1. include the token value and the header name in meta tags

```
<meta name="_csrf" content="${_csrf.token}"/>
<meta name="_csrf_header" content="${_csrf.headerName}"/
```

//2. Then let's retrieve the meta tag values with JQuery:

```
var token = $("meta[name='_csrf']").attr("content");
var header = $("meta[name='_csrf_header']").attr("content");
```

// 3. Finally, let's use these values to set our XHR header:

```
$(document).ajaxSend(function (e, xhr, options) {
    xhr.setRequestHeader(header, token);
});
```

Q&A