Lecture 6

Thymeleaf Notes
Spring MVC Examples
Spring Security Basics

Topics covered

- ✧ Thymeleaf notes
  - Expressions
  - Layout / Fragment
- ✧ Spring MVC + Thymeleaf Examples
- ✧ Spring Security basics
  - Security Chain setup
  - Managing users with JDBC

Some Thymeleaf Notes

Thymeleaf Expressions

- ✧ Thymeleaf expressions outside of `th:*` attributes will not be processed and *will be treated as plain text*.
  - This is intentional by design as Thymeleaf prioritizes **compatibility with standard HTML** and aims to function seamlessly even if used in non-Thymeleaf projects.
- ✧ 5 types of Thymeleaf expressions:
  - `${...}` : Variable expressions (Spring EL)
  - `*{...}` : Selection expressions.
  - `#{...}` : Message (i18n) expressions.
  - `@{...}` : Link (URL) expressions.
  - `~{...}` : Fragment expressions.

Thymeleaf Variable Expressions

- ✧ Called Spring EL when integrating Thymeleaf with Spring
- ✧ Used to work with *context variables* (a.k.a. model attributes)

Controller method: adds a variable to the `model` object

```
@RequestMapping(value = "/add")
public String addEmployee(Model model) {
    Employee employee = new Employee();
    model.addAttribute("employee", employee);
    return "employeeAdd";
}
```

Thymeleaf template: uses that variable in a variable expression

```
<form th:action="'/insert'" th:object="${employee}" method="post">
```

Link (URL) Expressions

- ✧ Because of the importance of URLs web applications, Thymeleaf has a special syntax for them
  - The `@` syntax: `@{...}`
- ✧ Different types of URLs:
  - Absolute URLs: `http://www.thymeleaf.org`
  - Relative URLs
- ✧ Examples of relative URLs:
  - Page-relative: `user/login.html`
  - Context-relative: `/itemdetails?id=3` (context path in server will be added automatically)
  - Server-relative: `~/billing/processInvoice` (allows calling URLs in another context in the same server.
  - Protocol-relative URLs: `//code.jquery.com/2.0.3.min.js`

Link (URL) Expressions

```
<!-- Will produce 'http://localhost:8080/order/details?orderId=3' -->
<a href="details.html"
   th:href="@{http://localhost:8080/order/details(orderId=${o.id})}">view</a>

<!-- Will produce '/order/details?orderId=3' -->
<a href="details.html" th:href="@{/details(orderId=${o.id})}">view</a>

<!-- Will produce '/order/3/details' -->
<a href="details.html" th:href="@{/{orderId}/details(orderId=${o.id})}">view</a>
```

- `th:href` is a modifier attribute: once processed, it will compute the link URL to be used and set that value to the `href` attribute of the `<a>` tag
- URL may have one or many parameters (separated by commas)
- Variables are also allowed in URL paths: `@{/{orderId}/details(orderId=${orderId})}`

Fragments

- ✧ In our templates, we will often want to include parts from other templates, parts like footers, headers, menus…
- ✧ Fragment expressions are an easy way to represent fragments of markup and move them around templates
  - ▪ This allows us to replicate them, pass them to other templates as arguments, and so on

Defining Fragments

- Thymeleaf allows us to define fragments (for inclusion later in other places) using the `th:fragment` attribute
- Example:
  - We want to add a standard copyright footer to all our pages, so we create a /resources/templates/footer.html file containing this code:

```
<div th:fragment="copy">
    &copy; 2024 FIT HANU

</div>
```

  - Then we can include this in any page using `th:insert` or `th:replace` attributes.

```
<div th:insert="~{footer :: copy}"></div>
```

Fragment Expressions

- ✧ A fragment expression (`~{...}`) is an expression that results in a fragment.
- ✧ The syntax of *fragment expressions*:

```
"~{templatename::selector}"
```

- ✧ The previous example was a non-complex one and can also be written as:

```
<div th:insert="footer :: copy"></div>
```

  - `th:insert` is the simplest: it will simply insert the specified fragment as the body of its host tag
  - `th:replace` actually replaces its host tag with the specified fragment

Expression Objects (reference docs)

- When evaluating expressions, some objects are available for use. They are referenced using # symbol.
- Expression Basic Objects
  - `#ctx`: the context object
  - `#vars`: the context variables
  - `#request`: *(for Web context)* the `HttpServletRequest` object
  - `#response`: *(for Web context)* the `HttpServletResponse` object
  - `#session`: *(for Web context)* the `HttpSession` object
- Expression Utility Objects
  - URIs/URLs
  - Dates / Numbers / Strings
  - Arrays / Lists / Sets / Maps

Formatting Date

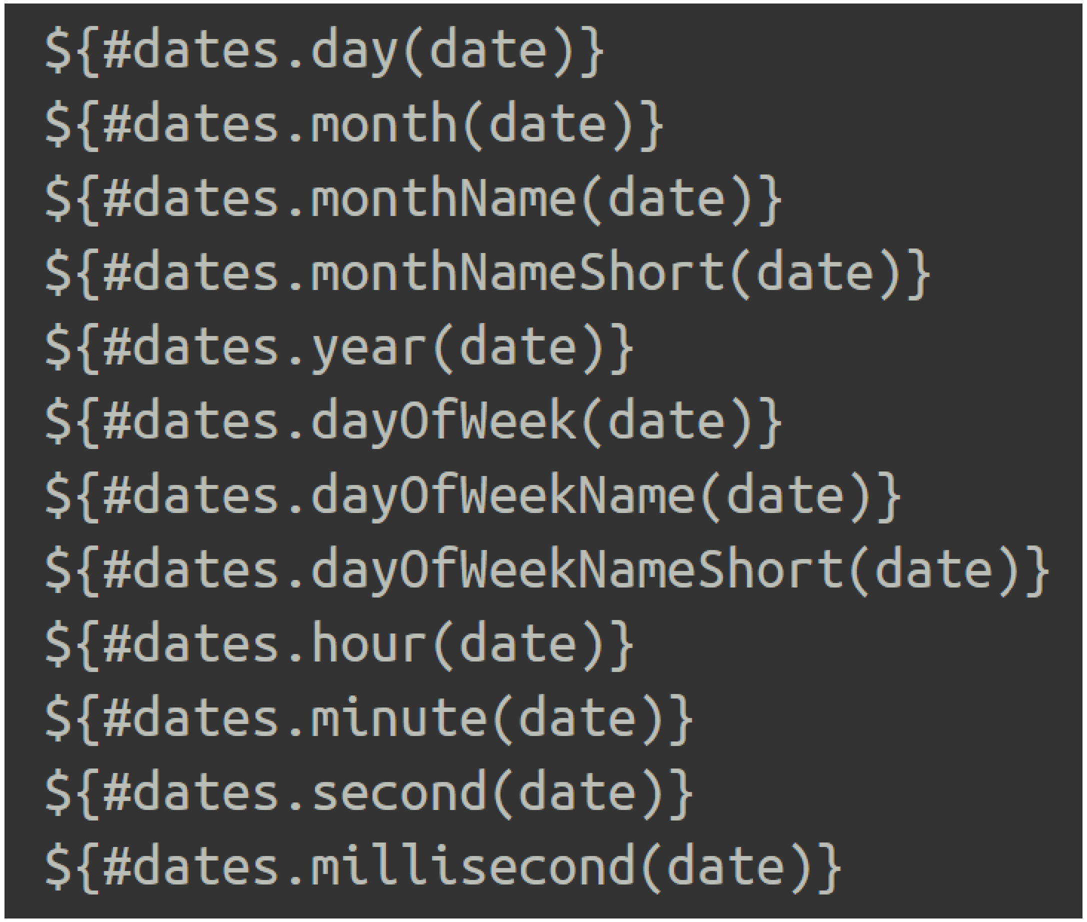
- ✧ You can format a `java.util.Date` object inside a Thymeleaf template:

```
${#dates.format(date, 'dd/MMM/yyyy HH:mm')}
```

```
${#calendars.format(date, 'dd/MMM/yyyy HH:mm')}
```

Extracting Date Values

✧ It is possible to extract date properties from `Date` object.



```
${#dates.day(date)}
${#dates.month(date)}
${#dates.monthName(date)}
${#dates.monthNameShort(date)}
${#dates.year(date)}
${#dates.dayOfWeek(date)}
${#dates.dayOfWeekName(date)}
${#dates.dayOfWeekNameShort(date)}
${#dates.hour(date)}
${#dates.minute(date)}
${#dates.second(date)}
${#dates.millisecond(date)}
```

URL Escaping

- URLs should be escaped to eliminate the side effects of special characters.
  - Such as `?`, `%`, `&`, and Unicode characters

In controller method:

```
model.addAttribute(
        "myURL",
        "http://fit.hanu.vn/?first=Quân&last=Đặng");
```

In template:

```
<div th:text="${#uris.escapePath(myURL)}"></div>
```

HTML result:

```
<div>http://fit.hanu.vn/%3Ffirst=Qu%C3%A2n&amp;
last=%C4%90%E1%BA%B7ng</div>
```

Formatting Numbers

✧ Formatting decimal with 3 integer digits and 2 decimal places:

```
${#numbers.formatDecimal(num,3,2)}
```

✧ Specifying thousand separator when formatting decimal number:

```
${#numbers.formatDecimal(num,3,2,'COMMA')}
```

Operations on Strings

```
${#strings.toString(obj)}

${#strings.isEmpty(name)}

${#strings.contains(name,'ez')}

${#strings.startsWith(name,'Don')}

${#strings.indexOf(name,frag)}

${#strings.substring(name,3,5)}

${#strings.toUpperCase(name)}

${#strings.arraySplit(namesStr,',')}

${#strings.arrayJoin(namesArray,',')}

${#strings.trim(str)}

${#strings.length(str)}
```

Thymeleaf – Check if a variable exists

- Two ways:
  - `#ctx.containsVariable`
    - The `#ctx` object represents the current template context and holds all passed variables
  - compare with `null` value

```
<p th:if="#ctx.containsVariable('message')">
    The message is: [[${message}]]
</p>

<p th:if="${message != null}">
    The message is: [[${message}]]

</p>
```

Handling a POST request in Spring MVC

- The `@RequestBody` annotation wouldn't work in many cases because no converter object is found for a form's content type.
  - In that case, a "Media Type Not Supported" error (code 415) is returned.
- The regular `www-urlencoded` POST request (sent by an HTML form submission) can be received as an object by using the `@ModelAttribute` annotation (or no annotation).

```
@RequestMapping(value = "/handleAddStudent",
        method = RequestMethod.POST)
public String handleAddStudent(@ModelAttribute Student s,
        Model model) {
    model.addAttribute("student", s);
    return "studentAdd";

}
```

Spring MVC + Thymeleaf Examples

- ✧ Using spring-boot-devtools
  - Auto restarting app to speed up development
- ✧ Populating a form in Thymeleaf
- ✧ Spring MVC HTTP Message Conversion
  - Converting form data into entity instance
  - Add vs. Edit example
- ✧ Call a Bean’s method from Thymeleaf view
  - The `${@beanName.method()}` syntax

Spring Security Basics

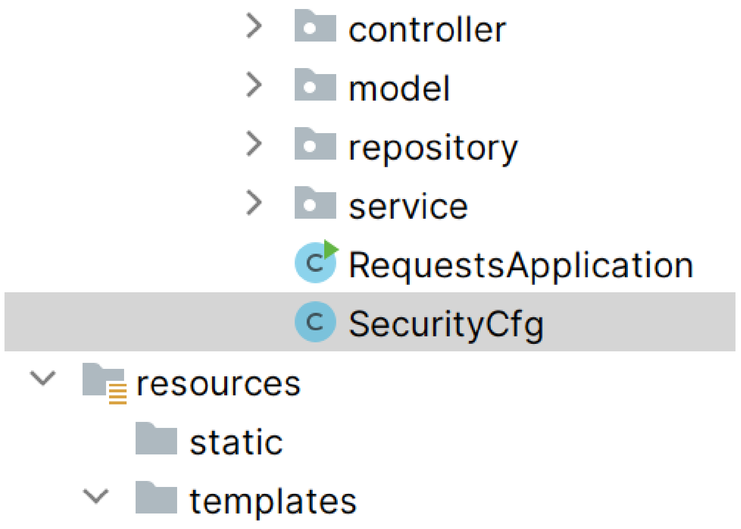
Enabling Spring Security in your project

- ✧ With Spring Boot, add the `spring-boot-security-starter` dependency into your `pom.xml`

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

</dependency>
```

- ✧ Add a configuration class in your project
  - Usually in a `config` sub-package or the main package of your project



Spring Security configuration class

- This class should be annotated with the `@Configuration` and `@EnableWebSecurity` annotations.

```
@Configuration
@EnableWebSecurity
public class SecurityCfg {
    @Bean
    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        return http
                // insert filters here
                .build();
    }
}
```

Using `AuthorizeHttpRequestConfigurer`

- ✧ Authorization of HTTP requests is configured by invoking the `authorizeHttpRequests()` method from the `HttpSecurity` object.
- ✧ We add rules that allow or restrict access to different (sets of) URLs.
- ✧ The order of the rules inside a customizer is significant
  - The more specific rules need to come first, followed by the more general ones.

```
.authorizeHttpRequests(req -> req
        // let everyone access homepage
        .requestMatchers("/").permitAll()
        // other URLs are for authenticated users only
        .anyRequest().authenticated()
)
```

Using default login implementation

- ✧ By default, Spring Security implements login page at `/login` and the logout URL is `/logout`
- ✧ Upon successful login, the application redirects to the previous URL and appends `?continue` to it.
- ✧ Upon failed login, the application redirects to `/login?error`

```
.formLogin(Customizer.withDefaults())
```

- ✧ However, it is desirable to change these defaults to not let attackers know that we're using Spring Security (for security purpose)

Using `FormLoginConfigurer`

- ✧ Form-based is one of several authentication methods
  - Others include basic HTTP, token-based, OAuth2…
- ✧ The below example:
  - Changes the login URL, login processing URL, the URLs to redirect to after successful/failed login.
  - Use `permitAll()` to make these URLs accessible to unauthenticated users.

```
.formLogin(formLogin -> formLogin
        .loginPage("/sign-in")
        // by default, redirects to previous URL
        // (adding "?continue") if login successfully
        // redirects to "/login?error" if failed
        .loginProcessingUrl("/sign-in")
        .defaultSuccessUrl("/member", true)
        .failureUrl("/sign-in?error=true")
        .permitAll()
)
```

Creating custom login page

✧ Create your own login template using Thymeleaf

```html
<form method="post" th:action="'/sign-in'">
    <h2>Please sign in</h2>
    <p th:if="${param.error}">Invalid credentials!</p>
    <p th:if="${param.logout}">You have been logged out!</p>
    <p>
        <label for="username">Username</label>
        <input type="text" id="username" name="username"/>
    </p>
    <p>
        <label for="password">Password</label>
        <input type="password" id="password" name="password"/>
    </p>
    <input type="submit" value="Sign in"/>

</form>
```

Creating custom authentication controller

✧ Create a controller method for mapping the login page template:

```java
@Controller
public class AuthController {
    @GetMapping("/sign-in")
    public String login() {
        return "sign-in";
    }

}
```

✧ In my project, I dedicate this `AuthController` to authentication-related stuffs (sign in, sign up, etc.)

Configuring Logout feature

- We can change logout URL and the URL to redirect to after successful logout.
  - No need to have a controller method for logout.

```
.logout(logout -> logout
        .logoutUrl("/sign-out")
        .logoutSuccessUrl("/sign-in?logout=true")
        .permitAll()
)
```

Configuring CSRF

- ✧ CSRF stands for Cross-Site Request Forgery.
  - E.g: Attacker tricks victim to sends POST requests from a different site to the target site for changing account’s email address…
- ✧ Spring Security has built-in CSRF protection.
  - But it requires us to use POST request for the logout feature if CSRF protection is enabled
- ✧ For simplicity, we’re going to disable this feature.

```
http
// disable this so that we can visit "/sign-out"
// using GET method (otherwise, we have to do a POST)
.csrf(c -> c.disable())
```

Notes on `securityFilterChain` configuration

- ✧ The order of `csrf()`, `formLogin()`, `logout()` and `authorizeHttpRequest()` does not matter.
  - …as they are independent configurers
- ✧ For `requestMatchers`, the pattern can include wildcard characters
  - Question mark (`?`) matches a single character
  - Single asterisk (`*`) matches zero or more characters
  - Double asterisk (`**`) matches zero or more 'directories' in a path

User Details Managers

Spring Security Users Basics

- ✧ If no bean of type `UserDetailsManager` is found, Spring will generate a default user named `user` with a randomly generated password.
  - ▪ The password will be shown on the console when you start the application.
- ✧ There are two built-in user details managers:
  - ▪ `InMemoryUserDetailsManager`
  - ▪ `JdbcUserDetailsManager`

In-Memory User Details Manager

- This method is useful when you need to quickly add a few users for testing.
  - The `{noop}` string is added when we don’t want to use any `PasswordEncoder`
  - We’ll add and use a password encoder later

```java
@Bean
InMemoryUserDetailsManager users() {
    return new InMemoryUserDetailsManager(
            User.withUsername("quan")
                    .password("{noop}123123")
                    .roles("USER")
                    .build()
    );
}
```

Using a password encoder

✧ `BCryptPasswordEncoder` is the most popular one and is sufficient in professional usage.

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

@Bean
InMemoryUserDetailsManager users() {
    return new InMemoryUserDetailsManager(
            User.withUsername("quan")
                    .password(passwordEncoder().encode("123123"))
                    .roles("USER")
                    .build()
    );

}
```

Using `JdbcUserDetailsManager`

✧ Before using JDBC for storing users, you need to create the database tables for it.

- The default schema is also exposed as a CLASSPATH resource:

```
org/springframework/security/core/userdetails/jdbc/users.ddl
```

- This DDL script provided uses a dialect not suitable for MySQL → We need to modify it a little and execute it manually.

Setup Script for `JdbcUserDetailsManager`

```
create table users(
    username varchar(50) not null primary key,
    password varchar(500) not null,
    enabled boolean not null
);

create table authorities (
    username varchar(50) not null,
    authority varchar(50) not null,
    constraint fk_authorities_users foreign key(username)
    references users(username)
);
create unique index ix_auth_username
    on authorities (username,authority);
```

Configuring a `DataSource`

- ✧ A `DataSource` is important for any database-driven application.
- ✧ A `DataSource` can be created using various ways, one of which is using `application.properties`

```
# MYSQL
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=
```

Configuring a `DataSource`

✧ Instead of `InMemoryUserDetailsManager` can now be replaced by the `JdbcUserDetailsManager` in the configuration class.

```
@Bean
UserDetailsManager users(DataSource dataSource) {
    UserDetails user = User.builder()
            .username("quan")
            .password(passwordEncoder().encode("123"))
            .roles("USER")
            .build();
    JdbcUserDetailsManager users =
            new JdbcUserDetailsManager(dataSource);
    users.createUser(user);
    return users;

}
```

What’s next?

- ✧ JPA-based authentication
- ✧ User registration
- ✧ Linking users with other entities