Thymeleaf

What is Thymeleaf?

- Thymeleaf is a Java library.
- Thymeleaf is a templating library that can be easily integrated with Spring Boot applications
- It is an XML/XHTML/HTML5 template engine able to apply a set of transformations to template files in order to display data and/or text produced by your applications.
- It is better suited for serving XHTML/HTML5 in web applications, but it can process any XML file, be it in web or in standalone applications.
- The main goal of Thymeleaf is to provide an elegant and well-formed way of creating templates.
- In order to achieve this, it is based on XML tags and attributes that define the execution of predefined logic on the *DOM (Document Object Model)*, instead of explicitly writing that logic as code inside the template.
- **By default, Thymeleaf expects us to place those templates in the *src/main/resources/templates* folder.**

Dependency

- **<dependency>**
  <groupId>org.springframework.boot**</groupId>**
  <artifactId>spring-boot-starter-thymeleaf**</artifactId>**
  </dependency>

Hello Malaykum.

- Lets create hello malaykum project

HelloMalaykum Example

- Create Controller

```
@Controller
@RequestMapping("/")
public class InitController {
    @RequestMapping("/")
    public String index() {
        return "index";
    }
}
```

- Create index.html file inside resources/templates/ folder

- index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index Page</title>
</head>
<body>
<h1>Hello Page</h1>
</body>
</html>
```

How to pass any data to view

- Using Modal or ModalView class we can send data to view

Modal passing data example

```
@Controller
@RequestMapping("/")
public class InitController {
    @RequestMapping("/")
    public String index(Model model) {
        model.addAttribute("server_time", LocalDateTime.now());
        return "index";
    }
}
```

- In index.html to get value

```
<p th:text="${server_time}"></p>
```

```
xmlns:th="http://www.thymeleaf.org"
```

ModalAndView passing data example

```
@Controller
@RequestMapping("/")
public class InitController {

    @RequestMapping("/time")
    public ModelAndView time() {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("server_time", LocalDateTime.now());
        modelAndView.setViewName("index");
        return modelAndView;
    }
}
```

- In index.html page to get data

```
<p th:text="${server_time}"></p>
```

Displaying Model Attributes

- **Simple Attributes**
  - *th:text=”${attributename}”*
- The *th:text=”${attributename}”* tag attribute can be used to display the value of model attributes.
- Let's add a model attribute with the name *serverTime* in the controller class:
- Current time is

Differences between Model, ModelMap, and ModelAndView

- **Model:** It is an Interface. It defines a holder for model attributes and primarily designed for adding attributes to the model.
- **ModelMap:** Implementation of Map for use when building model data for use with UI tools.Supports chained calls and generation of model attribute names.
- **ModelAndView:** This class merely holds both to make it possible for a controller to return both model and view in a single return value.

Model

```
@RequestMapping(method = RequestMethod.GET)
    public String printHello(Model model) {
        model.addAttribute("message", "Hello World!!");
        return "hello";
    }
```

ModelAndView

```
@RequestMapping("/welcome")
public ModelAndView helloWorld() {
        String message = "Hello World!";
        return new ModelAndView("welcome", "message", message);
    }
```

ModelMap

```
@RequestMapping("/helloworld")
public String hello(ModelMap map) {
    String helloWorldMessage = "Hello world!";
    String welcomeMessage = "Welcome!";
    map.addAttribute("helloMessage", helloWorldMessage);
    map.addAttribute("welcomeMessage", welcomeMessage);
    return "hello";
}
```

Sending Object to view

Sending Object to view

```
@Controller
@RequestMapping("/")
public class InitController {

@RequestMapping("/student")
    public ModelAndView student() {
        StudentDTO studentDTO = new StudentDTO();
        studentDTO.setAge(5);
        studentDTO.setName("Alish");
        studentDTO.setSurname("Aliyev");

        ModelAndView modelAndView = new ModelAndView();
        modelAndView.addObject("server_time", LocalDateTime.now());
        modelAndView.addObject("student", studentDTO);
        modelAndView.setViewName("index");
        return modelAndView;
    }
}
```

Showing Object in view

- Inside index.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Index Page</title>
</head>
<body>
<h1>Hello Page</h1>
<p th:text="${server_time}"></p>
<div>
    Name: <span th:text="${student.name}"></span> <br>
    Surname: <span th:text="${student.surname}"></span> <br>
    Age: <span th:text="${student.age}"></span>
</div>
</body>
</html>
```

For loop - th:each 1

- In Thymeleaf, iteration is achieved by using the *th:each* attribute.
- One of the interesting things about this attribute is that **it will accept and iterate over some different data types**, such as:
  - objects implementing *java.util.Iterable*
  - objects implementing *java.util.Map*
  - arrays
  - any other object is treated as if it were a single-valued list containing one element

```
<tr th:each="student: ${studentList}">
    <td th:text="${student.id}" />
    <td th:text="${student.name}" />
</tr>
```

For loop - th:each - Status Variable

- Thymeleaf also **enables a useful mechanism to keep track of the iteration process via the status variable.**
- The status variable provides the following properties:
  - **index**: the current iteration index, starting with 0 (zero)
  - **count**: the number of elements processed so far
  - **size**: the total number of elements in the list
  - **even/odd**: checks if the current iteration index is even or odd
  - **first**: checks if the current iteration is the first one
  - **last**: checks if the current iteration is the last one
-

```
<tr
  th:each="student, iStat : ${studentList}"
  th:style="${iStat.odd}? 'font-weight: bold;'"
  th:alt-title="${iStat.even}? 'even' : 'odd'">
    <td th:text="${student.id}" />
    <td th:text="${student.name}" />
</tr
```

For loop - th:each – Controller Example

- In Controller

```
@RequestMapping("/list")
   public ModelAndView getList() {
       List<StudentDTO> studentList = new LinkedList<>();
       studentList.add(new StudentDTO("Ali", "Aliyev", 20));
       studentList.add(new StudentDTO("Vali", "Valiyev", 17));
       studentList.add(new StudentDTO("Toshmat", "Toshmatov", 22));
       studentList.add(new StudentDTO("Eshmat", "Eshmatov", 19));

       ModelAndView modelAndView = new ModelAndView();
       modelAndView.addObject("studentList", studentList);
       modelAndView.setViewName("student_list");
       return modelAndView;
   }
```

For loop - th:each – View Example

- In student_list.html page

```
<table>
    <tr>
        <th>№</th>
        <th>Age</th>
        <th>Name</th>
        <th>Surname</th>
    </tr>
    <tr th:each="student, iStat : ${studentList}">
        <td th:text="${iStat.count}"></td>
        <td th:text="${student.age}"></td>
        <td th:text="${student.name}"></td>
        <td th:text="${student.surname}"></td>
    </tr>
</table>
```

Condition

- th:if – if true
- th:unless – *if false*
- ="${add} ? 'Create a Contact:' : 'Edit a Contact:'"

Condition - *th:if and th:unless*

- The *th:if=”${condition}”* attribute is used to display a section of the view if the condition is True.
- The *th:unless=”${condition}”* attribute is used to display a section of the view if the condition is False.

- ```
  <td>

  </td>
  ```

- // example in student_temp page
  ```

Condition – inline condition

- <h1 th:text="${add} ? 'Create a Contact:' : 'Edit a Contact:'" />

- If add is True show ‘Create a Contact’
- Else shows ‘Edit a Contact’

Condition - switch and *case*

- The *th:switch* and *th:case* attributes are used to display content conditionally using the switch statement structure.
- <**td** th:switch="${student.gender}">
  <**span** th:case="'M'" th:text="Male" />
  <**span** th:case="'F'" th:text="Female" />
  </**td**>

Handling User Input

Form

- HTML Forms in Thymeleaf is used send POST data to on the backend side.
- Thymeleaf comes with several special attributes used for building and handling forms:
  - th:field - used for binding inputs with properties on the form-backing bean,
  - th:errors - attribute that holds form validation errors,
  - th:errorclass - CSS class that will be added to a form input if a specific field has validation errors,
  - th:object - an attribute that holds a command object (main form bean object) that is a form representation on the backend side.

Form - Command object

- Command object is a base bean object attached to the Form which contains all attributes related to input fields.
- This is the main POJO class with declared setter and getter methods. Command objects shouldn't contain any business logic.
- The following example shows how to use th:object attribute that holds a reference to the Command object:

```
<form th:action="@{/authorization}" th:object="${autObj}" method="post">
      Login: <input type="text" th:field="{autObj.login}" /> <br/>
       Password: <input type="text" th:field= “ *{pswd} " />
</form>
```

Inputs

- Thymeleaf provides a special attribute th:field responsible for binding input fields with a property in the bean class.
- This attribute behaves differently depending on whether it is attached to.
- Thymeleaf supports all-new input types introduced in HTML5 such as type="color" or type="datetime".
- In the following simple example HTML text input element was bind with property username:

```
<input type="text" th:field="*{username}" />
```

Individual fields

- Individual fields are mapped using the *th:field=”*{name}”* attribute, where the *name* is the matching property of the object.

Form example

```
<form action="#" th:action="@{/saveStudent}" th:object="${student}" method="post">
  <table border="1">

      <tr>
          <td><label>Id</label></td>
          <td><input type="number" th:field="*{id}" /></td>
      </tr>
      <tr>
          <td><label>Name</label></td>
          <td><input type="text" th:field="*{name}" /></td>
      </tr>
      <tr>
          <td><input type="submit" value="Submit" /></td>
      </tr>

  </table>
</form>
```

Handling User Input example

- @Controller **public class StudentController** {

  @RequestMapping(value = "/saveStudent", method = RequestMethod.POST)
  public String **saveStudent**(@ModelAttribute Student student) {
  // logic to process input data
  }
  }

Form - links

- https://frontbackend.com/thymeleaf/working-with-forms-in-thymeleaf
- https://www.thymeleaf.org/doc/tutorials/2.1/thymeleafspring.html#creating-a-form

CSS and JS

- **For CSS and JavaScript files, the default directory is *src/main/resources/static*.**
- Let's create *static/styles/style.css and static/js/main.js* folders for our CSS and JS files, respectively.

- <link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
- <script type="text/javascript" th:src="@{/js/main.js}"></script>

- // spring_boot_thymeleaf_02_inludes

ToDo Example

Link To The URL

- <a th:href="@{/list}">Task List

- @RequestMapping("/list")
  public ModelAndView userTempPage() {
  // ….
  }

Display Task List

- ```
  @RequestMapping("/list")
  public ModelAndView goToTaskListPage() {
      ModelAndView modelAndView = new ModelAndView();
      modelAndView.setViewName("task_list");
      modelAndView.addObject("taskList", taskDTOList);

      return modelAndView;
  }
  ```
- In task_list.html :

```
<tr th:each="task : ${taskList}">
  <td th:utext="${task.id}">...</td>
  <td  th:utext="${task.title}">...</a></td>
  <td th:utext="${task.content}">...</td>
  <td>
    <a th:href="@{/contacts/{taskid}/edit(taskid=${task.id})}">Edit</a></td>
</tr>
```
```

Generating link parameter

- <a th:href="@{/task/{taskid}/edit(taskid=${task.id})}">
  Go to Edit

Redirecting to another url

- ```
  @PostMapping(value = "/add")
  public String saveTask( …… ) {
          return "redirect:/list";
  }
  ```

- ```
  @RequestMapping("/list")
  public ModelAndView goToTaskListPage() {
   //….
  }
  ```

Url Redirection

- return "redirect:/list";
- **return new ModelAndView**("redirect:/redirectedUrl", model);
- **return new RedirectView**("redirectedUrl");

- Link:
- https://www.baeldung.com/spring-redirect-and-forward

Tutorial links

- https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html#fragment-specification-syntax

Spring Security + Thymeleaf

dasturlash.uz

Configuration 1

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

- @Override
  protected void configure(HttpSecurity http) throws Exception {

```
    http.csrf().disable()
            .authorizeRequests() ……
            .and()
            .formLogin().loginPage("/newLoginUrl").permitAll()
            .and()
            .exceptionHandling().accessDeniedHandler(accessDeniedHandler);
}
```

Configuration 2

- .formLogin() – we are going to customize login form
- *.loginPage("*/loginUrl*")*
  - when authentication is required, redirect the browser to /loginUrl
  - we are in charge of rendering the login page when /loginUrl is requested
  - when authentication attempt fails, redirect the browser to /loginUrl?error (since we have not specified otherwise)
  - we are in charge of rendering a failure page when /loginUrl?error is requested
  - when we successfully logout, redirect the browser to /loginUrl?logout (since we have not specified otherwise)
  - we are in charge of rendering a logout confirmation page when /loginUrl?logout is requested

Configuration 3

- *loginProcessingUrl()* – the URL to submit the username and password to
  - tells Spring Security to process the submitted credentials when sent the specified path and, by default, redirect user back to the page user came from.
  - *It will not pass the request to Spring MVC and your controller.*
- *defaultSuccessUrl()* – the landing page after a successful login
- *failureUrl()* – the landing page after an unsuccessful login
- *logoutUrl()* – the custom logout

Configuration 4. Additional configs

- ```
  @Override
  protected void configure(final HttpSecurity http) …{
  http ...
  .and()
  .formLogin()
          .loginPage("/login.html")
          .loginProcessingUrl("/perform_login")
          .defaultSuccessUrl("/homepage.html", true)
          .failureUrl("/login.html?error=true")
          .failureHandler(authenticationFailureHandler())
          .and()
          .logout()
          .logoutUrl("/perform_logout");
  }
  ```
  ```

Configuration – example 1

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http.csrf().disable()
            .authorizeRequests() ……
            .and()
            .formLogin().loginPage("/newLoginUrl")
           .loginProcessingUrl("/loginDo").permitAll()
            .and()
            .exceptionHandling().accessDeniedHandler(accessDeniedHandler);
}
```

Configuration – example 2

- ```
  @GetMapping({"/newLoginUrl“, "/403"})
  public String login(Model model) {
     return "login";
  }
  ```

- ```
  <form th:action="@{/loginDo}" method="post">
      <label><b>Username</b></label>
      <br>
      <input type="text" placeholder="Enter Username" name="username" required>
      <br>
      <label><b>Password</b></label>
      <br>
      <input type="password" placeholder="Enter Password" name="password" required>
      <br>
      <button type="submit">Login</button>
  </form>
  ```

AccessDeniedHandler - 1

- When user not have permission Spring Security throws an Exception.
- So we can handle this exception and redirect to login page.
- // handle 403 page

```
@Component
public class MyAccessDeniedHandler implements AccessDeniedHandler {
    @Override
    public void handle(HttpServletRequest httpServletRequest,
                       HttpServletResponse httpServletResponse,
                       AccessDeniedException e) throws IOException, ServletException {
        httpServletResponse.sendRedirect(httpServletRequest.getContextPath() + "/403");
    }
}
```

- So in case exception it redirects to “/403” url. We will handle it in controller.

AccessDeniedHandler - 2

- **In Configuration**
- ```
  @Override
      protected void configure(HttpSecurity http) throws Exception {
          http.csrf().disable()
                  ....
   .and()
   .exceptionHandling().accessDeniedHandler(accessDeniedHandler);

   }
  ```

- **In Controller**
- ```
  @GetMapping({"/login", "/newLoginUrl", "/403"})
  public String login(Model model) {
      model.addAttribute("authObj", new AuthDTO());
      return "login";
  }
  ```

Spring security + Custom Authentication

dasturlash.uz

Custom Authentication 1

- In Configuration
- @Override
  protected void configure(HttpSecurity http) throws Exception {

```
    http.csrf().disable()
            .authorizeRequests()
            .antMatchers("/admin", "/admin/*").hasRole("ADMIN")
            .anyRequest().permitAll()
            .and()
            .exceptionHandling().accessDeniedHandler(accessDeniedHandler);
}
```

Custom Authentication 2

- In configuration class we need to enabled authentication manager.

```
@Bean
public AuthenticationManager authenticationManagerBean() throws Exception
{
    return super.authenticationManagerBean();
}
```

- // full code in spring_boot_thymeleaf_04_authorization_02 project

Custom Authentication 3 - Controller

```
@PostMapping(value = "/aut")
public String saveTask(@ModelAttribute("authObj") AuthDTO authDTO, Model model) {
    Authentication authenticate = authenticationManager
            .authenticate(new UsernamePasswordAuthenticationToken(authDTO.getUsername(),
authDTO.getPswd()));

    UserDetails user = (UserDetails) authenticate.getPrincipal();
    Optional<SimpleGrantedAuthority> optional = (Optional<SimpleGrantedAuthority>)
user.getAuthorities().stream().findAny();

    String role = "";
    if (optional.isPresent()) {
        role = optional.get().getAuthority();
    }
    if(role.equals("USER")){
        return "redirect:user";
    }else if(role.equals("ADMIN")){
        return "redirect:admin";
    }

    model.addAttribute("authObj", new AuthDTO());
    return "login";
}
```

Fragments

Fragments

- **Thymeleaf Fragments used to reuse some common parts of a site.**
- **There are three basic ways to include content from a fragment:**
- ***insert*** - inserts content inside the tag
- ***replace*** - replaces the current tag with the tag defining the fragment
- ***include*** - this is deprecated but it may still appear in a legacy code

How to use fragment - 1

- Let us say you want to define a reusable footer component to add copyright information to all of your web pages, so you just create a footer.html file with the following content.
-
```
<!DOCTYPE html>
 <html lang="en" xmlns:th="http://www.thymeleaf.org">
      <body>
           <footer th:fragment="footer">
                <p>&copy; 2020 attacomsian.com</p>
           </footer>
      </body>
 </html>
```

- The above code defined a fragment called footer that you can easily include to your homepage by using one of the th:insert or th:replace attributes.

How to use fragment - 2

- <body>
  . . .

  . . .
  </body>

- // in spring_boot_thymeleaf_05_fragment_01 project

Reuse fragments - 1

- You can also **define more than one fragment in a single HTML document** as shown in the blow file called components.html:
-

```
<!DOCTYPE html>
 <html lang="en" xmlns:th="http://www.thymeleaf.org">
        <body>

            <header th:fragment="header">
            <h1>Welcome to My Blog</h1>
            </header>

            <nav th:fragment="menu">
              <a th:href="@{/}">Homepage</a>
              <a th:href="@{/about}">About Me</a>
              <a th:href="@{/blog}">Blog</a>
              <a th:href="@{/contact}">Contact</a>
            </nav>

            <footer th:fragment="footer">
              <p>&copy; 2020 attacomsian.com</p>
            </footer>

        </body>
</html>     // 3ta fragment bor. Ularni xoxlagan joyda ishlatsak bo'ladi.
           // in spring_boot_thymeleaf_05_fragment_01 project
```

How to use fragment - 3

- Additional information can be found in below links.
- https://attacomsian.com/blog/thymeleaf-fragments
- https://www.baeldung.com/spring-thymeleaf-fragments

Bootstrap

- https://frontbackend.com/thymeleaf/how-to-add-bootstrap-css-and-js-to-thymeleaf

How to add Bootstrap CSS and JS

- Add Bootstrap using WebJars
- In Maven projects we can simply add Boostrap webjar dependency in the POM.xml file like in following example:
- <dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>4.0.0-2</version>
</dependency>
- Bootstrap requires jQuery and popper.js library and luckily we don't have to add these frameworks separately because Bootstrap webjar includes these projects as dependencies.

Add Bootstrap using WebJars

```
<!DOCTYPE HTML>
  <html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>
       <meta charset="UTF-8"/>
       <title>Spring Boot Thymeleaf - Bootstrap WebJars</title>
       <link th:rel="stylesheet" th:href="@{/webjars/bootstrap/4.0.0-2/css/bootstrap.min.css} "/>
  </head>
<body>
<div class="container">
  <div class="row">
       <div class="col">...</div>
 </div>
</div>

<script th:src="@{/webjars/jquery/3.0.0/jquery.min.js}"></script>
<script th:src="@{/webjars/popper.js/1.12.9-1/umd/popper.min.js}"></script>
<script th:src="@{/webjars/bootstrap/4.0.0-2/js/bootstrap.min.js}"></script>
                                                          dasturlash.uz
  </body>
</html>
```

Adding Bootstrap library using static asset files

- Adding Bootstrap library using static asset files
-
```
<!DOCTYPE HTML>
  <html lang="en" xmlns:th="http://www.thymeleaf.org">
   <head>
   <meta charset="UTF-8"/>
    <title>Spring Boot Thymeleaf - Bootstrap Static Files</title>
    <link th:rel="stylesheet" th:href="@{/assets/bootstrap/css/bootstrap.min.css} "/>
</head>
<body>

<div class="container">
  <div class="row">
        <div class="col"> </div>
  </div>
</div>

 <script th:src="@{/assets/jquery/jquery.min.js}"></script>
 <script th:src="@{/assets/popper.js/popper.min.js}"></script>
 <script th:src="@{/assets/bootstrap/js/bootstrap.min.js}"></script>

</body>
</html>
```

Add Bootstrap using CDN

- Content Delivery Network is a distributed system whose main task is to provide content in the shortest possible time to which many users from different places have access.
- Bootstrap, jQuery, and many other popular libraries can be included in the website using CDN links.
- <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-9aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk" crossorigin="anonymous">
- <script src="https://code.jquery.com/jquery-3.5.1.min.js" integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0=" crossorigin="anonymous"></script>
- <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/2.4.1/cjs/popper.min.js" integrity="sha256-T3bYsIPyOLpEfeZOX4M7J59ZoDMzuYFUsPiSN3Xcc2M=" crossorigin="anonymous"></script>