**Contents**

[Model View Controller (MVC) 1](#_Toc386604198)

[Thymeleaf Tags within HTTP Requesting 3](#_Toc1625835961)

[Thymeleaf Forms 4](#_Toc689717912)

[Thymeleaf Attributes in HTML Pages 6](#_Toc958114019)

[SpringBoot Security Attributes/Tags 6](#_Toc1068704590)

[Using Sec:Authentication 7](#_Toc1193310751)

[How implementing: org.springframework.security.core.userdetails.UserDetails; library works: 7](#_Toc173172390)

[How to add a custom method: 8](#_Toc590147071)

# Model View Controller (MVC)

Model-View-Controller (MVC) is a design pattern used to separate concerns in a software application. The basic idea is to separate the application into three interconnected components: the model, the view, and the controller.

In Spring Boot, Thymeleaf can be used as a view technology to render HTML templates. Here's a brief explanation of how MVC works in Spring Boot with Thymeleaf:

**Model**: The model represents the data and the business logic of the application. In Spring Boot, the model is typically implemented using Java classes that are annotated with the @Entity and @Service annotations. These classes define the structure of the data and provide methods to interact with it.

**View**: The view is responsible for presenting the data to the user. In Spring Boot, Thymeleaf is often used as a view technology to render HTML templates. Thymeleaf templates are typically located in the src/main/resources/templates directory and are used to generate HTML pages.

**Controller**: The controller is responsible for handling user requests and returning the appropriate view. In Spring Boot, controllers are implemented as Java classes that are annotated with the @Controller annotation. These classes define methods that handle HTTP requests and return the appropriate view.

**Here's an example of how MVC works in Spring Boot with Thymeleaf:**

**Model**: Let's say we have a User class that represents a user in our application. We would define this class with the @Entity annotation:

@Entity

public class User {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String username;

private String password;

// getters and setters

}

We might also have a UserService class that provides methods to interact with users:

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

public User getUserById(Long id) {

return userRepository.findById(id).orElse(null);

}

// other methods to interact with users

}

**View**: We would define a Thymeleaf template to display user information:

<!DOCTYPE html>

<html xmlns:th="http://www.thymeleaf.org">

<head>

<title>User Information</title>

</head>

<body>

<h1>User Information</h1>

<p th:text="${user.username}"></p>

<p th:text="${user.password}"></p>

</body>

</html>

**Controller**: We would define a controller class to handle requests for user information and return the appropriate view:

@Controller

public class UserController {

@Autowired

private UserService userService;

@GetMapping("/users/{id}")

public String getUser(@PathVariable Long id, Model model) {

User user = userService.getUserById(id);

model.addAttribute("user", user);

return "user";

}

}

In this example, the getUser() method handles GET requests for user information at the URL /users/{id}. The method calls the getUserById() method of the UserService to retrieve the user with the specified ID. It then adds the user to the model using the addAttribute() method and returns the name of the Thymeleaf template to be rendered ("user" in this case). The Thymeleaf template uses the th:text attribute to display the user's username and password.

# Thymeleaf Tags within HTTP Requesting

In Spring Boot with Thymeleaf, **@PostMapping**, **@RequestMapping**, and **@GetMapping** are annotations used to handle HTTP requests.

|  |
| --- |
| 1. **@PostMapping** is used to map HTTP POST requests onto specific handler methods. It is used to handle form submissions and create new resources. |
| Example: |
| typescriptCopy code |
| @PostMapping("/employees") public String addEmployee(Employee employee) {  //code to add employee to database  return "redirect:/employees";  } |

}

|  |
| --- |
| 1. **@RequestMapping** is a general-purpose annotation used to map HTTP requests onto specific handler methods. It can be used to handle any HTTP method (GET, POST, PUT, DELETE, etc.). |
| Example: |
| typescriptCopy code |
| @RequestMapping("/employees") public String listEmployees(Model model) {  //code to fetch employees from database  model.addAttribute("employees", employees);  return "employee-list";  } |

|  |
| --- |
| 1. **@GetMapping** is a specific type of **@RequestMapping** annotation used to handle HTTP GET requests. |
| Example: |
| lessCopy code |
| @GetMapping("/employees/{id}") public String showEmployee(@PathVariable("id") Long id, Model model) {  //code to fetch employee by id from database  model.addAttribute("employee", employee); |
| return "employee-details"; } |

In summary, **@PostMapping** is used for handling POST requests, **@GetMapping** is used for handling GET requests, and **@RequestMapping** is a general-purpose annotation used to handle any type of HTTP request.

# Thymeleaf Forms

In Spring Boot Thymeleaf, th:action and th:object are two attributes used to bind a form to a Spring MVC controller.

th:action is used to specify the URL that the form data will be submitted to. It is usually set to the URL mapping of the controller method that will handle the form submission. For example, th:action="@{/my-form}" will submit the form data to the /my-form URL.

th:object is used to bind the form data to a model object in the Spring MVC controller. It is usually set to the name of the model attribute that the form data will be bound to. For example, th:object="${myModel}" will bind the form data to the myModel attribute in the controller.

Here is an example of how to use th:action and th:object in a Thymeleaf template:

<form th:action="@{/my-form}" th:object="${myModel}" method="post">

<input type="text" th:field="\*{name}" />

<input type="email" th:field="\*{email}" />

<button type="submit">Submit</button>

</form>

In this example, the form data will be submitted to the /my-form URL and bound to the myModel attribute in the controller. The th:field attribute is used to bind each input field to a corresponding property of the myModel object.

**Inside of the Controller:**

@Controller

public class MyController {

@GetMapping("/my-form")

public String myForm(Model model) {

model.addAttribute("myModel", new MyModel());

return "my-form";

}

@PostMapping("/my-form")

public String myFormSubmit(@ModelAttribute("myModel") MyModel myModel) {

// Do something with myModel

return "redirect:/my-success-page";

}

}

In this example, the myForm method handles the GET request for the /my-form URL and adds a new MyModel object to the model with the name "myModel". The my-form template can then reference this object using th:object="${myModel}".

The myFormSubmit method handles the POST request for the /my-form URL and takes a MyModel object as a parameter annotated with @ModelAttribute("myModel"). This tells Spring MVC to bind the form data to the myModel object and pass it as a parameter to the method.

Once the form data has been processed, the method returns a redirect to a success page, in this case /my-success-page.

Of course, you'll need to replace MyModel with the name of your own model class and update the URLs to match your own application.

# Thymeleaf Attributes in HTML Pages

1. th:text - sets the text content of an element
2. th:utext - sets the unescaped text content of an element
3. th:each - iterates over a collection and generates HTML for each element
4. th:if - conditionally includes or excludes HTML based on an expression
5. th:href - sets the URL for a link or form action
6. th:src - sets the source URL for an image or other media element
7. th:class - sets the CSS classes for an element
8. th:style - sets the inline CSS style for an element
9. th:attr - sets any attribute for an element
10. th:replace - replaces the content of an element with another template fragment

# SpringBoot Security Attributes/Tags

1. <sec:authorize> - this tag is used to authorize access to a specific block of content based on the user's role or authentication status.
2. <sec:authentication> - this tag is used to access information about the currently authenticated user, such as their username or authorities.
3. <sec:csrfInput> - this tag is used to generate a hidden input field containing a CSRF token, which is used to protect against cross-site request forgery attacks.
4. <sec:csrfMetaTags> - this tag is used to generate meta tags containing CSRF tokens, which can be used by JavaScript code to protect against CSRF attacks.
5. <sec:logout> - this tag is used to generate a logout button or link that logs the user out of the application.
6. <sec:form-login> - this tag is used to generate a login form for users to authenticate themselves.
7. <sec:http> - this tag is used to configure security for a specific HTTP request pattern or set of URL patterns.
8. <sec:intercept-url> - this tag is used to configure security for a specific URL pattern, including specifying the required roles or authentication status.
9. <sec:remember-me> - this tag is used to generate a "remember me" checkbox on the login form, allowing users to stay authenticated even after closing their browser.
10. <sec:accessDeniedHandler> - this tag is used to specify a custom access denied handler, which is called when a user attempts to access a protected resource without sufficient privileges.

# Using Sec:Authentication

<!-- import the Spring Security tag library -->

<html xmlns:sec="http://www.springframework.org/security/tags">

<!-- display the user's username -->  
<**h1** th:inline=*"text"*>Hello <**span** sec:authentication=*"name"*>Username</**span**>!</**h1**>  
OR   
<**h1** th:inline=*"text"*>Hello <**span** th:text=*"${#authentication.getPrincipal().getUsername}"*>User</**span**>!</**h1**>  
<!-- display the user's firstName -->  
<**h1** th:inline=*"text"* Hello Welcome ><**span** th:text=*"${#authentication.getPrincipal().getFirstName}"*>User</**span**>!</**h1**>

In this example, we're using the property attribute to specify that we want to access the fullName property of the principal object, which represents the currently authenticated user.

Assuming that you have a User class with a fullName property and that your Spring Security configuration is correctly set up to authenticate users, this should display the user's full name on the page.­

Note that the principal object returned by the <sec:authentication> tag is an instance of the UserDetails interface, which is typically implemented by a custom UserDetailsService in Spring Security. If you need to access additional properties of the authenticated user, you can add them to your UserDetails implementation and access them using the property attribute of the <sec:authentication> tag.

Breakdown on how it works:

The Thymeleaf expression ${#authentication.getPrincipal().getUsername()} is used to display the username of the currently authenticated user in a Thymeleaf template.

Here's how it works:

1. The #authentication expression is a Thymeleaf Security utility object that provides access to information about the currently authenticated user. This object is automatically added to the context by the Spring Security integration for Thymeleaf.
2. The getPrincipal() method returns the principal object that represents the currently authenticated user. In Spring Security, the principal object is typically an instance of UserDetails, which provides information about the user, including their username.
3. The getUsername() method returns the username of the authenticated user as a string.
4. Finally, the th:text attribute is used to set the text content of the HTML element to the value of the expression, which is the username of the authenticated user.

So, putting it all together, the expression ${#authentication.getPrincipal().getUsername()} retrieves the username of the currently authenticated user and displays it as text in the HTML element.

# How implementing: org.springframework.security.core.userdetails.UserDetails; library works:

The UserDetails interface is part of the Spring Security framework and is used to represent a user's details, such as their username, password, and authorities. It is an interface that provides methods for retrieving information about the user, including their username, password, and authorities.

Here are the main methods defined in the UserDetails interface:

1. getUsername(): Returns the username of the user.
2. getPassword(): Returns the password of the user.
3. getAuthorities(): Returns a collection of the authorities granted to the user.
4. isEnabled(): Indicates whether the user account is enabled.
5. isAccountNonExpired(): Indicates whether the user account has expired.
6. isAccountNonLocked(): Indicates whether the user account is locked.
7. isCredentialsNonExpired(): Indicates whether the user's credentials (password) have expired.

Implementations of the UserDetails interface are typically provided by the application developer and are used by Spring Security to authenticate users and authorize access to protected resources. The UserDetails interface can be implemented by a custom UserDetailsService, which is responsible for retrieving user details from a database or other data source.

Overall, the UserDetails interface is a key component of the Spring Security framework, providing a standard way to represent user details and authenticate users in a secure and reliable way.

## How to add a custom method:

To create a custom method to retrieve data about a user using the UserDetails interface from the org.springframework.security.core.userdetails library, you would need to create a class that implements the UserDetails interface and define your custom method within that class.

First you will make sure a method and available reference are first implemented in the selected model as seen in the following USER Model:

|  |
| --- |
| package edu.group3.EmployeeManagement.models;  import jakarta.persistence.CascadeType;  import jakarta.persistence.Entity;  import jakarta.persistence.FetchType;  import jakarta.persistence.GeneratedValue;  import jakarta.persistence.GenerationType;  import jakarta.persistence.Id;  import jakarta.persistence.JoinTable;  import jakarta.persistence.ManyToMany;  import jakarta.persistence.OneToOne;  import jakarta.persistence.Table;  import jakarta.persistence.JoinColumn;  import java.util.ArrayList;  import java.util.List;  import com.fasterxml.jackson.annotation.JsonIdentityInfo;  import com.fasterxml.jackson.annotation.ObjectIdGenerators;  import lombok.AllArgsConstructor;  import lombok.Data;  import lombok.NoArgsConstructor; |
| @Entity  @Data  @NoArgsConstructor  @AllArgsConstructor  public class User {    @GeneratedValue(strategy = GenerationType.IDENTITY)  @Id  private Integer id;    private String firstName;    private String lastName;    private String username;    private String password;  @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)  @JoinTable(  name="user\_role",  joinColumns={@JoinColumn(name="USER\_ID", referencedColumnName="ID")},  inverseJoinColumns={@JoinColumn(name="ROLE\_ID", referencedColumnName="ID")})    private List<Role> roles = new ArrayList<>();    public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public String getFirstName() {  return firstName;  }  public void setFirstName(String firstName) {  this.firstName = firstName;  }  public String getLastName() {  return lastName;  }  public void setLastName(String lastName) {  this.lastName = lastName;  }    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;  }  public List<Role> getRoles() {  return roles;  }  public void setRoles(List<Role> roles) {  this.roles = roles;  }  public void addRole(Role roleUser) {  this.roles.add(roleUser);    }  } |

Then using a defined class that refers to your User model and will implement UserDetails as the shown in the following UserPrincipal class:

|  |
| --- |
| package edu.group3.EmployeeManagement.models;  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;  }    public String getFirstName() {  return this.user.getFirstName();  }  } |

Within this class reference a method from the USER Model, **getFirstName()**. This allows you to locate the signed in USER’s first name.

Next you want to make sure you have a custom class that will implement UserDetailService, that allows authorization of the logged in user while also loading in needed information seen in the following class:

|  |
| --- |
| package edu.group3.EmployeeManagement.service;  import java.util.ArrayList;  import java.util.Collection;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.security.core.GrantedAuthority;  import org.springframework.security.core.authority.AuthorityUtils;  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 edu.group3.EmployeeManagement.models.Role;  import edu.group3.EmployeeManagement.models.User;  import edu.group3.EmployeeManagement.models.UserPrincipal;  import edu.group3.EmployeeManagement.repository.RoleRepository;  import edu.group3.EmployeeManagement.repository.UserRepository; |
| @Service  public class MyUserDetailsService implements UserDetailsService {    @Autowired  private UserRepository userRepository;    @Autowired  private RoleRepository roleRepository;  @Override  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  User user = userRepository.findByUsername(username);  if (user == null) {  throw new UsernameNotFoundException("User not found!");  }  return new UserPrincipal(user);  }  private static Collection<? extends GrantedAuthority> getAuthorities(User user) {  String[] userRoles = user.getRoles().stream().map((role) -> role.getName()).toArray(String[]::new);  Collection<GrantedAuthority> authorities = AuthorityUtils.createAuthorityList(userRoles);  return authorities;  }    public void registerDefaultUser(User user)  {  Role roleUser = roleRepository.findByName("User");  user.addRole(roleUser);    userRepository.save(user);  }  } |

Finally to setup your HTML page where you reqirue that custom data information will require the following HTML tags:

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"

xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity6">

And then the following code is the set up of the page:

|  |
| --- |
| <!Doctype html>  <html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="https://www.thymeleaf.org"  xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-springsecurity6">  <head>  <title>Dashboard</title>  <meta charset="utf-8">  <meta name="viewport" content="wdith=device-width, initial-scale=1, shrink-to-fit=no">  <link href="css/dashboard.css" type="text/css" rel="stylesheet">  </head>  <header>  <a th:href="@{/}">GAZEBO HQ</a>  </header>  <body>  <div id="PageHeader">  <p>Dashboard</p>  <h1 th:inline="text"> Hello Welcome <span th:text="${#authentication.getPrincipal().getFirstName}">User</span>!</h1>  </div>  <div class="dropdown">  <button class="dropbtn">Menu</button>  <div class="dropdown-content">  <a href="#">Dashboard</a>  <a th:href="@{/timesheet}">Time Sheet</a>  <a th:href="@{/manageusers}">Manage Users</a>  <a th:href="@{/inbox}">Inbox</a>  <a th:href="@{/fileserver}">File Server</a>  <a th:href="@{/systemlog}">System Log</a>  <a th:href="@{/payrollAdmin}">Payroll Admin</a>  <a th:href="@{/payrollUser}">Payroll User</a>  <!-- Will be used later to add in Roles -->  <!-- <a sec:authorize="hasRole('ADMIN')" th:href="@{/payrollAdmin}">Payroll</a>  <a sec:authorize="hasRole('USER')" th:href="@{/payrollUser}">Payroll</a> -->  <a th:href="@{/accountsettings}">Account Settings</a>  </div>  </div>  <div id="TodaysDate", style="float:right;">  <p>Today is:</p>  <p id="date"></p>  <script>  n = new Date();  y = n.getFullYear();  m = n.getMonth() + 1;  d = n.getDate();  document.getElementById("date").innerHTML = m + "/" + d + "/" + y;  </script>  </div>  <div id="ClockedIn">  <p> Clocked In: (N/A Yet)</p>  </div>  <div id="HoursThisWeek">  <p> Hours This Week: (N/A Yet)</p>  </div>    <div id="Login Out">  <form th:action="@{/logout}" method="post">  <input type="submit" value="Sign Out"/>  </form>  </div>    </body>  </html> |

The Tag:

**<h1 th:inline="text"> Hello Welcome <span th:text="${#authentication.getPrincipal().getFirstName}">User</span>!</h1>**

# Apache POI library for reading and writing Microsoft Office binary and OOXML file formats.