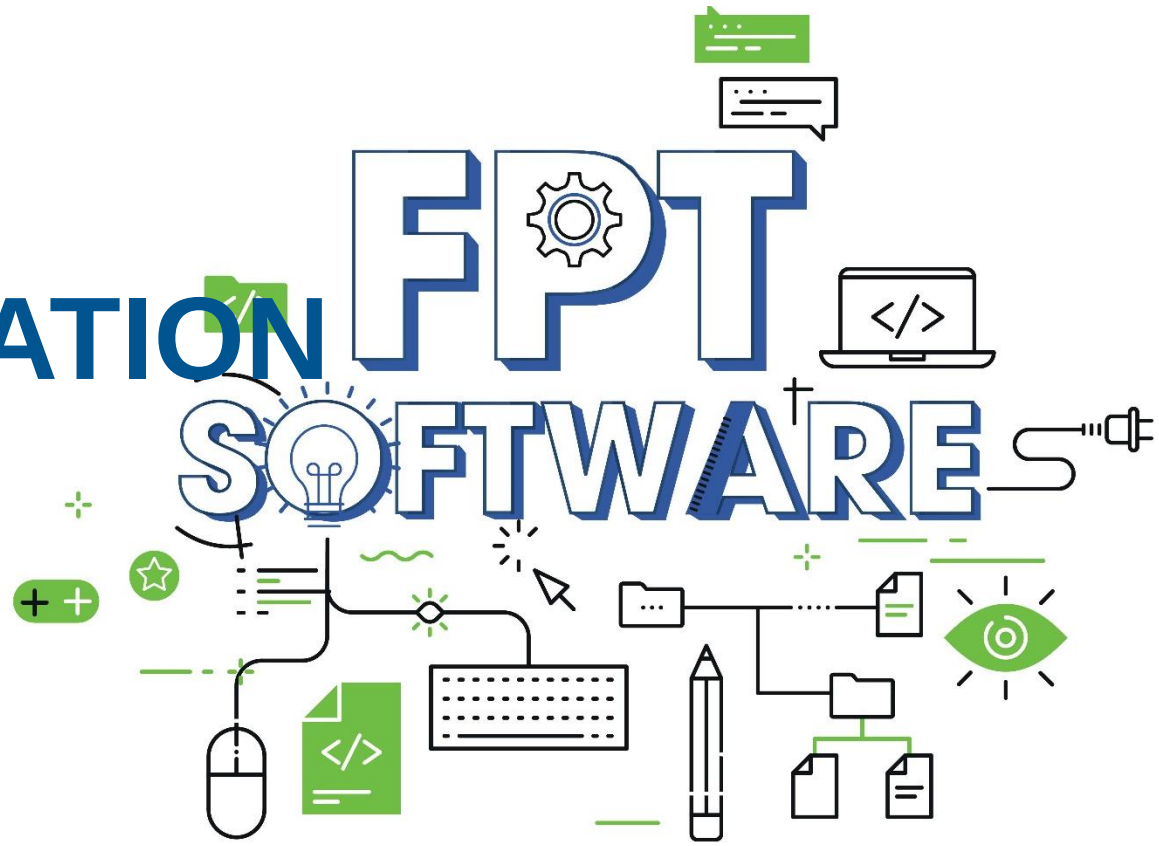
HIBERNATE AND SPRING MVC INTEGRATION with THYMELEAF



Design by: DieuNT1

Agenda

1. • @RequestParam, @PathVariable, @ModelAttribute
2. • Model, ModelMap and ModelAndView classes
3. • Spring Web MVC and Hibernate ORM
4. • Spring Framework Thymeleaf Tags Introduction
5. • Build a simple Web App (CRUD)

Learning Goals

- **After the course, attendees will be able to:**

Understand Spring Web MVC Framework and Hibernate ORM Integration.
Know how to write a Web application.

Section 1

@RequestParam and @ModelAttribute

A Simple Mapping

- We can use *@RequestParam* to extract query parameters, form parameters, and even files from the request.
- In this example, we used *@RequestParam* to extract the *id* query parameter.

```
@GetMapping("/api/foos")
public String getFoos(@RequestParam String id) {
        return "ID: " + id;
}
```

- A simple GET request would invoke *getFoos*:

```
http://localhost:8080/api/foos?id=abc
----ID: abc
```

Specifying the Request Parameter Name

- **Sometimes we want these to be different, though.**
- Fortunately, **we can configure the *@RequestParam* name using the *name* attribute**:

```
@PostMapping("/api/foos")
public String addFoo( @RequestParam(name = "id") String fooId,
                              @RequestParam String name) {
        return "ID: " + fooId + " Name: " + name;
}
```

- We can also do *@RequestParam(value = "id")* or just *@RequestParam("id")*.

Optional Request Parameters

- Method parameters annotated with *@RequestParam* are required by default.
- This means that if the parameter isn't present in the request, we'll get an error:

```
GET /api/foos HTTP/1.1
-----
400 Bad Request
Required String parameter 'id' is not present
```

- **We can configure our *@RequestParam* to be optional, though, with the *required* attribute:**

```
@GetMapping("/api/foos")
public String getFoos(@RequestParam(required = false) String id) {
        return "ID: " + id;
}
```

Optional Request Parameters

- **In this case, both:**

```
http://localhost:8080/api/foos?id=abc
----ID: abc
And
http://localhost:8080/api/foos
----ID: null
```

- *will correctly invoke the method.*
- When the parameter isn't specified, the method parameter is bound to null.

Using Java 8 *Optional*

- Alternatively, we can wrap the parameter in **Optional**:

```
@GetMapping("/api/foos")
public String getFoos(@RequestParam Optional<String> id) {
        return "ID: " + id.orElseGet(() -> "not provided");
}
```

- In this case, **we don't need to specify the *required* attribute.**
- And the default value will be used if the request parameter is not provided:

```
http://localhost:8080/api/foos
---- ID: not provided
```

Mapping a Multi-Value Parameter

- A single *@RequestParam* can have multiple values:

```
@GetMapping("/api/foos")
public String getFoos(@RequestParam List<String> id) {
        return "IDs are " + id;
}
```

- **And Spring MVC will map a comma-delimited *id* parameter:**

```
http://localhost:8080/api/foos?id=1,2,3
----IDs are [1,2,3]
```

- **or a list of separate *id* parameters**:

```
http://localhost:8080/api/foos?id=1&id=2
----IDs are [1,2]
```

@ModelAttribute Annotation

- The *@ModelAttribute* is an annotation that binds a **method parameter or method return value** to a named model attribute and then exposes it to a web view.
- **Method Argument:**
  - ✓ When used as a method argument, it indicates the argument should be retrieved from the model: **the arguments fields should be populated from all request parameters that have matching names**.
  - ✓ **Example:** In the code snippet that follows the *employee* model attribute is populated with data from a form submitted to the *saveEmployee* endpoint.

```
@RequestMapping(value = "/saveEmployee", method = RequestMethod.POST)
   public String submit(@ModelAttribute("employee") Employees employee) {
       // Code that uses the employee object

       return "employeeView";
   }
```

Form Example

- **The View:** Let's first create a simple form with id and name fields

```
<form method="POST" action="/spring-mvc-basics/addEmployee">
        <label>Name</label><input name="name" />
        <label >Id</label> <input name="id" />

        <input type="submit" value="Submit" />
</form>
```

- **The Model:**

```
public class Employee {
        private long id;
        private String name;

        public Employee(long id, String name) {
                this.id = id;
                this.name = name;
        }
        // standard getters and setters removed
}
```

@ModelAttribute Annotation

- **Method Level:**
  - ✓When the annotation is used at the method level it indicates the **purpose of that method is to add one or more model attributes**.

```
@ModelAttribute
    public void addAttributes(Model model) {
        model.addAttribute("msg",

        "Welcome to the Netherlands!");
    }
```

  - ✓Spring-MVC will always make a *call first to that method, before it calls any request handler methods*.
  - ✓That is, **@ModelAttribute** methods are invoked before the controller methods annotated with **@RequestMapping** are invoked.

Section 2

Spring @Pathvariable Annotation

A Simple Mapping

- **The *@PathVariable* annotation can be used to handle template variables in the request URI mapping**, and use them as method parameters.
- A simple use case of the *@PathVariable* annotation would be an endpoint that identifies an entity with a primary key:

```
@GetMapping("/api/employees/{id}")
@ResponseBody
public String getEmployeesById(@PathVariable String id) {
        return "ID: " + id;
}
```

- A simple *GET request* to */api/employees/{id}* will invoke ***getEmployeesById*** with the extracted id value:

```
http://localhost:8080/api/employees/111
---- ID: 111
```

Specifying the Path Variable Name

- If the path variable name is different, we can specify it in the argument of the *@PathVariable* annotation:

```
@GetMapping("/api/employeeswithvariable/{id}")
@ResponseBody
public String getEmployeesByIdWithVariableName(@PathVariable("id")

                                String employeeId) {
        return "ID: " + employeeId;
}
```

- We can also define the path variable name as *@PathVariable(value="id")* instead of *PathVariable("id")* for clarity.

Multiple Path Variables in a Single Request

- Depending on the use case, **we can have more than one path variable in our request URI for a controller method, which also has multiple method parameters**:

```
@GetMapping("/api/employees/{id}/{name}")
@ResponseBody
public String getEmployeesByIdAndName(@PathVariable String id,

                    @PathVariable String name) {
        return "ID: " + id + ", name: " + name;
}
```

Multiple Path Variables in a Single Request

- We can also handle more than one *@PathVariable* parameters using a method parameter of type *java.util.Map<String, String>*:

```
@GetMapping("/api/employeeswithmapvariable/{id}/{name}")
@ResponseBody
public String getEmployeesByIdAndNameWithMapVariable(
                                        @PathVariable Map<String,
String> pathVarsMap) {
        String id = pathVarsMap.get("id");
        String name = pathVarsMap.get("name");
        if (id != null && name != null) {
            return "ID: " + id + ", name: " + name;
        } else {
            return "Missing Parameters";
        }
}
```

- Example:

```
http://localhost:8080/api/employees/1/bar
---- ID: 1, name: bar
```

@PathVariable as Not Required

- Since method parameters annotated by *@PathVariables* are mandatory by default, it doesn't handle the requests sent to */api/employeeswithrequired* path:
- You can set required = false:

```
@GetMapping(value = {    "/api/employeeswithrequiredfalse",
                         "/api/employeeswithrequiredfalse/{id}" })
@ResponseBody
public String getEmployeesByIdWithRequiredFalse( @PathVariable(required = false) String id) {
        if (id != null) {
            return "ID: " + id;
        } else {
            return "ID missing";
        }
}
```

```
http://localhost:8080/api/employeeswithrequiredfalse
---- ID missing
```

Section 3

Model, ModelMap and ModelAndView

Model class

- The model works a container that contains the data of the application. A data can be in any form such as ***objects***, ***strings***, ***information from the database***, etc.
- The model can supply attributes used for rendering views.
- To provide a view with usable data, we simply add this data to its *Model* object.
- Maps with attributes can be merged with *Model* instances:

```
@GetMapping("/showViewPage")
public String passParametersWithModel(Model model) {
    Map<String, String> map = new HashMap<>();
    map.put("spring", "mvc");
    model.addAttribute("message", “Welcome to Spring framework");
    model.mergeAttributes(map);
    return "viewPage";
}
```

ModelMap class

- Just like the *Model* interface above, *ModelMap* is also used to pass values to render a view.
- The advantage of *ModelMap* is it gives us the ability to pass a collection of values and treat these values as if they were within a *Map*.
- ModelMap class subclasses **LinkedHashMap**. It add some methods for convenience.
- **ModelMap** uses as generics and checks for null values.

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

  return "hello";
}
```

ModelMap class

- **Example:**

```
@Controller
public class EmployeeController {

private Map<Long, Employee> employeeMap = new HashMap<>();

@RequestMapping(value = "/addEmployee", method = RequestMethod.POST)
public String submit( @ModelAttribute("employee") Employee employee, BindingResult result, ModelMap model) {
    if (result.hasErrors()) {
        return "error";
    }

    model.addAttribute("name", employee.getName());
    model.addAttribute("id", employee.getId());

    employeeMap.put(employee.getId(), employee);

    return "employeeView";
}

@ModelAttribute
public void addAttributes(Model model) {
    model.addAttribute("msg", "Welcome to the Netherlands!");
}
}
```

ModelMap class

- **Results View:** Now let's print what we received from the form

```
<h3>${msg}</h3>
Name : ${name}
ID : ${id}
```

ModelAndView

- This interface allows us to pass all the information required by Spring MVC in one return.

```
@GetMapping("/goToViewPage")
public ModelAndView passParametersWithModelAndView() {
    ModelAndView modelAndView = new ModelAndView("viewPage");
    modelAndView.addObject("message", "Welcome to FA");
    return modelAndView;
}
```

Section 4

@Component vs @Repository and @Service

Introduction

- In most typical applications, we have distinct layers like data access, *presentation*, *service*, *business*, etc.
- In each layer, we have various beans. Simply put, **to detect them automatically.** Spring uses classpath scanning annotations. Then, it registers each bean in the **ApplicationContext**.
  - ✓`@Component:` is a generic stereotype for any Spring-managed component.
  - ✓`@Service:` annotates classes at the service layer
  - ✓`@Repository:` annotates classes at the persistence layer, which will act as a *database repository*.

Examples:

- @Service:

```
public interface UserService {
    User login(User user) throws Exception;
}
@Service("userService")
public class UserServiceImpl implements UserService {
        // TODO Auto-generated method stub
}
```

- @Repository

```
public interface UserDao {
    User login(User user) throws Exception;
}

@Repository("userDao")
@Transactional
public class UserDaoImpl implements UserDao {

  @Autowired
  private SessionFactory sessionFactory;
}
```

Section 5

HIBERNATE AND SPRING MVC INTEGRATION

Add dependencies

```
<properties>
     <spring.version>6.0.10</spring.version>
</properties>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-orm -->
<dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-orm</artifactId>
     <version>${spring.version}</version>
</dependency>

<dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-tx</artifactId>
     <version>${spring.version}</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.hibernate.orm/hibernate-core -->
<dependency>
     <groupId>org.hibernate.orm</groupId>
     <artifactId>hibernate-core</artifactId>
     <version>6.2.6.Final</version>
</dependency>
```

Create a DataSource Bean: XML Config

- Spring provides **many ways** to establish connection **to a database** and **perform operations** such as *retrieval of records*, *insertion of new records* and *updating / deletion* of existing records.
- The most basic of them is using **DriverManagerDataSource**.

```
<!-- DataSourse -->
<bean id="dataSource"
                class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
                value="com.microsoft.sqlserver.jdbc.SQLServerDriver" />
        <property name="url"
                value="jdbc:sqlserver://localhost:1433;databaseName=HumanResourceDB" />
        <property name="username" value="sa" />
        <property name="password" value="12345678" />
</bean>
```

Create a SessionFactory Bean: XML Config

- For using Hibernate 5 or 6 with Spring, we have to use:

  org.springframework.orm.hibernate5.LocalSessionFactoryBean.

```
<bean id="sessionFactory"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource" /> <!-- Dependency Injection -->
        <property name="packagesToScan" value="fa.training.entities" />
        <property name="hibernateProperties">
        <props>
                <prop key="hibernate.hbm2ddl.auto">none</prop>
                <prop key="hibernate.dialect">
                                org.hibernate.dialect.SQLServer2012Dialect</prop>
                <prop key="hibernate.show_sql">true</prop>
        </props>
        </property>
</bean>
```

Transaction Management

- **Enable the transaction support:**

```
<tx:annotation-driven transaction-manager="transactionManager" />

<bean id="transactionManager"
        class="org.springframework.orm.hibernate5.HibernateTransactionManager">

        <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Create DataSource and SessionFactory: Java Configuration

- Create **HibernateConfig** class:

```
@Configuration
@EnableTransactionManagement
@ComponentScans(value = @ComponentScan("fa.training"))
public class HibernateConfig {

@Bean
public LocalSessionFactoryBean sessionFactory() {
     LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
     sessionFactory.setDataSource(dataSource());
     sessionFactory.setPackagesToScan("fa.training.entities");
     sessionFactory.setHibernateProperties(hibernateProperties());
     return sessionFactory;
}
@Bean
public DataSource dataSource() {
     BasicDataSource dataSource = new BasicDataSource();
     dataSource.setDriverClassName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
     dataSource.setUrl("jdbc:sqlserver://localhost:1433;databaseName=database_name;encrypt=false");
     dataSource.setUsername("sa");
     dataSource.setPassword("12345678");
     return dataSource;
}
```

Create DataSource and SessionFactory: Java Configuration

- Create **HibernateConfig** class:

```
@Bean
public PlatformTransactionManager hibernateTransactionManager() {
     HibernateTransactionManager transactionManager = new HibernateTransactionManager();
     transactionManager.setSessionFactory(sessionFactory().getObject());
          return transactionManager;
}

private final Properties hibernateProperties() {
     Properties hibernateProperties = new Properties();
     hibernateProperties.setProperty("hibernate.hbm2ddl.auto", "update");
     hibernateProperties.setProperty("hibernate.dialect", "org.hibernate.dialect.SQLServer2012Dialect");
     return hibernateProperties;
}

}
```

Config DispatcherServlet: Java Configuration

- Update **WebInitializer** class:

```
public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { HibernateConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { WebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

@Transactional

- With Spring **@Transactional**, the above code gets reduced to simply this:

```
@Transactional
public void businessLogic() {
      ... use entity manager inside a transaction ...
}
```

- By using **@Transactional**, many important aspects such as ***transaction propagation are handled automatically***.
- In this case if another transactional method is called by **businessLogic**(), that method will have the option of joining the ongoing transaction.

@Transactional: DAO classes

- **Create DAO class:**

```
public interface UserDao {
    void save(User user);
    List<User> list();
}

@Repository
@Transactional
public class UserDaoImp implements UserDao {

    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public void save(User user) {
        sessionFactory.getCurrentSession().persist(user);

    }

    @Override
    public List<User> list() {
        SelectionQuery<User> query = sessionFactory.getCurrentSession()
                                                   .createQuery("FROM User", User.class);
        return query.list();
    }
}
```

Section 6

Integrating Thymeleaf With Spring

Overview

- Thymeleaf is a Java template engine for processing and creating HTML, XML, JavaScript, CSS and text.
- The library is **extremely extensible**, and its natural templating capability ensures we can prototype templates without a back end.
- This makes development very fast when compared with other popular template engines such as JSP.

https://www.thymeleaf.org/doc/tutorials/3.1/thymeleafspring.html

Overview

Thymeleaf is a modern server-side Java template engine for both web and standalone environments.

- Thymeleaf's main goal is to bring elegant *natural templates* to your development workflow — HTML that can be correctly displayed in browsers and also **work as static prototypes**, allowing for stronger collaboration in development teams.
- With modules for Spring Framework: a **host of integrations** with your favourite tools, and the **ability to plug** in your own functionality
- Thymeleaf is ideal for modern-day HTML5 JVM web development — although there is much more it can do.

Overview

Natural templates

- HTML templates written in Thymeleaf still look and work like HTML, letting the actual templates that are run in your application keep working as useful design artifacts.

```
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod: ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price, 1, 2)}">0.99</td>
    </tr>
  </tbody>
</table>
```

Simple Attributes

- **Syntax:**

```
th:text="${attributename}"
```

- **Example:**

```
model.addAttribute("serverTime", dateFormat.format(new Date()));
```

And here's the HTML code to display the value of *serverTime* attribute:

```
Current time is <span th:text="${serverTime}" />
```

Collection Attributes

- If the model attribute is a collection of objects, we can use the ***th:each*** tag attribute to iterate over it.
- **Example:**

```
public class Student implements Serializable {
        private Integer id;
        private String name;
        private Character gender;
        // standard getters and setters
}
```

Now we will add a list of students as model attribute in the controller class:

```
List<Student> students = new ArrayList<Student>();
// logic to build student data
model.addAttribute("students", students);
```

- **Thymeleaf template code** to iterate over the list of students and display all field values:

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

Collection Attributes

Conditional Evaluation

- ***if* and *unless***
  - ✓ We use the ***th:if="${condition}"*** attribute to display a section of the view if the condition **is met.**
  - ✓ And we use the ***th:unless="${condition}"*** attribute to display a section of the view if the condition **is not met**.
- **Example:**
  - ✓ The *gender* field has two possible values (M or F) to indicate the student's gender.
  - ✓ If we wish to display the words "Male" or "Female" instead of the single character, we could do this using this Thymeleaf code:

```
<td>
        <span th:if="${student.gender} == 'M'" th:text="Male" />
        <span th:unless="${student.gender} == 'M'" th:text="Female" />
</td>
```

Conditional Evaluation

- ***switch* and *case***
  - ✓ We use the ***th:switch*** and ***th:case*** attributes 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

- ***th:action*** attribute to provide the form action URL
- ***th:object*** attribute to specify an object to which the submitted form data will be bound
- ***th:field="*{name}"*** attribute to specify individual fields are mapped, *where the name is the matching property of the object*.
- **For the *Student* class, we can create an input form:**

```
<form action="#" th:action="@{/saveStudent}" th:object="${student}" method="post">
        <table border="1">
        <tr>
                <td><label th:text="#{msg.id}" /></td> <td>
                <input type="number" th:field="*{id}" /></td>
        </tr>
        <tr>
                <td><label th:text="#{msg.name}" /></td>
                <td><input type="text" th:field="*{name}" /></td>
        </tr>
        <tr>
                <td><input type="submit" value="Submit" /></td>
                </tr>
        </table>
</form>
```

Handling User Input

- */saveStudent* is the form action URL and a *student* is the object that holds the form data submitted.

```
@Controller
public class StudentController {
  @RequestMapping(value = "/saveStudent", method = RequestMethod.POST)
  public String saveStudent(@ModelAttribute @Valid Student student,
                                                    BindingResult errors, Model model) {
          // logic to process input data
  }
}
```

Displaying Validation Errors

- We can use the *#fields.hasErrors()* function to check if a field has any validation errors.
- And we use the *#fields.errors()* function to display errors for a particular field.

```
<ul>
        <li th:each="err : ${#fields.errors('id')}" th:text="${err}" />
        <li th:each="err : ${#fields.errors('name')}" th:text="${err}" />
</ul>
```

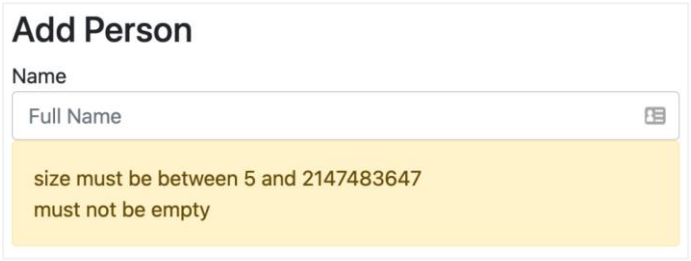
Displaying Validation Errors

- **Example**:

```
@Entity
public class Person {
        // ...
        @NotEmpty
        @Size(min = 5)
        private String fullName;
}
```

```
<div class="form-group">
    <label for="fullName">Name</label>
    <input class="form-control" type="text" th:field="*{fullName}" id="fullName" placeholder="Full Name">
    <div class="alert alert-warning" th:if="${#fields.hasErrors('fullName')}" th:errors="*{fullName}"></div>
</div>
```

- **Output:**



Displaying Validation Errors

- Instead of field name, the above functions accept the wild card character * or the constant *all* to indicate all fields.

```
<ul>
      <li th:each="err : ${#fields.errors('*')}" th:text="${err}" />
</ul>
```

- And here we're using the constant *all*:

```
<ul>
      <li th:each="err : ${#fields.errors('all')}" th:text="${err}" />
</ul>
```

Displaying Validation Errors

- We can display global errors in Spring using the ***global*** constant.

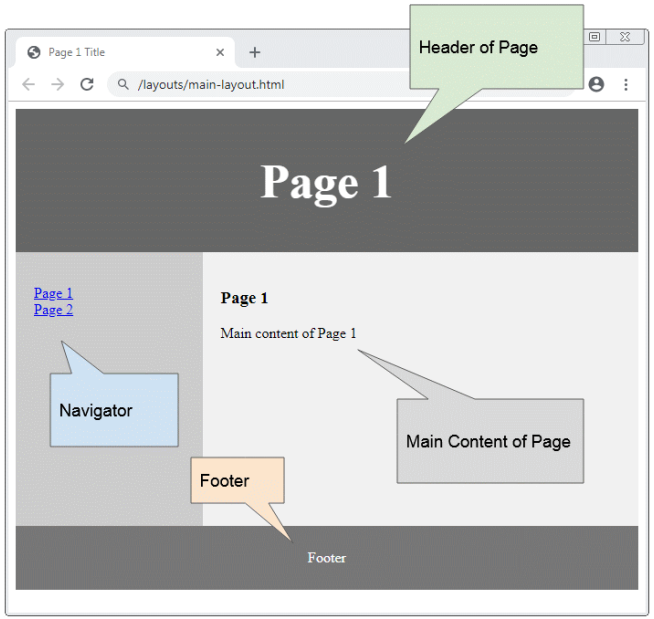
```
<ul>
	<li th:each="err : ${#fields.errors('global')}" th:text="${err}" />
</ul>
```

- We can use the ***th:errors*** attribute to display error messages.
- The previous code to display errors in the form can be rewritten using *th:errors* attribute:

```
<ul>
	<li th:errors="*{id}" />
	<li th:errors="*{name}" />
</ul>
```

Thymeleaf Page Layouts

- Usually websites share common page components like the **header**, **footer**, **menu** and possibly **many more**.
- These page components can be used by the same or different layouts.
- There are two main styles of organizing layouts in projects:
  - ✓ *include style* and
  - ✓ *hierarchical style*.
- Both styles can be easily utilized with Thymeleaf without losing its biggest value: natural templating.



https://www.thymeleaf.org/doc/articles/layouts.html

Thymeleaf Page Layouts

- **Include-style layouts:** In this style pages are built by embedding common page component code directly within each view to generate the final result.
- In Thymeleaf this can be done using Thymeleaf Standard Layout System:

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

*The include-style layouts are **pretty simple** to understand and implement and in fact they offer flexibility in developing views, which is their biggest advantage.*

*The **main disadvantage** of this solution, though, is that some code duplication is introduced so modifying the layout of a large number of views in big applications can become a bit cumbersome.*

Build a simple Web CRUD App

- Demo!

Summary

- @RequestParam, @PartVariable, @ModelAttribute
- Model, ModelMap and ModelAndView classes
- Spring Web MVC and Hibernate ORM
- Spring Framework Thymeleaf Tags Introduction
- Build a simple Web App (CRUD)

THANK YOU!