Gradle CRUD

config package is created to store the beans created during the usage

in the **BeanConfiguration** class, **@Component, @Configuration, @Controller, @Service, @Repository** annotation is used to make the class visible during a component scan

Default scope for design pattern: **singleton**

Every time we create an instance of a object, then the default scope singleton will create only once and use it for each newly created instances. For that reason, their references will be same. If you want to change the design pattern then you should add **@Scope("name")** on the top to identify it.

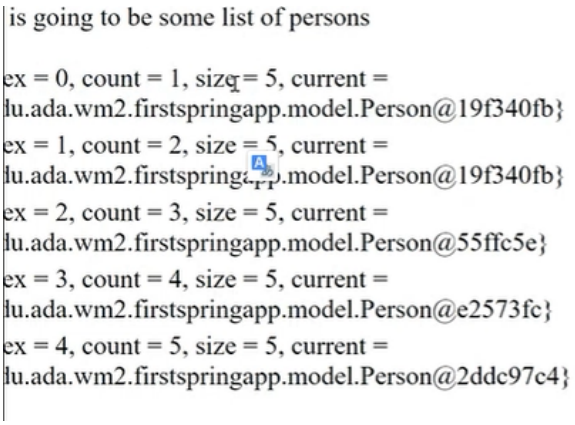
If you have two different beans for the same type then it is a problem. For example the IoC container may confuse which object to inject to the dependency. To solve the problem we should use **@Primary** annotation to define which bean is the primary one. Or during the autowired process, we may use **@Qualifier** annotation to select which bean to inject.

In A Spring Java web application, when we mark an instance variable with the `**final**` keyword in a class that serves as a reference to a Spring bean, we're essentially indicating that this variable should be set only once and not modified afterward. This is a standard Java language feature and not specific to Spring. Now, concerning constructor dependency injection and Lombok, here's how they come into play: Constructor injection is a method in Spring for providing dependencies to a class through its constructor, ensuring that all required dependencies are provided when an object is created. **Lombok**, on the other hand, is a library that helps reduce boilerplate code in Java classes by automatically generating methods like getters, setters, and constructors during compilation. So, when we combine the use of the `final` keyword with Lombok annotations in a Spring component class, Lombok can **automatically** generate a constructor taking into account the final fields, which Spring can then use for dependency injection. This means we don't need to write the constructor ourselves. For instance, in a service class annotated with `@Service`, if we mark a dependency as `final` and use Lombok's `@RequiredArgsConstructor`, Lombok will generate the necessary constructor for us, simplifying our code and allowing Spring to inject the dependencies properly.

In thymeleaf:

```
<tr th:each="person, rowStat : ${persons}">
   <!--       <td th:text="${rowStat}">Stat</td>-->
          <td th:text="${rowStat.index}">Stat</td>
          <td th:text="${person.id}">ID</td>
          <td th:text="${person.firstName}">First Name</td>
          <td th:text="${person.lastName}">Last Name</td>
</tr>
```

it is a statistic for the row and looks like that:



but the second bold object is just the index mentioned

```
@PostMapping("/save")
public String savePerson(@ModelAttribute("person") Person newPerson){
   this.personsList.add(newPerson);
   return "redirect:/persons";
}
```

In there, we use **@ModelAttribute** to pass the data gotten from the form to the PostMapping (if it is MVC container). If it is Restful container then we use Request body

In the Service package we generally do all the job using the methods and Controller is just handles the requests using Service methods.

Maven SessionAttributeDemo

In a Cart example, we store the items inside the Session to see them *(saving in the session scope)*

Code example

1.
```
@GetMapping("/test1")
public String testEndpoint(Model model){
   model.addAttribute("object", "Welcome!");
   return "redirect:/demo/test2";
}
```

   model.addAttribute("object", "Welcome!"): This line adds an attribute to the Model object. Attributes added to the Model object are typically used to pass data from the controller to the view. In this case, an attribute named "object" is added with the value "Welcome!".

2.
```
@Controller
@RequestMapping("/demo")
@SessionAttributes({"object"})
```

```
public class DemoController {

    @GetMapping("/test1")
    public String testEndpoint(Model model){
        model.addAttribute("object", "Welcome!");
        return "redirect:/demo/test2";
    }

    @GetMapping("/test2")
    public String test2Endpoint(){

        return "demo/list";
    }
}
```

In there, we pass the model to the test2 but to do that, we have to have the object saved somewhere else. For that reason, we need to use some form of persistence like **session attributes**, query parameters, or path variables.
In Spring MVC, the default scope of model attributes is **request scope**. This means that model attributes added in a controller method are only available for the duration of the current request.

3. The HTML snippet `**<p th:text="${object}">** The value of project attribute</p>` is a representation of how Thymeleaf, a server-side Java template engine, dynamically sets the text content of an HTML paragraph element. The `th:text` attribute is a Thymeleaf directive used to bind data from the model to the HTML view. In this case, `${object}` represents a variable named "object" retrieved from the model, and its value will be displayed within the paragraph element. If the "object" attribute is present in the model, its value will replace the static text "The value of project attribute" within the paragraph. If the attribute is not found, the fallback static text will be displayed instead. This mechanism allows for dynamic content rendering in web pages based on data provided by the server-side application.

4. **@SessionAttributes({"object"})** can be used to have the object persisted inside session. We may use this every time even if there is no other attribute to save. But there is an alternative option to save only one attribute which is:
@SessionAttribute({"object"})
To use the Session attribute *(09:04)*

5. Both the `**@AllArgsConstructor**` and `**@NoArgsConstructor**` annotations provided by Lombok offer valuable tools for streamlining constructor management in Java classes, particularly within Spring applications. The `@AllArgsConstructor` annotation generates a constructor with arguments for all fields within a class, significantly reducing the need for developers to manually write constructors with numerous parameters. This is especially beneficial in scenarios involving dependency injection, where Spring can automatically inject dependencies into a class's constructor. On the other hand, the `@NoArgsConstructor` annotation instructs Lombok to generate a constructor without any arguments, commonly known

as a "no-args" constructor. This is invaluable for creating instances of a class without specifying initial values for its fields, which is often required in Spring when defining beans or working with frameworks that require default constructors. Combining these annotations allows for comprehensive constructor management, ensuring that classes can be easily instantiated with or without initial values for their fields, thus enhancing code readability and maintainability in Spring applications.

6.

```
@ModelAttribute("cart")
public Order cart(){
   return new Order();
}
```

The `**@ModelAttribute**` annotation in Spring MVC plays a crucial role in preparing model data for handling requests within a controller. When a method is annotated with `@ModelAttribute`, like the `cart()` method in the `ProductsController`, it signifies that this method acts as a setup phase before the main operations of the controller. In simpler terms, it's like getting everything ready before starting to work. This method is invoked automatically by Spring MVC before executing any handler methods in the controller. Its purpose is to populate the model with specific attributes, such as the "cart" object, which may be needed across multiple requests. By doing this, it ensures that necessary data is available for the controller to use without having to recreate it every time. This simplifies the handling of model attributes and allows the controller's handler methods to access or manipulate the "cart" object without the need to explicitly create it in each method. Additionally, when combined with `@SessionAttributes`, it enables the storage of attributes in the session between requests, ensuring data persistence across the user's session, such as in the case of a shopping cart in a web application. In essence, `@ModelAttribute` provides a convenient and efficient way to prepare and manage model data within a Spring MVC controller, enhancing code readability and maintainability while facilitating smoother request processing.

```
@Controller
@RequestMapping("/product")
@SessionAttributes({"cart"})
public class ProductsController {
   @ModelAttribute
   public Order cart(){
       return new Order();
   }
}
```

7. It is possible to have several mapping in **@GetMapping({“/”,{“/home”})**

8. In the example below, we see that it is returned as *“products/list”*. Why is that so,

```
@GetMapping({"/", "/list"})
public String getProductsPage(){
   return  "products/list";
}
```

We say that in the templates folder, the list view is located inside products folder. For that reason, we indicate it as products/list

9.

```
@GetMapping({"/"})
public String getProductsPage(Model model,
@ModelAttribute@ModelAttribute("cart") Order cart){
   model.addAttribute("cart", cart);
   return  "products/list";
}
```

This controller method, annotated with `@GetMapping({"/", "/list"})`, handles GET requests to the "/" and "/list" URLs. It takes two parameters: `Model model` and `@ModelAttribute Order cart`. The `Model model` parameter represents the model, a container for data passed to the view for rendering. Meanwhile, the `@ModelAttribute Order cart` parameter is annotated with `@ModelAttribute`, instructing Spring to bind the "cart" object from the model if it exists or create a new instance of the "Order" class if not. The method then adds the "cart" object to the model using `model.addAttribute("cart", cart)`, associating it with the key "cart". This ensures that the "cart" object is available to the view for rendering. Finally, the method returns the name of the view, "products/list", which Spring MVC will render to display product-related data to the user.

10.

Thymeleaf:

```
<form action="#" method="POST" th:action="@{/product/addToOrder}"
th:object="${order}">
   Select the product:
   <input type="text" th:name="product" placeholder="Product name"/>
<br/>
   <input type="submit" value="Add product"/>
</form>
```

Controller:

```
@PostMapping("/addToOrder")
public String addProductToOrder(@ModelAttribute("cart") Order cart,
                                @RequestParam("product") String
productName,
                                Model model){
   if(!productName.isBlank())
       cart.addProduct(productName);
   model.addAttribute("cart", cart);
   return "redirect:/product/list";
```

```
}
```

The provided Thymeleaf form captures user input for adding a product to an order. With its action attribute set to `@{/product/addToOrder}`, it directs the form data to the `/product/addToOrder` endpoint upon submission. The form is associated with the "order" object through the `th:object="${order}"` attribute, ensuring that the form data will be bound to this object for processing. The controller method annotated with `@PostMapping("/addToOrder")` handles the incoming POST request from the form. It receives the "order" object as a model attribute through `@ModelAttribute Order cart`, enabling access to the existing order details. Additionally, the `@RequestParam("product") String productName` parameter retrieves the product name entered by the user. The method then adds the selected product to the order using the provided product name. Upon completion, the user is redirected to the `/product/list` endpoint to view the updated order details. Through this seamless interaction between the Thymeleaf form and the controller method, user input is collected and processed, facilitating the addition of products to the order while ensuring a smooth user experience.

11.

```
<link rel="stylesheet" th:href="@{/styles.css}"/>
```

In there, the program will automatically go to the root directory which is static folder inside the resources. We may have multiple css files there. If you want to have a subfolder for that, then it will be like /css/styles.css then.

12.

```
@Controller
@RequestMapping("/order")
public class OrderController {

}
```

The configuration made to a class so that it behaves as a controller and accepts the requests

13.

```
<form action="#" method="POST" th:action="@{/orders/save}"
th:object="${cart}">
      Enter your name:
   <input type="text" th:field="*{customerName}" placeholder="Customer
name"/> <br/>
   <input type="submit" value="Save order"/>
</form>
```

If you see the template version, you notice that there is *cart* object binded inside the form. *BY the way,* the *th:action* there does send the input taken from the user in the form to the /orders/save

To get this object from the user, we add this:

```
public String listOrder(Model model, @SessionAttribute("cart") Order
cart){
   model.addAttribute("cart", cart);
   return "orders/order_detail";
}
```

In there, Model model and model.addAttribute will take the user input and bind it to our method. Then, SessionAttribute will just take the object saved in the session and pass it to the model.

14.

```
<form action="#" method="POST" th:action="@{/order/save}"
th:object="${order}">
      Enter your name:
   <input type="text" th:field="*{customerName}" placeholder="Customer
name"/> <br/>
   <input type="submit" value="Save order"/>
</form>
```

In there, as you see, we are using **order** object and getting **customerName**.
to pass these values to the controller, we use:

```
@PostMapping("/save")
public String saveOrder(@ModelAttribute Order order,
                        @RequestParam("customerName") String name){
   return "redirect:/product/";
}
```

In there, we use **ModelAttribute Order order** to take the object **order** that is taken and then we use **RequestParam** to take the value customerName

15.

```
@PostMapping("/save")
public String saveOrder(@SessionAttribute("cart") Order order,
                        @RequestParam("customerName") String name){
   System.out.println(name + " compilation of the order" + order);
   return "redirect:/product/";
}
```

By default, save mapping looks like this. After we put the name into the form, it shows the print statement in the ide and redirects to the product page. But the session attribute still stays the same for a reason. We have to delete it.

```
@PostMapping("/save")
public String saveOrder(@SessionAttribute("cart") Order order,
                        @RequestParam("customerName") String name,
                        SessionStatus ss,
```

```
                        WebRequest webRequest){
    order.setcustomerName(name);
    System.out.println(name + " compilation of the order: " + order);
    ss.setComplete();
    webRequest.removeAttribute("cart", WebRequest.SCOPE_SESSION);
    return "redirect:/product/";
}
```

But when we added some lines there, we use the power of *SessionStatus*, and *WebRequest* so that at the end we can complete the session and remove its attribute. `WebRequest.SCOPE_SESSION` is there to mention the scope but it may be replaced by 1 simply.