Practice 4: MVC, Services, Controllers and Views, part I

1. Open the project lab4 in IntelliJ IDE: File – New Project from Existing Sources. Add H2 and MySql run configurations **-Dspring.profiles.active=H2**. When H2 profile is running the H2 console is available at: http://localhost:8080/h2-console.

2. Add a new package com.awbd.lab4.services and a new interface com.awbd.lab4.services .ProductsService:

```
public interface ProductService {
    List<Product> findAll();
    Product findById(Long l);
    Product save(Product product);
    void deleteById(Long id);
}
```

3. Implement com.awbd.lab4.services .ProductsService:

```
@Service
public class ProductServiceImpl implements ProductService {
    ProductRepository productRepository;

    @Autowired
    public ProductServiceImpl(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

    @Override
    public List<Product> findAll(){
        List<Product> products = new LinkedList<>();
        productRepository.findAll(Sort.by("name")
).iterator().forEachRemaining(products::add);
        return products;
    }

    @Override
    public Product findById(Long l) {
        Optional<Product> productOptional =
productRepository.findById(l);
        if (!productOptional.isPresent()) {
            throw new RuntimeException("Product not found!");
        }
        return productOptional.get();
    }

    @Override
    public Product save(Product product) {
        Product savedProduct = productRepository.save(product);
        return savedProduct;
    }

    @Override
    public void deleteById(Long id) {
        productRepository.deleteById(id);
    }
}
```

Info

Stereotypes annotations are used for different classification. [1]

@Service – component holding the business logic, service layer
@Repository -- persistence layer, database repository

Both annotations are specializations of @Component annotation.

4. Add a new test package com.awbd.lab4.services and a new test class:

```
@ExtendWith(MockitoExtension.class)
public class ProductServiceTest {
    @Mock
    ProductRepository productRepository;

    @InjectMocks
    ProductServiceImpl productService;

    @Test
    public void findProducts() {
        List<Product> productsRet = new ArrayList<Product>();
        Product product = new Product();
        product.setId(4L);
        product.setCode("TEST");
        productsRet.add(product);


when(productRepository.findAll(Sort.by("name"))).thenReturn(productsRet)
;
        List<Product> products = productService.findAll();
        assertEquals(products.size(), 1);
        verify(productRepository, times(1)).findAll(Sort.by("name"));
    }
}
```

Info

Unit Tests
Test specific sections of code/individual units of a software.
test a method
No external dependencies (SpringContext, database etc.)
Few inputs.
Faster than integration tests.

Integration Tests
May use external dependencies.
Test interaction between objects.
Slower than unit test, big-bang approach/top-down, bottom-up, sandwich-approach.

Mock
Dummy implementation for a class.
Simulate real behavior, simplified implementation of an object used in tests.
Register the calls it receives.
("fake object"/"stub object").

Spy
Mock with some real implementations.
Useful dependencies (see spring Initializr):
JUnit, Spring Test, Spring Boot Test, Mockito, AssertJ

Junit Annotations:

@Test test method
@Before method executed before each test, used for initializations
@After method executed after each test, used for cleanup
@BeforeClass method executed only once, before all tests.
@AfterClass method executed only once, after all tests.
@Ignore test will not be performed
@Test (expected = Exception.class)
test succeeds if Exception.class is thrown
@Test (timeout = 100)
test succeeds if it runs in less than100 ms.

Mockito annotations [2][3]:

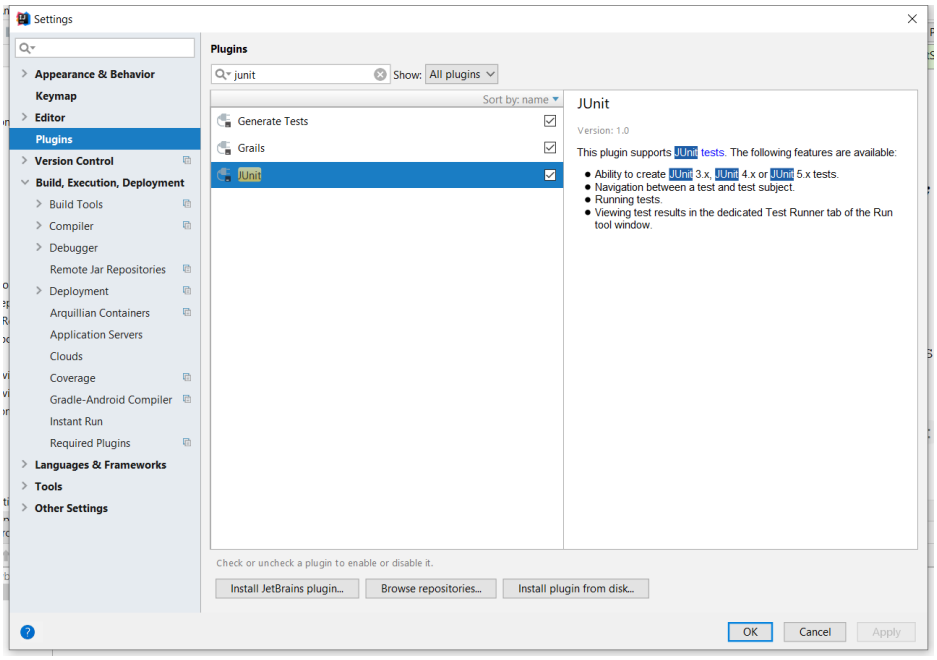
@Rule JUnit versions (>= 4.7), @RunsWith may be replaced with @Rule. @Rule is used by a to indicate work done before and after a test's execution. Older versions: @RunWith(MockitoJUnitRunner.class)

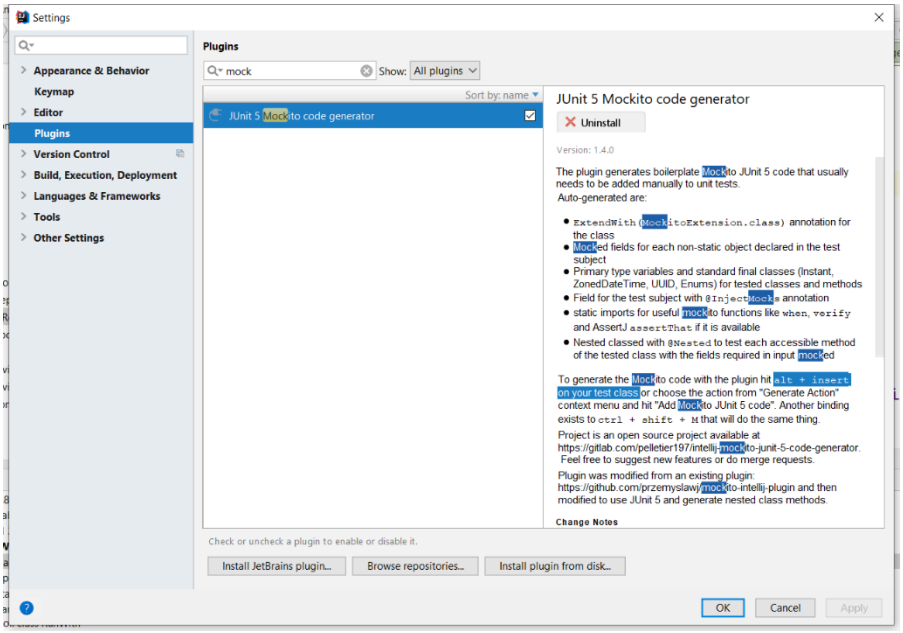
@Mock
@InjectMock create and inject mocked instances

When()
thenReturn() are used to specify a return value for a method call.

verify() is used to check the number of calls for a given method.

IntelliJ plugin:





5. Add in pom.xml thymeleaf dependency. SpringBoot will autoconfigure a ViewResolver for Thymeleaf templates.

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

Info **Spring MVC**

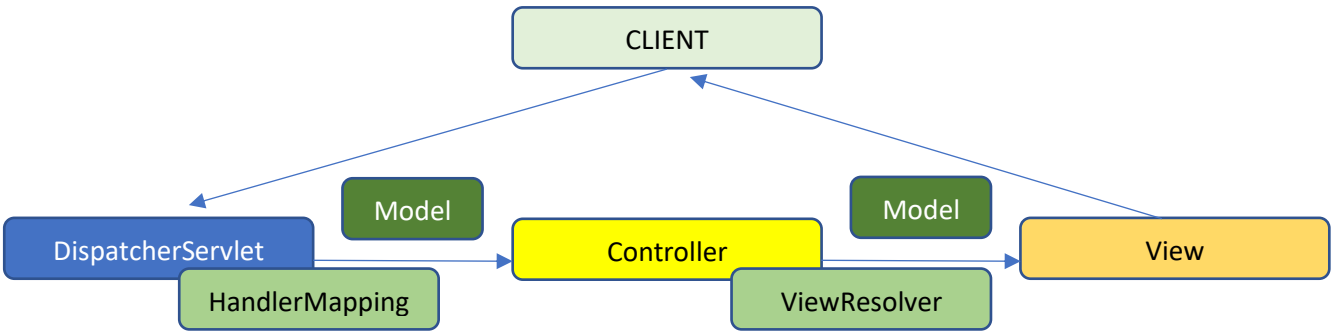
Spring MVC framework **[4][5]** is designed around a central Servlet: **DispatcherServlet** that dispatches requests to controllers.

WebApplicationContext contains:

HandlerMapping: maps incoming requests to handlers. The most common implementation is based on annotated **Controllers**

HandlerExceptionResolver: maps exceptions to views.

ViewResolver: resolves string-based view names based on view types.



6. Add webjar dependency:

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>4.5.0</version>
</dependency>
```

Info **WebJars** [6]:

Client dependencies packed in JAR archives. Easy to manage with maven.
Popular webjars: Bootstrap, JQuery, Angular JS etc.

To automatically resolve the version of any WebJars assets we must include webjars-locator as dependency:

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator</artifactId>
    <version>0.43</version>
</dependency>
```

7. Modify products.html, add bootstrap and thymeleaf namespace:

```
<link rel="stylesheet"
href="/webjars/bootstrap/4.5.0/css/bootstrap.min.css"/>
<script src="/webjars/jquery/3.5.1/jquery.min.js"></script>
<script src="/webjars/bootstrap/4.5.0/js/bootstrap.min.js"></script>
```

8. Add a new package com.awbd.lab4.controllers and a new class ProductController and test http://localhost:8080/products:

```
@Controller
@RequestMapping("/products")
public class ProductController {
    ProductService productService;

    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @RequestMapping("")
    public String productList() {
        return "productList";
    }

}
```

9. Add thymeleaf tags in producs.html:

```
<tr th:each="product, stat : ${products}"
            th:class="${stat.odd}? 'table-light':'table-dark'">
            <td th:text="${product.id}">1</td>
            <td th:text="${product.name}">Product 1</td>
            <td th:text="${product.code}">Code</td>
            <td th:text="${product.reservePrice}">Reserved price</td>
            <td th:text="${product.reservePrice}">Best offer</td>
            <td><a href="#" th:href="@{'/products/' + ${product.id}}"><i
class="fa-solid fa-circle-info"></i></a></td>
            <td><a href="#" th:href="@{'/products/edit/' +
${product.id}}"><i class="fa-solid fa-pen"></i></a></td>
            <td><a href="#" th:href="@{'/products/delete/' +
${product.id}}"><i class="fa-solid fa-trash"></i></a></td>
        </tr>
```

Info **Thymeleaf**

Thymeleaf [7][8] is a Java template engine for processing HTML, XML, CSS etc.

Model attributes from Spring are available in Thymeleaf as “context variables”. Context variables are accesed with Spring EL expressions [9]. Spring Expression Language is a language for query and manipulate object graph at runtime.

Model Attributes are accesed with:
${attributeName}

Request parameters are accessed with:

${param.param_name}

Iteration [10]

th:each iterates collections (java.util.Map, java.util.Arrays, java.util.Iterable etc.)

```
<tr th:each="product : ${products}">
```

The following properties may be accessed via status variable:
index (itteration index, starting from 0), ***count*** (total number of elements processed), ***size*** (total number of elements), ***even/odd*** boolean, ***first*** (boolean – true if current element is the first element of the collection), ***last*** (boolean true if current element is the last element of the collection)

```
<tr th:each="product, stat : ${products}"
    th:class="${stat.odd}? 'table-light':'table-dark'">
```

10. Modify class ProductController and test http://localhost:8080/product. Check the HTML generated with Thymeleaf.

```
@RequestMapping("")
public String productList(Model model) {
    List<Product> products = productService.findAll();
    model.addAttribute("products",products);
    return "productList";
}
```

Model, ModelAndView

Info

Model [11][4] holds the attributes for rendering views. @RequestMapping annotated methods accept an attribute of type Model. Attributes are added in model with ***addAttribute*** method.

ModelAndView stores both the model and the view resolved by ViewResolver. Model attributes are store as a map and added with ***addObject***.

11. Modify @RequestMapping productList, use ModelAndView:

```
@RequestMapping("")
public ModelAndView products(){
    ModelAndView modelAndView = new ModelAndView("productList");
    List<Product> products = productService.findAll();
    modelAndView.addObject("products",products);
    return modelAndView;
}
```

12. Add a method to process request to show a form with details about a product with a given id:
http://localhost:8080/products/2

```
@GetMapping("/{id}")
public String getById(@PathVariable String id, Model model){
    model.addAttribute("product",
            productService.findById(Long.valueOf(id)));
    return "productDetails";
}
```

Info

RequestMapping [12][13]

@RequestMapping map annotation is used to map web requests to Spring Controller methods.
If ***method*** parameter is not specified @RequestMapping will map any HTTP request.
Parameters ***headers*** or ***produces*** may be used to specify value for header *accept*.

```
@RequestMapping(
 value = "/ex/foos",
 method = GET,
 produces = { "application/json"}
)
@ResponseBody
```

```
@RequestMapping(
 value = "/ex/foos",
 method = GET,
 headers = "Accept=application/json")
@ResponseBody
```

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping are shortcuts .

@GetMapping("/get/{id}") is a shortcut for
@RequestMapping(value = "/get/{id}", method = RequestMethod.GET)

@PathVariable maps parts of the URL mapping to variables.
@GetMapping("/product/info/{id}")
public String showById(@PathVariable String id, Model model)

@GetMapping("/product/info/{id}")
public String showById(@PathVariable("id") String id, Model model)

If the name of the method matches the name of the path variable, the value of @Pathvariable may be omitted:
@GetMapping("/product/info/{id}")
public String getById(@PathVariable String id, Model model)

13. Add Thymeleaf context variables in info.html**.**

```
<h1 class="panel-title" th:text="${product.name}">Product</h1>

<li th:each="category:${product.categories}"
th:text="${category.getName()}">
category 3
</li>


<p th:text="${product.code != null ?(product.code):'code'}">code</p>

<p th:text="${product.reservePrice != null ? product.reservePrice:'$'}">
price
</p>


<p th:text="${product.info != null ? product.info.description:''}">
Info
</p>
```

14. Add a method in class Product to remove a category from the product's categories list, and change delete method in ProductService to remove all categories before deleting the product:

```
public void removeCategory(Category category) {
    category.getProducts().remove(this);
    categories.remove(category);
}


@Override
public void deleteById(Long id) {
    Optional<Product> productOptional = productRepository.findById(id);
    if (!productOptional.isPresent()) {
        throw new RuntimeException("Product not found!");
    }
    Product product = productOptional.get();
    List<Category> categories = new LinkedList<Category>();

product.getCategories().iterator().forEachRemaining(categories::add);
categories.iterator().forEachRemaining(product::removeCategory);


    productRepository.save(product);
    productRepository.deleteById(id);
}
```

15. Add the request to delete a product by id:

```
@RequestMapping("/delete/{id}")
public String deleteById(@PathVariable String id){
    productService.deleteById(Long.valueOf(id));
    return "redirect:/products";
}
```

16. Add in productform.html:

```
<form enctype="multipart/form-data" method="post"
th:action="@{/products}" th:object="${product}">


<input th:field="*{id}" type="hidden"/>


<input class="form-control" th:field="*{name}" type="text"
placeholder="product name"/>

<input class="form-control" th:field="*{code}" type="text"
placeholder="product code"/>

<input class="form-control" th:field="*{reservePrice}" type="number"
placeholder="reserve price"/>

<textarea class="form-control" th:field="*{info.description}"
placeholder="description" />


<label th:for="restored">Restored</label>
<input th:field="*{restored}" type="checkbox"/>


<button class="btn btn-primary" type="submit">Submit</button>
```

17. Add @RequestMapping method to return productForm.html:

```
@RequestMapping("/form")
public String newProduct(Model model) {
    model.addAttribute("product", new Product());
    return "productform";
}
```

18. Add a post mapping to add a new product:

```
@PostMapping("")
public String saveOrUpdate(@ModelAttribute Product product,
                           @RequestParam("imagefile") MultipartFile file
){
    Product savedProduct = productService.save(product);
    return "redirect:/products" ;
}
```

19. Add classes com.awbd.lab4.services.CategoryServiceImpl and com.awbd.lab6.services.CategoryService.

Add in ProductsController a field of type com.awbd.lab4.services.CategoryService. Annotate categoryService @Autowired.

```
public interface CategoryService {
    List<Category> findAll();
}
```

```
@Service
public class CategoryServiceImpl implements CategoryService{

    CategoryRepository categoryRepository;

    @Autowired
    public CategoryServiceImpl(CatagoryRepository categoryRepository) {
        this.categoryRepository = categoryRepository;
    }

    @Override
    public List<Category> findAll() {
        List<Category> categories = new LinkedList<>();
     categoryRepository.findAll().iterator().forEachRemaining(categories
::add);
        return categories ;
    }
}
```

20. Add in productForm.html:

```
<ul id="categories" style="list-style: none;">
    <li th:each="category: ${categoriesAll}">
        <input th:field="*{categories}"
               th:value="${category.id}"
               type="checkbox"/>
        <label
                th:for="${#ids.prev('categories')}"
                th:text="${category.name}">
        </label>
    </li>
</ul>
```

21. Add in info.html:

```
<ul>
    <li
th:each="category:${product.categories}"th:text="${category.getName()}">
        category
    </li>
</ul>
```

22. When "/product/new" request is processed add in model an object containing the list of categories:

```
@RequestMapping("/form")
public String newProduct(Model model) {
    model.addAttribute("product", new Product());
    List<Category> categoriesAll = categoryService.findAll();
    model.addAttribute("categoriesAll", categoriesAll );
    return "productform";
}
```

23. Add a request mapping that will return the form to update an existing product:

```
@RequestMapping("/edit/{id}")
public String edit(@PathVariable String id, Model model) {
    model.addAttribute("product",
            productService.findById(Long.valueOf(id)));

    List<Category> categoriesAll = categoryService.findAll();
    model.addAttribute("categoriesAll", categoriesAll );

    return "productForm";
}
```

24. Add in productDetails.html tymeleaf template:

```
<div th:if="${product.currency ==
T(com.awbd.lab6.domain.Currency).EUR}">
    &euro;
</div>

<div th:if="${product.currency ==
T(com.awbd.lab6.domain.Currency).USD}">
    &dollar;
</div>

<div th:if="${product.currency ==
T(com.awbd.lab6.domain.Currency.GBP}">
    &pound;
</div>
```

25. Modify Currency Enumeration. Add an attribute description, a constructor, and a getter method:

```
public enum Currency {
    USD("USD $"), EUR("EUR"), GBP("GBP");

    private String description;

    public String getDescription() {
        return description;
    }

    Currency(String description) {
        this.description = description;
    }
}
```

26. Add in productform.html template:

```
<label for="currency">Currency: </label>
<select id="currency" name="currency" th:field="*{currency}">
    <option

th:each="currencyOpt:${T(com.awbd.lab4.domain.Currency).values()}"
            th:value="${currencyOpt}"
th:text="${currencyOpt.getDescription()}">

    </option>
</select>
```

Info **Enumerations** [14]

T from Spring Expression Language specifies an instance of a class or static methods.
Enums are special types of classes extending *java.lang.Enum*. We can define custom methods and constructors. We may use Enums in if or switch statements.

B

[1] https://www.baeldung.com/spring-component-repository-service

[2] https://www.baeldung.com/mockito-annotations

[3] https://alexecollins.com/tutorial-junit-rule/

[4] https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html

[5] https://www.baeldung.com/spring-mvc-tutorial

[6] https://www.baeldung.com/maven-webjars

[7] https://www.thymeleaf.org/doc/articles/springmvcaccessdata.html

[8] https://www.baeldung.com/thymeleaf-in-spring-mvc

[9] https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#expressions

[10] https://www.baeldung.com/thymeleaf-iteration

[11] https://www.baeldung.com/spring-mvc-model-model-map-model-view

[12] https://www.baeldung.com/spring-requestmapping

[13] https://www.baeldung.com/spring-new-requestmapping-shortcuts

[14] https://www.baeldung.com/thymeleaf-enums