Practice 5: Testing, form validations, exception handling

1. Open the project lab5 in IntelliJ IDE: File – New Project from Existing Sources. Add H2 and MySql run configurations **-Dspring.profiles.active=** MySql.

2. Add a new test package *com.awbd.lab5.controllers* and a new test class *com.awbd.lab5.controllers.ProductsControllerTest*. The method *showById* will test if *ProductController* adds in *Model* argument the object of type *Product* returned by *findById* method of *ProductsService* class.

```
@ExtendWith(MockitoExtension.class)
public class ProductControllerTest {

    @Mock
    Model model;

    @Mock
    ProductService productService;

    ProductController productController;


    @BeforeEach
    public void setUp() throws Exception {
        productController = new ProductController(productService);
    }

    @Test
    public void showById() {
        Long id = 11;
        Product productTest = new Product();
        productTest.setId(id);

        when(productService.findById(id)).thenReturn(productTest);

        String viewName = productController.getById(id.toString(),
model);
        assertEquals("productDetails", viewName);
        verify(productService, times(1)).findById(id);

        ArgumentCaptor<Product> argumentCaptor =
ArgumentCaptor.forClass(Product.class);
        verify(model, times(1))
                .addAttribute(eq("product"), argumentCaptor.capture() );

        Product productArg = argumentCaptor.getValue();
        assertEquals(productArg.getId(), productTest.getId() );

    }

}
```

Info **ArgumentCaptor** [1] is used to capture an argument passed in the invocation of a method. The constructor takes as argument the type of the argument to be captured.

Instead of using the ArgumentCaptor(type) constructor, we can inject an ArgumentCaptor object with annotation **@Captor**

Method **getValue**() returns the value of the argument.

3.

```
ArgumentCaptor<Product> argumentCaptor =
ArgumentCaptor.forClass(Product.class);
```

Replace

with class filed:

```
@Captor
ArgumentCaptor<Product> argumentCaptor;
```

Info **MockMvc** [2][3] object encapsulates web application beans and allows testing web requests. Available options are:

- Specifying headers for the request.
- Specifying request body.
- Validate the response:
  check HTTP - status code,
  check response headers,
  check response body.

When running an **integration test** different layers of applications are involved.

@AutoConfigureMockMvc annotation instructs Spring to create a *MockMvc* object, associated with the application context, prepared to send requests to **TestDispatcherServlet** which is an extension of DispatcherServlet**.** Requests are sent by calling the *perform* method.
If @AutoConfigureMockMvc annotation is used, MockMvc object can be injected with @Autowired annotation.

@SpringBootTest [4] bootstraps the entire Spring container.
Values for **webEnvironment** [5] property of @SpringBootTest annotation:

RANDOM_PORT: EmbeddedWebApplicationContext, real servlet environment.
Embedded servlet containers are started and listening on a random port.

DEFINED_PORT: EmbeddedWebApplicationContext, real servlet environment.
Embedded servlet containers are started and listening on a defined port (i.e from application.properties or on the default port 8080).

NONE: loads ApplicationContext using SpringApplication, does not provide any servlet environment.

Info **Junit 5 extensions** [12] extend the behavior of test class or methods. Extensions are related to a certain event in the execution of a test (extension point). For each extension point we implement an interface. **@ExtendWith** annotation registers test extensions.

MockitoExtension.class finds member variables annotated with **@Mock** and creates a mock implementation of those variables. Mocks are then injected into member variables annotated with the **@InjectMocks** annotation, using either construction injection or setter injection.

@MockBean [6] adds mock objects to Spring application context. The mock will replace any existing bean of the same type in the application context.

4. Add integration test *com.awbd.lab5.ProductsControllerIntegrationTest* which will test if the view return by request /product/info/{id} is "info.html":

```
@SpringBootTest
@AutoConfigureMockMvc
public class ProductControllerIntegrationTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    public void showByIdMvc() throws Exception {

        mockMvc.perform(get("/products/{id}", "1"))
                .andExpect(status().isOk())
                .andExpect(view().name("productDetails"));
    }
}
```

5. Use @MockBean to test that *ProductController* adds “product” object in Model:

```
@SpringBootTest
@AutoConfigureMockMvc
public class ProductControllerMockMvcTest {

    @Autowired
    MockMvc mockMvc;

    @MockBean
    ProductService productService;

    @MockBean
    Model model;

    @Test
    public void getByIdMockMvc() throws Exception {
        Long id = 11;
        Product productTest = new Product();
        productTest.setId(id);
        productTest.setName("test");

        when(productService.findById(id)).thenReturn(productTest);

        mockMvc.perform(get("/products/{id}", "1"))
                .andExpect(status().isOk())
                .andExpect(view().name("productDetails"))
                .andExpect(model().attribute("product", productTest))
                .andExpect(content().contentType("text/html;charset=UTF-
8"));;
    }
}
```

6. Verify request "products/getimage/{id}", check that the **content type** of the response is "image/jpeg":

```
@SpringBootTest
@AutoConfigureMockMvc
public class ImageControllerTest {
    @Autowired
    MockMvc mockMvc;

    @Test
    public void getImage() throws Exception {

        //test product with info.image not null
        mockMvc.perform(get("/products/getimage/{id}", "5"))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.IMAGE_JPEG));

    }
}
```

Info **Exception Handling**

@ResponseStatus [7] annotate custom exception class to indicate the HTTP status to be return when the exception is thrown.
Server Unhandled exceptions – **HTTP 500** status code.

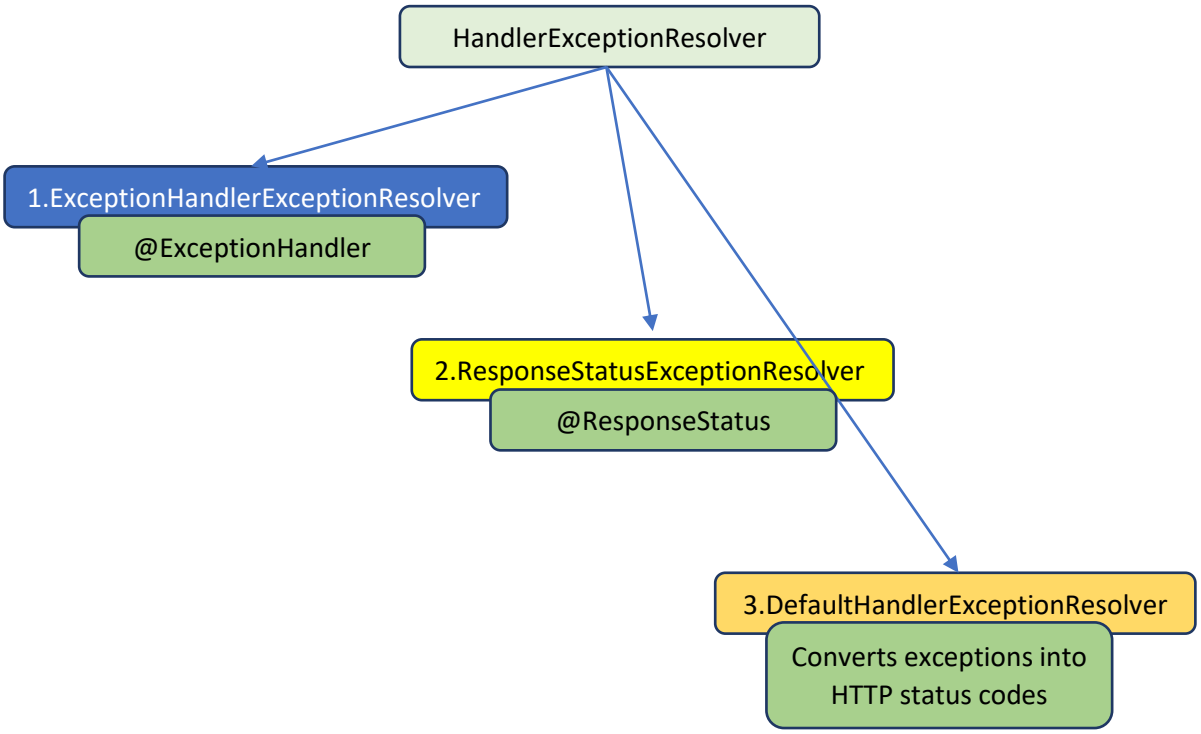
Client Errors: **400 Bad Request.**
401 Unauthorized -- Authentication Required.
404 Not Found -- Resource not found
405 Method not Allowed.

@ExceptionHandler [8] Defines custom exception handling at Controller level:
can define a specific status code.
can return a specific view with details about the error.
can work with *ModelAndView* object.
@ExceptionHandler methods don't have access to context Model.

HandlerExceptionResolver [8] is used internally by Spring to intercept and process any exception raised in the MVC system and not handled by a Controller.

```
public interface HandlerExceptionResolver {
   ModelAndView resolveException(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex);
}
```

The parameter handler refers to the controller that generated the exception.
Three default implementations are created for HandlerExceptionResolver and processed in order by *HandlerExceptionResolverComposite* bean:



SimpleMappingExceptionResolver [8]

--Map exception class names to view names.
--Specify a fallback error page for exceptions not associated with a specific view.
--Add *exception* attribute to the model.

7. Create a new package *com.awbd.lab5.exceptions* and a custom exception class that will be thrown if a participant id or a product id is not found in the database.

```
package com.awbd.lab5.exceptions;

public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException() {
    }

    public ResourceNotFoundException(String message) {
        super(message);
    }

    public ResourceNotFoundException(String message, Throwable
throwable) {
        super(message, throwable);
    }
}
```

8. Throw a *ResourceNotFoundException* error when the participant id or the product id is not found in the database, modify methods *findById* in *ProductService* and *ParticipantService*.
Test http://localhost:8080/products/10.

```
throw new ResourceNotFoundException("product " + id + " not found");
```

9. Annotate ResourceNotFoundException with @ResponseStatus.
Test http://localhost:8080/products/10:

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
```

10. Write an @ExceptionHandler method in *ProductController* class. Test http://localhost:8080/products/10

```
@ExceptionHandler(ResourceNotFoundException.class)
public ModelAndView handlerNotFoundException(Exception exception){
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.getModel().put("exception",exception);
    modelAndView.setViewName("notFoundException");
    return modelAndView;
}
```

11. Test *ProductControllerIntegrationTest* add a method to test expected status and expected view in case of ResourceNotFoundException.

```
@Test
public void showByIdNotFound() throws Exception {

    mockMvc.perform(get("/products/{id}", "17"))
            .andExpect(status().isNotFound())
            .andExpect(view().name("notFoundException"));
}
```

12. Annotate *handlerNotFoundException* method with @ResponseStatus(HttpStatus.NOT_FOUND).
Re-run integration test *ProductControllerIntegrationTest*:

13. Test http://localhost:8080/products/getimage/20. What view is return by *ImageController*?
In order to handle the *ResorceNotFoundException* thrown in *ImageController*, without duplicating code, add a **@ControllerAdvice** class which will handle Exceptions globally, for all controllers.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler(ResourceNotFoundException.class)
    public ModelAndView handlerNotFoundException(Exception exception){
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.getModel().put("exception",exception);
        modelAndView.setViewName("notFoundException");
        return modelAndView;
    }

}
```

14. Test http://localhost:8080/products/abc. You will get a NumberFormatException.
Add *package com.awbd.lab5.configuration*.

15. Create a SimpleMappingExceptionResolver bean that will map NumberFormatException to a default view, *error.html.*

```
@Configuration
public class MvcConfiguration implements WebMvcConfigurer {

    @Bean(name="simpleMappingExceptionResolver")
    public SimpleMappingExceptionResolver
    getSimpleMappingExceptionResolver() {
        SimpleMappingExceptionResolver r =
                new SimpleMappingExceptionResolver();

        r.setDefaultErrorView("defaultException");
        r.setExceptionAttribute("ex");     // default "exception"


        return r;
    }
```

16. Map errors to view and status codes:

```
Properties mappings = new Properties();
mappings.setProperty("NumberFormatException", "numberFormatException");
r.setExceptionMappings(mappings);

Properties statusCodes = new Properties();
statusCodes.setProperty("NumberFormatException", "400");
r.setStatusCodes(statusCodes);
```

Info **Java bean validation API** (Hibernate) [9] allows to express and validate application constraints ensuring that the beans meet specific criteria.

Examples of annotations:

- **@Size** filed length
- **@Min @Max** used for numbers
- **@Pattern** checking regular expressions
- **@NotNull**

17. Add dependencies for Java bean validation API:

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
</dependency>
```

18. Check that the minimum price for a product is 100. Check that the product code contains only letters.

```
@Min(value=100, message ="min price 100")
private Double reservePrice;
```

```
@Pattern(regexp = "[A-Z]*", message = "only letters")
private String code;
```

19. Change saveOrUpdate method, add parameter BindingResult bindingResult.

```
@PostMapping("")
public String saveOrUpdate(@Valid @ModelAttribute Product product,
                           BindingResult bindingResult,
                           @RequestParam("imagefile") MultipartFile file
){
    if (bindingResult.hasErrors()){
        return "productForm";
    }


  …
}
```

20. In Thymeleaf template productform.html, add a label to display errors for reservedPrice filed.

```
<label th:if="${#fields.hasErrors('reservePrice')}"
th:errors="*{reservePrice}">Error</label>
```

B

[1] https://www.baeldung.com/mockito-argumentcaptor

[2] https://spring.io/guides/gs/testing-web/

[3] https://www.baeldung.com/integration-testing-in-spring

[4] https://www.baeldung.com/spring-boot-testing

[5] https://docs.spring.io/spring-boot/docs/1.5.3.RELEASE/reference/html/boot-features-testing.html

[6] https://www.baeldung.com/java-spring-mockito-mock-mockbean

[7] https://www.baeldung.com/spring-response-status

[8] https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc

[9] http://hibernate.org/validator/

[10] https://www.baeldung.com/javax-validation

[11] https://www.infoworld.com/article/3543268/junit-5-tutorial-part-2-unit-testing-spring-mvc-with-junit-5.html

[12] https://www.baeldung.com/junit-5-extensions