Practice 6: Testing, form validations, exception handling

1. Open the project lab6 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.lab6.controllers and a new test class com.awbd.lab6.controllers. ProductsControllerTest. The method showById will test if *ProductController* adds in Model the Product object returned by *findById* method of *ProductsService* class.

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

    @Mock
    Model model;

    @Mock
    ProductService productService;

    ProductsController productsController;


    @BeforeEach
    public void setUp() throws Exception {
         productsController = new ProductsController();
         productsController.setProductService(productService);
    }

    @Test
    public void showById() {
        Long id = 1l;
        Product productTest = new Product();
        productTest.setId(id);

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

        String viewName = productsController.showById(id.toString(),
model);
        assertEquals("info", 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 by 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. Replace

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

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.** Requests are sent by calling the *perform* method. **TestDispatcherServlet** is an extension of DispatcherServlet.
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 finds member variables annotated with the **@InjectMocks** annotation, using either construction injection or setter injection.

4. Add integration test com.awbd.lab6.ProductsControllerTest which will test if the view return by request /product/info/{id} is “info.html”:

```
@SpringBootTest
@AutoConfigureMockMvc
public class ProductsControllerTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    public void showByIdMvc() throws Exception {

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

Info

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

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

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@SpringBootTest
@AutoConfigureMockMvc
public class ProductsControllerTest {

    @Autowired
    MockMvc mockMvc;


    @MockBean
    ProductService productService;

    @MockBean
    Model model;

    @Test
    public void showByIdMockMvc() throws Exception {
        Long id = 1l;
        Product productTest = new Product();
        productTest.setId(id);
        productTest.setName("test");

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

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

    }
}
```

6. Verify "product/getimage/{id}" request, 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("/product/getimage/{id}", "5"))
                .andExpect(status().isOk())
                .andExpect(content().contentType(MediaType.IMAGE_JPEG));

    }
}
```

Info

Exception Handling

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.	

@ResponseStatus [7] annotate custom exception class to indicate the HTTP status to be return when the exception is thrown.

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

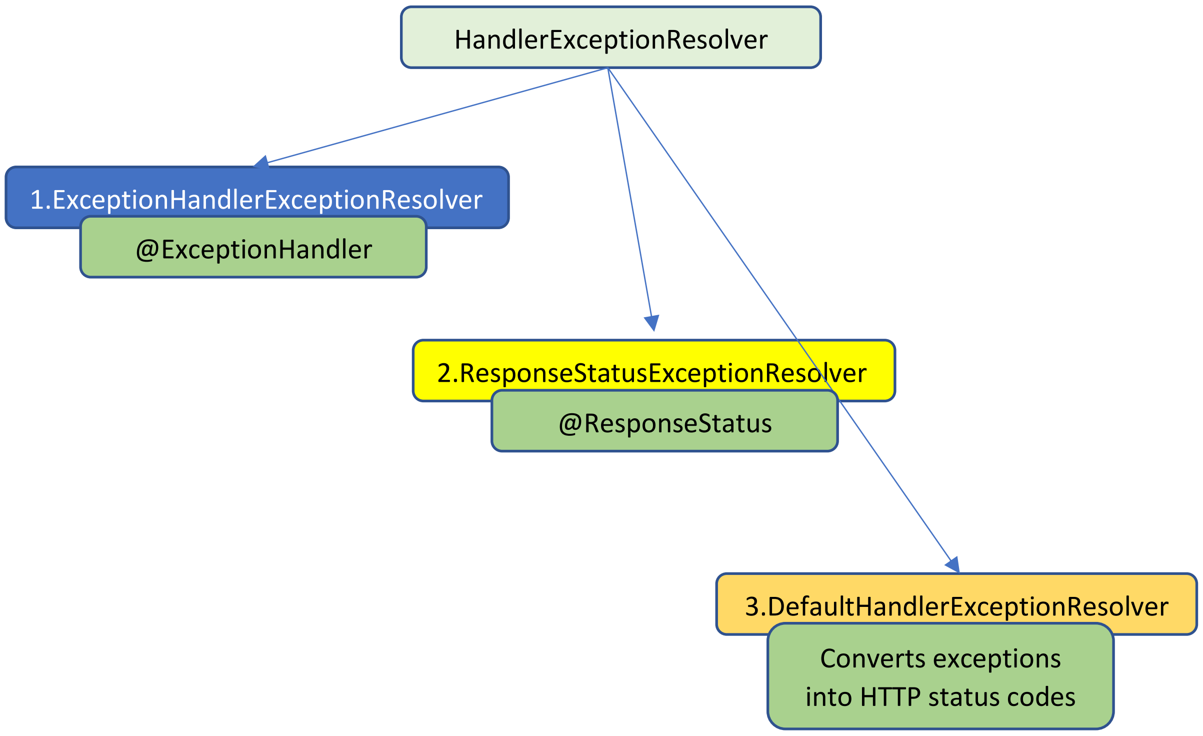
HandlerExceptionResolver [8] used internally by Spring to intercept and process any exception raised in the MVC system and not handled by a Controller.

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

The parameter handler refers to the controller that generated the exception.

Info

Three default implementations are created for HandlerExceptionResolver and processed in order by *HandlerExceptionResolverComposite* bean:


HandlerExceptionResolver
1.ExceptionHandlerExceptionResolver
@ExceptionHandler
2.ResponseStatusExceptionResolver
@ResponseStatus
3.DefaultHandlerExceptionResolver
Converts exceptions into HTTP status codes

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.lab6.exceptions* and a custom exception class that will be thrown if a participant id is not found in the database.

```
package com.awbd.lab7.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/participant/info/10.

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

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

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

10. Write an @ExceptionHandler method in ParticipantController class. Test http://localhost:8080/participant/info/10

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

11. Test ParticipantController with id 17, set expected status *not found*.

```
package com.awbd.lab7.controllers;

import ...


@SpringBootTest
@AutoConfigureMockMvc
public class ParticipantControllerITest {

    @Autowired
    MockMvc mockMvc;

    @Test
    public void showByIdMvc() throws Exception {


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

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

13. Test http://localhost:8080/product/info/8. What view is return by *ProducController*?
In order to handle the *ResorceNotFoundException* thrown in *ProductController*, 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("notfound");
        return modelAndView;
    }

}
```

14. Test http://localhost:8080/participant/info/abc. You will get a NumberFormatException.
Add com.awbd.lab6.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("error");
        r.setExceptionAttribute("ex");     // default "exception"

        return r;
    }

}
```

16. Map errors to view and status codes:

```
SimpleMappingExceptionResolver r =
        new SimpleMappingExceptionResolver();

Properties mappings = new Properties();
mappings.setProperty("NumberFormatException", "numberformaterr");
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 regullar expressions
• **@NotNull**	

17. Add dependecies for Java bean validation API:

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.2.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
    <version>6.0.2.Final</version>
</dependency>
```

18. Check that the minim price for a product is 100. Check that participant name is required.

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

```
@NotNull(message = "required field")
private String lastName;
```

19. Change saveOrUpdate method, add patameter BindingResult bindingResult.

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

    Product savedProduct = productService.save(product);
    imageService.saveImageFile(Long.valueOf(savedProduct.getId()),
file);
    //return "redirect:/product/info/" + savedProduct.getId();
    return "redirect:/product/list" ;
}
```

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