Programming 5

Testing - integration tests

Karel de Grote Hogeschool

- **Spring Boot Testing**
- **Testing a repository**
- **Testing a service**
- **Testing a controller**
- **Handling setup/teardown**

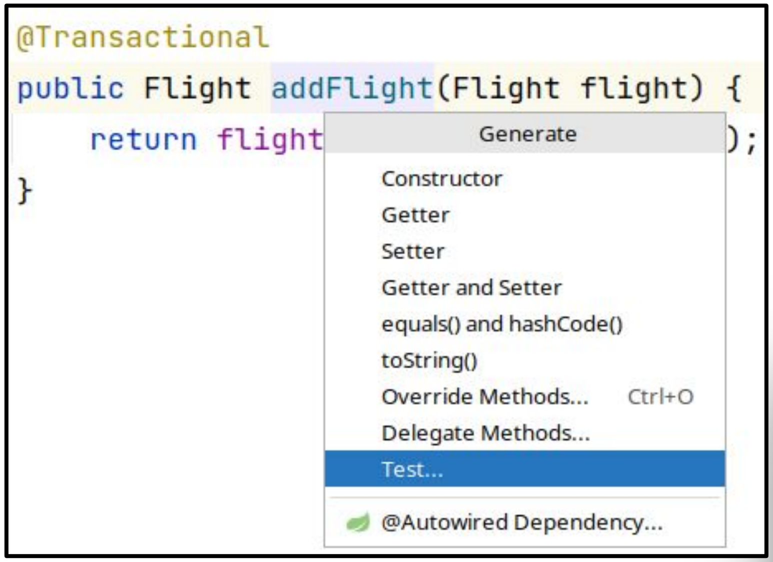
Spring Boot Testing

- In this course, when we test Spring components, we will initialize a Spring context for convenience

- **build.gradle**:

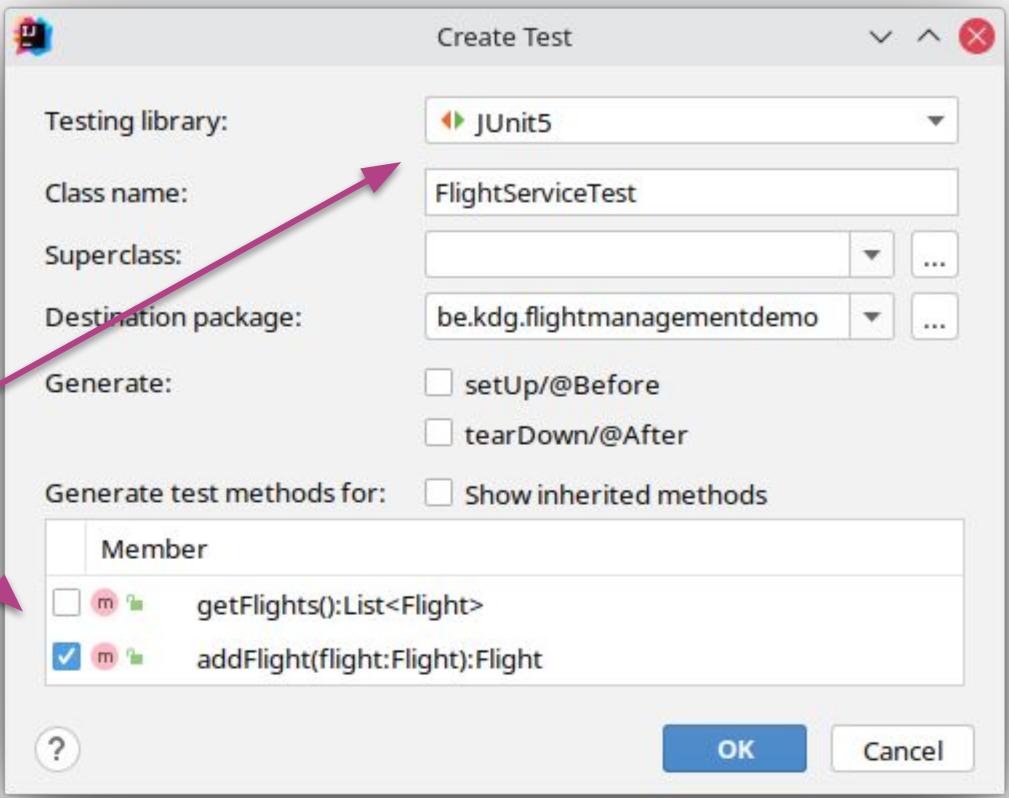
```
dependencies {
    ...
    testImplementation
            'org.springframework.boot:spring-boot-starter-test'
}

test {
    useJUnitPlatform()
}
```

Creating a test



```
@Transactional
public Flight addFlight(Flight flight) {
    return flight
}
```

Press Alt+Insert



Create Test

Testing library:	JUnit5
Class name:	FlightServiceTest
Superclass:	
Destination package:	be.kdg.flightmanagementdemo
Generate:	☐ setUp/@Before
	☐ tearDown/@After
Generate test methods for:	☐ Show inherited methods

	Member
☐	getFlights():List<Flight>
✓	addFlight(flight:Flight):Flight

JUnit 5 support is bundled with IntelliJ.
Selected the appropriate methods.

Creating a test

Add this annotation.

Causes the application context to be available for the duration of the test. (@Autowired, …)

```
@SpringBootTest
public class FlightServiceTest {

    // Code omitted

}
```

- Spring Boot Testing
- Testing a repository
- Testing a service
- Testing a controller
- Handling setup/teardown

Testing a repository

Useful:

- Testing JPA annotations
  - Owning side, cascading rules, uniqueness, …
- Testing custom queries (automatic queries, **`@Query`**, FetchType, …)
- Regression testing related to the database
  - DB provider version updates

Not useful:

- Usually tested as part of integration tests (service layer or presentation layer)
- Don’t write tests for an API (test only *your* code)

Testing a repository

Testing a JPA query or FetchType is more useful …

```
@SpringBootTest
class TrainRepositoryTest {
    @Autowired
    private TrainRepository trainRepository;

    @Test
    public void trainDateIsMandatory() {
        // Arrange
        var newTrain1 = new Train("TRAIN1", LocalDate.now());
        var newTrain2 = new Train("TRAIN2", null);

        // Act
        var createdTrain = trainRepository.save(newTrain1);

        // Assert
        assertTrue(createdTrain.getId() > 0);
        assertThrows(DataIntegrityViolationException.class,
                () -> trainRepository.save(newTrain2));
    }
}
```

- Spring Boot Testing
- Testing a repository
- Testing a service
- Testing a controller
- Handling setup/teardown

Testing a service

- Makes *a lot* of sense! (business logic)
- Can be an integration test
  - Test the service together with its dependencies (repositories, other services or components)
- Can be a unit test
  - Mock the service’s dependencies (see “mocking” later)

Testing a service

```java
@SpringBootTest
public class BookServiceTests {
    @Autowired
    private BookService bookService;

    @Autowired
    private AuthorRepository authorRepository;

    @Autowired
    private UserRepository userRepository;

    @Test
    public void deleteBookShouldOnlyDeleteThatBook() {
        // Arrange
        var amountOfBooks = bookService.getAllBooks().size();
        var amountOfAuthors = authorRepository.findAll().size();
        var amountOfUsers = userRepository.findAll().size();
        // Act
        bookService.deleteBook(bookId);
        // Assert
        assertEquals(amountOfBooks - 1, bookService.getAllBooks().size());
        assertEquals(amountOfAuthors, authorRepository.findAll().size());
        assertEquals(amountOfUsers, userRepository.findAll().size());
    }
}
```

Should be replaced by test-specific seeding!

- Spring Boot Testing
- Testing a repository
- Testing a service
- Testing a controller
- Handling setup/teardown

Testing a controller

- Makes *a lot* of sense!
    - REST API ‘contract’/‘interface’
        - URLs, status codes, content-negotiation, body, …
    - MVC
        - Displayed information, view information, …
- Can be integration tests
    - Test the controller together with its dependencies (services or other components)
- Can be tested as a unit
    - Mock the controller’s dependencies

Testing a controller

```
@AutoConfigureMockMvc
@SpringBootTest
class BookControllerTests {
    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private BookRepository bookRepository;

    @Test
    void allBooksShouldShowAllBooks() throws Exception {
        mockMvc. ... see the following slides ...
    }
}
```

Enable MockMvc with default settings. Allows autowiring of a **`MockMvc`** object.

Testing a controller

- MockMvc only mocks the MVC environment
- Currently, we **aren't** mocking any of our code (yet)
- We'll look into mocking custom code next week!

- Assertions will be covered by Hamcrest:
  - http://hamcrest.org

Testing a controller

- Test the view name
- Test if exactly six books are in the MVC model
    - Makes sense **only if** exactly six books were seeded as part of, for example, **`@BeforeAll`**

```
@Test
void allBooksShouldShowAllBooks() throws Exception {
    mockMvc.perform(get("/book/all"))
            .andExpect(view().name("books_list"))
            .andExpect(model().attribute(
                    "books", hasSize(6)));
}
```

Testing a controller

- Test the view name
- Test if the books are all books of the repository
  - Autowire the repository to pull this off
  - White-box testing

```java
@Test
void allBooksShouldShowAllBooks() throws Exception {
    var expectedBooks = bookRepository.findAll();
    mockMvc.perform(get("/book/all"))
        .andExpect(view().name("books_list"))
        .andExpect(model().attribute(
                    "books", equalTo(expectedBooks)));
}
```

Testing a controller

- Alternatively (more verbose):

```
@Test
void allBooksShouldShowAllBooks() throws Exception {
    var mvcResult = mockMvc.perform(get("/book/all"))
            .andExpect(view().name("books_list"))
            .andReturn();
    var actualBooks = (List<Book>) mvcResult
            .getModelAndView().getModel().get("books");
    var expectedBooks = bookRepository.findAll();
    assertEquals(6, actualBooks.size());
    assertEquals(expectedBooks, actualBooks);
    var actualBook = actualBooks.get(0);
    var expectedBook = expectedBooks.get(0);
    assertEquals(expectedBook, actualBook);
}
```

RequestBuilder

```
get("/url/...").accept( ... ). ...
```

RequestBuilder

- All verbs are supported
- Class: **`MockMvcRequestBuilders`**

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.delete;
                                                                                   ...
```

```
mockMvc.perform(get("/book/all"))
        .andExpect(/* ... */)

mockMvc.perform(post("/book/add"))
        .andExpect(/* ... */)

mockMvc.perform(delete("/api/books/{id}", "10"))
        .andExpect(/* ... */)
```

RequestBuilder

- Request parameters

```
mockMvc.perform(
                get("/book/details")
                        .queryParam("id", "10")
        )
        .andExpect(/* ... */)
```

- Path variables

```
mockMvc.perform(
                get("/api/books/{id}", "10")
        )
        .andExpect(/* ... */)
```

RequestBuilder

- Common HTTP headers (accept, contentType)

```
mockMvc.perform(
                get("/api/books")
                    .accept(MediaType.APPLICATION_JSON)
        )
        .andExpect(/* ... */)
```

RequestBuilder

- Less common HTTP headers

```
mockMvc.perform(
                get("/api/books")
                    .header("X-XSRF-TOKEN", "15...eb")
       )
       .andExpect(/* ... */)
```

ResultActions

andExpect, andReturn, …

ResultActions

- Assertions

```
mockMvc.perform(get("/book/all"))
        .andExpect(/* ... */)
```

- Take any action

```
mockMvc.perform(get("/book/all"))
        .andExpect(/* ... */)
        .andDo(print());
```

ResultActions

- Take full control of the result

```
var mvcResult = mockMvc.perform(get("/book/all"))
        .andExpect(/* ... */)
        .andReturn();

var actualBooks = (List<Book>) mvcResult
        .getModelAndView()...
```

ResultMatchers

view, model, status, jsonPath, …

ResultMatchers

- Check the view name

```
mockMvc.perform(get("/book/all"))
        .andExpect(view().name("books_list"))
```

- Check the contents of the model

```
mockMvc.perform(get("/book/all"))
        .andExpect(view().name("books_list"))
        .andExpect(model().attribute("books",
                          equalTo(expectedBooks)))
        .andExpect(model().attributeExists("username"))
        .andExpect(model().attributeHasErrors(
                          "password"));
```

ResultMatchers

- Check the status code

```
mockMvc.perform(get("/book/all"))
        .andExpect(status().isOk())
```

Don’t just test the happy path!

- Check header fields

```
mockMvc.perform(get("/book/all"))
      .andExpect(header().string(
                 HttpHeaders.CONTENT_TYPE,
                 MediaType.APPLICATION_JSON.toString()))
```

ResultMatchers

⚠️ Using *jsonPath* results in the most correct and complete test! (for an API) ⚠️

- Check the response body as JSON

```
mockMvc.perform(
          get("/api/books/{id}", "10")
                 .accept(MediaType.APPLICATION_JSON)
   )
   .andExpect(status().isOk())
   .andExpect(header().string(HttpHeaders.CONTENT_TYPE,
        MediaType.APPLICATION_JSON.toString()))
   .andExpect(jsonPath("$.title")
               .value("The Fellowship of the Ring"))
```

- JsonPath: https://github.com/json-path/JsonPath

ResultMatchers

- Check the response body as a string (not recommended!)

```
mockMvc.perform(
        get("/api/books/{id}", "10")
                .accept(MediaType.APPLICATION_JSON)
 )
 .andExpect(status().isOk())
 .andExpect(content().string(containsString("FANTASY")))
```

Can easily be replaced with a more precise **jsonPath** expression.

Disabling Security

… for MockMvc tests

Disabling Security for tests

- We will only disable security **temporarily**!
  - Testing for status 401 and 403 is important and will be added later.

```java
@AutoConfigureMockMvc(addFilters = false)
@SpringBootTest
class StationStopControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void deletingAStationStopShouldReturn404ForNonExistingStation()
            throws Exception {
        mockMvc.perform(delete("/api/stationstops/{id}", "999"))
                .andExpect(status().isNotFound());
    }
}
```

Disabling Security for tests

- We will only disable security **temporarily**!
    - Testing for status 401 and 403 is important and will be added later.

- Some common approaches:
    - Use **Spring Profiles** to disable your security configuration class **during tests**
    - Additional suggestions

- Spring Boot Testing
- Testing a repository
- Testing a service
- Testing a controller
- **Handling setup/teardown**

Using @BeforeAll or @AfterAll

- JUnit: Methods **must be static**
- Spring Test: Methods are **not allowed to be static**

More info Baeldung

```
@SpringBootTest
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class StationRepositoryTest {
    @Autowired
    private StationRepository stationRepository;

    @BeforeAll
    public void setup() {
        stationRepository.save(
                new Station("ANR", "Antwerp", null));
    }
// ...
```