



Java - Module 06

JUnit/Mockito

Summary:

In this module you will learn the basics of module and integration testing.

Version: 1.00

Contents

I	General Rules	2
II	Rules of the Day	3
III	Exercise 00: First Tests	4
IV	Exercise 01: Embedded DataBase	6
V	Exercise 02: Test for JDBC Repository	8
VI	Exercise 03: Test for Service	10

Chapter I

General Rules

- Use this page as the only reference. Do not listen to any rumors and speculations about how to prepare your solution.
- You must use the latest LTS version of Java. Make sure that compiler and interpreter of this version are installed on your machine.
- You must use both JVM and GraalVM to run your code.
- You can use IDE to write and debug the source code (we recommend IntelliJ Idea).
- The code is read more often than written. Read carefully the [document](#) where code formatting rules are given. When performing each exercise, make sure you follow the generally accepted [Oracle standards](#)
- Pay attention to the permissions of your files and directories.
- To be assessed, your solution must be in your GIT repository.
- You should not leave in your directory any other file than those explicitly specified by the exercise instructions. It is recommended that you modify your .gitignore to avoid accidents.
- When you need to get precise output in your programs, it is forbidden to display a precalculated output instead of performing the exercise correctly.
- Have a question? Ask your neighbor on the right. Otherwise, try with your neighbor on the left.
- Your reference manual: mates / Internet / Google. And one more thing. There's an answer to any question you may have on Stackoverflow. Learn how to ask questions correctly.
- Read the examples carefully. They may require things that are not otherwise specified in the subject.
- Use "System.out" for output
- And may the Force be with you!
- Never leave that till tomorrow which you can do today ;)


Chapter II

Rules of the Day

- Use JUnit 5 framework in all tasks
- Use the following dependencies and plugins to ensure correct operation:
 - maven-surefire-plugin
 - junit-jupiter-engine
 - junit-jupiter-params
 - junit-jupiter-api
- Use adequate names for testing methods.
- Consider different situations.
- Ensure that tests cover 100% of today's code (it should be at least 80% for big projects).
- Each test method should contain a small portion of code and be executed quickly.
- Test methods must be isolated from one another and have no side effects.
- All tests must be launchable by running `mvn clean compile test` command

Chapter III

Exercise 00: First Tests

	Exercise 00
First Tests	
Turn-in directory : <i>ex00/</i>	
Files to turn in : Tests-folder	
Allowed functions : All	

Now you need to implement `NumberWorker` class that contains the following functionality:

```
public boolean isPrime(int number) {  
    ...  
}
```

This method determines if a number is prime and returns true/false for all natural (positive integer) numbers. For negative numbers, as well as 0 and 1, the program shall throw an unchecked exception. `IllegalNumberException`.

```
public int digitsSum(int number) {  
    ...  
}
```

This method returns the sum of digits of a source number.

We also need to create `NumberWorkerTest` class that implements the module testing logic. Methods of `NumberWorkerTest` class shall check the correct operation of `NumberWorker` methods for various input data:

1. `isPrimeForPrimes` method to check `isPrime` using prime numbers (at least three)
2. `isPrimeForNotPrimes` method to check `isPrime` using composite numbers (at least three)
3. `isPrimeForIncorrectNumbers` method to check `isPrime` using incorrect numbers (at least three)
4. a method to check `digitsSum` using a set of at least 10 numbers

Requirements:

- `NumberWorkerTest` class must contain at least 4 methods to test `NumberWorker` functionality
- Use of `@ParameterizedTest` and `@ValueSource` is mandatory for methods 1-3.
- Use of `@ParameterizedTest` and `@CsvFileSource` is mandatory for method 4.
- You need to prepare `data.csv` file for method 4 where you shall specify at least 10 numbers and their correct sum of digits. A file content example:


```
$> cat data.csv
# number, digitsSum
1234, 10
$>
```

Project structure:

```
Tests
├── src
│   ├── main
│   │   ├── java
│   │   │   └── fr.42.numbers
│   │   │       └── NumberWorker
│   │   └── resources
│   └── test
│       ├── java
│       │   ├── fr.42.numbers
│       │   └── NumberWorkerTest
│       └── resources
│           └── data.csv
└── pom.xml
```

Chapter IV

Exercise 01: Embedded DataBase

	Exercise 01
Embedded DataBase	
Turn-in directory : <i>ex01/</i>	
Files to turn in : Chat-folder	
Allowed functions : all	

Do not use a heavy DBMS (like PostgreSQL) to implement integration testing of components that interact with the database. It is best to create a lightweight in-memory database with prearranged data.

Implement `DataSource` creation mechanism for HSQL DBMS. To do so, connect `spring-jdbc` and `hsqldb` dependencies to the project. Prepare `schema.sql` and `data.sql` files where you will describe product table structure and test data (at least five).

Product table structure:

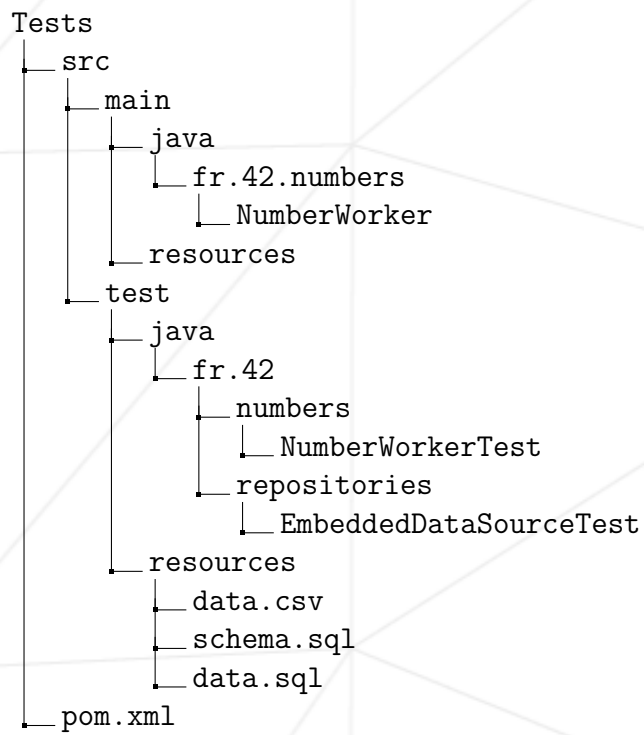
- identifier
- name
- price

Also create `EmbeddedDataSourceTest` class. In this class, implement `init()` method marked with `@BeforeEach` annotation.

In this class, implement functionality to create `DataSource` using `EmbeddedDataBaseBuilder` (a class in `spring-jdbc` library).


Implement a simple test method to check the return value of `getConnection()` method created by `DataSource` (this value must not be null).

Project structure:



Chapter V

Exercise 02: Test for JDBC Repository

	Exercise 02
Test for JDBC Repository	
Turn-in directory : <i>ex02/</i>	
Files to turn in : Tests	
Allowed functions : all	

Implement `ProductsRepository/ProductsRepositoryJdbcImpl` interface/class pair with the following methods:

```
List<Product> findAll();
Optional<Product> findById(Long id);
void update(Product product);
void save(Product product);
void delete(Long id);
```

You shall implement `ProductsRepositoryJdbcImplTest` class containing methods checking repository functionality using the in-memory database mentioned in the previous exercise. In this class, you should prepare in advance model objects that will be used for comparison in all tests.

Example of declaring test data is given below:

```
class ProductsRepositoryJdbcImplTest {
    final List<Product> EXPECTED_FIND_ALL_PRODUCTS = ...;
    final Product EXPECTED_FIND_BY_ID_PRODUCT = ...;
    final Product EXPECTED_UPDATED_PRODUCT = ...;
}
```



Each test shall be isolated from behavior of other tests. Thus, the database must be in its initial state before each test is run.




Test methods may call other methods that are not under the current test. For instance, when testing `update()` method, `findById()` method may be called to check the entity update validity in the database.

Project structure:



Chapter VI

Exercise 03: Test for Service

	Exercise 03
Test for Service	
Turn-in directory : <i>ex03/</i>	
Files to turn in : Tests	
Allowed functions : all	

An important rule for module tests: an individual system component shall be tested without calling its dependencies' functionality. This approach allows developers to create and test components independently, as well as postpone the implementation of specific application parts.

Now you need to implement the business logic layer represented by `UsersServiceImpl` class. This class contains a user authentication logic. It also has a dependency on `UsersRepository` interface (in this task, you do not need to implement this interface).

`UsersRepository` interface (that you have described) shall contain the following methods:

```
User findByLogin(String login);
void update(User user);
```

It is assumed that `findByLogin` method returns a `User` object found via login, or throws `EntityNotFoundException` if no user is found with the login specified. Update method throws a similar exception when updating a user that does not exist in the database.

`User` entity shall contain the following fields:

- Identifier
- Login
- Password
- Authentication success status (true - authenticated, false - not authenticated)

In turn, `UsersServiceImpl` class calls these methods inside the authentication function: `boolean authenticate(String login, String password)`

This method:

1. Checks if a user has been authenticated in the system using this login. If authentication was performed, `AlreadyAuthenticatedException` must be thrown.
2. The user with this login is retrieved from `UsersRepository`.
3. If the retrieved user password matches the specified password, the method sets the status of the authentication success for the user, updates its information in the database and returns true. If passwords mismatch, the method returns false.

Your goal is to:

1. Create `UsersRepository` interface
2. Create `UsersServiceImpl` class and `authenticate` method
3. Create a module test for `UsersServiceImpl` class

Since your objective is to check correct operation of `authenticate` method independently of `UsersRepository` component, you should use mock object and stubs of `findByLogin` and `update` methods (see Mockito library).

`Authenticate` method shall be checked for three cases:

1. Correct login/password (check calling `update` method using `verify` instruction of Mockito library)
2. Incorrect login
3. Incorrect password

Project structure (see next page):

