Spring Essentials Labs

Prerequisite :

- JDK 11+
- IDE (IntelliJIDEA, Eclipse, STS, VSCode)
- Lombok lib: https://projectlombok.org/downloads/lombok.jar
- Docker
- Git

Lab1 : Spring Core / First configuration

The first lab uses the example developed in the presentation.
We want to develop a business object which list all the films of a particular director. The films are stored in a persistent store (A simple text file for the moment).

The business object will implement the single following method :

```
public List<Movie> moviesDirectedBy(String director)
```

A Java interface specify the method that DAO objects should implement.

```
public interface MovieDAO {
	public List<Movie> findAll();
}
```

An implementation of this interface for a tabulated file respecting a particular format is provided (*org.formation.dao.FileDAO*).

Finally, a test class is also used to test the implementation of your business object.

1.1 XML Configuration

1. Set up the projet
Just import the Maven project provided

2. Implement org.formation.service.MovieLister
Implement the business method, constructors and/or setters allowing dependency injection.

3. Configure Spring
In the ***src/test/resources/test.xml*** file, declare the necessary beans and set their attributes.

4. Test
Execute as Junit Test *org.formation.service.MovieListerTest.*

1.2 Creation Cycle of a Bean

In ***MovieLister***, implement the interfaces *BeanNameAware, BeanClassLoaderAware, BeanFactoryAware, ResourceLoaderAware, ApplicationEventPublisherAware, MessageSourceAware, ApplicationContextAware, BeanPostProcessor, InitializingBean*
Just print a message on the console.

Observe the console while running the test

Lab2 Configuration with annotations

2.1 Configuration with annotations

Go on previous lab:

Add the dependency ***javax.annotation :javax.annotation-api :1.3.2*** in order to have the *@PostConstruct* and *@PreDestroy* annotations available

Implement a configuration equivalent to the previous lab but with annotations

2.2 Profiles

We want to run the test in 2 separate profiles ***"file"*** and **"jdbc"**

Add the following dependencies :

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-jdbc</artifactId>
	<version>5.3.13</version>
</dependency>
<dependency>
	<groupId>org.postgresql</groupId>
	<artifactId>postgresql</artifactId>
	<version>42.3.1</version>
</dependency>
```

Start a postgresql server with :
docker-compose -f postgres-docker-compose.yml up -d

Use pgAdmin : connect on localhost:81 with ***admin@admin.com/admin***
Declare the server with the following connection properties:
movies-postgresql postgres/postgres
Create a database ***movies*** and execute the initialization script provided

Retrieve the provided JDBC classes and annotate them correctly.

Rewrite the test class by defining 2 methods:

- 1 using the *file* profile
- the other the *jdbc* profile

Lab3 : Sample usage of Spring AOP

Objectives

We want to apply a cross-cuting concern to all the classes inside the *org.formation.service* package. The advice must print the execution time of each methods

We will use *@ApectJ*.

Work to be done

1. Set up the project

Add the following dependencies:

```
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-aop</artifactId>
	<version>5.3.13</version>
</dependency>
<dependency>
	<groupId>org.aspectj</groupId>
	<artifactId>aspectjweaver</artifactId>
	<version>1.8.13</version>
</dependency>
```

2. Configure Spring

Enable AOP via @AspectJ in config

3. Advice implementation

Declare a new bean « *performanceMonitor* »

In this bean, implement the advice which will log (with INFO Level) the execution of each method of *org.formation.service* package

4. Test

Execute the test class and observe the console

Lab4 : Getting started with SpringBoot

- Ceate a new Spring Project with the Wizard(New → SpringStarter Project)
  - Choose the starter module web
  - Execute, (*Run As → Spring Boot App*)
  - Access to localhost:8080
  - In *Run → Configurations*, override the property *server.port*
  - Access to the new URL

- Create a class with the following code :

```
@Component
@ConfigurationProperties("hello")
public class HelloProperties {

	/**
	 * Greeting message returned by the Hello Rest service.
	 */
	private String greeting = "Welcome ";

	public String getGreeting() {
		return greeting;
	}

	public void setGreeting(String greeting) {
		this.greeting = greeting;
	}
}
```

- Another class which implements a REST resource:

```
@RestController
public class HelloController {

	@Autowired
	HelloProperties props;

	@RequestMapping("/hello")
	public String hello(@RequestParam String name) {
		return props.getGreeting()+name;
	}
}
```

Aadd another starter module which handles the following annotation *@ConfigurationProperties* during build time:

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-configuration-
processor</artifactId>
		</dependency>
```

- Then test the completion in the property editor

- Tester the application

Auto configuration et Configuration de Debug

In the Main class, use the return value of `SpringApplication.run(Application.class, args);` to print the configured Spring beans.

Start the application from the command line after packaging it (*mvn package* or *gradlew bootJar*)

Add the dependency `spring-boot-configuration-processor`

and enable ***devtools***

Test automatic restart when changing Java code

External configuration

Transform *application.properties* into ***application.yml***

Set the following configuration properties:

- Random value in yml file assigned to controller variable. Display value in a controller endpoint
- In *HelloProperties* (Add dependency for validation):
  - *hello.greeting* : Not empty
  - *hello.styleCase* : Upper, Lower or Camel
  - *hello.position* : 0 or 1

Profil e

Set a different port for the profile *prod*

Activate the profile:

- Via the IDE
- Via the command line using the *fat jar*

DEBUG mode and trace configuration

Activate the ***-debug*** option at start-up

Modify the configuration in order to generate a log file

Modify the logger trace level *org.springframework.boot* to DEBUG without the option ***-debug***

Lab5 : SpringBoot and SpringData

5.1 Spring Data JPA

Auto configuration par défaut de Spring JPA

Create a project with :

- a dependency on ***Spring JPA***
- a dependency on ***hsqldb***

Get the model classes provided as well as the sql script.

The *Member* and *Document* classes are used for the JPA part, the *Customer* class for the NoSQL part

If necessary add Maven or Gradle dependency

Configure Hibernate to show executed SQL statements

Develop an ***import.sql*** file that inserts test data into the database

Start the application and verify that the insertions take place

Define *Repository* interface which allow the following methods :

- Find members with a particular email
- Find the member for a given email and password
- All members whose name contains a particular string
- Search all documents of a member from its name (Remember to use the @Query annotation)
- Count members
- Count documents
- Find a member from their ID with all associated documents pre-loaded

Modify the test class generated by SpringIntializr to verify that the methods perform the correct requests

Optional:

Inject an *EntityManager* or a *Datasource* to work directly at the JPA or JDBC level

Datasource Configuration

Add a dependency on the driver PostgresSQL

Define a postgres database using a pool of connections (maximum 10 connections) in a production profile.

Create a runtime configuration that enables this profile

Service implementation Service

Implement a transactional business method that allows you to add a document to all users of the database

Test the method

5.2 Repository MongoDB

Add the following dependencies :

- MongoDB
- Embedeed MongoDB embarqué

Use the Customer class :

```
public class Customer {

    @Id
    public String id;

    public String firstName;
    public String lastName;

    public Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return String.format(
                "Customer[id=%s, firstName='%s', lastName='%s']",
                id, firstName, lastName);
    }
}
```

Define a *MongoRepository i*nterface that allows searching for *Customer* classes by *firstName* or *lastName* attributes

Modify the test class to execute the following method

```
private void _playWithMongo() {
            customerRepository.deleteAll();

            // save a couple of customers
            customerRepository.save(new Customer("Alice", "Smith"));
            customerRepository.save(new Customer("Bob", "Smith"));

            // fetch all customers
            System.out.println("Customers found with findAll():");
            System.out.println("-------------------------------");
            for (Customer customer : customerRepository.findAll()) {
                  System.out.println(customer);
            }
            System.out.println();

            // fetch an individual customer
            System.out.println("Customer found with
findByFirstName('Alice'):");
            System.out.println("--------------------------------");

      System.out.println(customerRepository.findByFirstName("Alice"));

            System.out.println("Customers found with
findByLastName('Smith'):");
            System.out.println("--------------------------------");
            for (Customer customer :
customerRepository.findByLastName("Smith")) {
                  System.out.println(customer);
            }

      }
```

Lab6 : SpringBoot and SpringMVC

6.1 Application Web and Spring MVC (Optional)

Dependecies

On the previous project, add the following starters:

- web
- thymeleaf
- dev-tools

Add alsor the following ***webjar*** **dependencies**

```
<dependency>
     <groupId>org.webjars</groupId>
     <artifactId>bootstrap</artifactId>
     <version>4.4.1-1</version>
</dependency>
<dependency>
     <groupId>org.webjars</groupId>
     <artifactId>jquery</artifactId>
     <version>3.4.1</version>
</dependency>
```

Execute ***mvnw install*** and reload the login page

Retrieve the sources provided (static directory and templates to be placed in ***/src/main/resources***)

Configuration

Add an MVC configuration declaring the ***viewControllers***, allowing access to the pages present in the template directory: ***home.html*** and ***documents.html***

Controller

Implement member registration use case in MemberController which will :

- Respond to ***GET /web/register*** *and route to the* ***register*** *page present in templates*

- Respond to ***POST from /web/register*** which registers a member in the database and place it in the session attribute ***loggedUser*** then redirects to the documents page

6.2 API REST

6.2.1 Controllers

Create a REST controller ***MemberRestController*** which will :

- Perform all CRUD methods on a member
- Show members containing a particular string

Create a REST controller ***DocumentRestController*** which will :

- Retrieve all documents of a given member
- Add a document to a member

Some methods will be able to send business exceptions« *MemberNotFoundException* », « *DocumentNotFoundException* »

Disable the pattern « *Open Session in View* »

Test the URLs GET

6.2.2 Configuration

Configure CORS

Add a *ControllerAdvice* centralizing the exceptions handling

6.2.3 OpenAPI et Swagger

Add following dependency to get OpenApi 3.0 documentation

```
<dependency>
	<groupId>org.springdoc</groupId>
	<artifactId>springdoc-openapi-ui</artifactId>
	<version>1.5.9</version>
</dependency>
```

Access the description of our REST API (*http://server:port/swagger-ui.html*)

Add OpenAPI annotations to perfect documentation

6.3 REST calls

Create another project that implements a service implementing 3 methods making REST calls to the previous application:

- Load 1 member
- Load all members
- Create a member

Steps:

- Create a project with the web starter
- Disable tomcat startup with property:
  `spring.main.web-application-type=none`
- Implement a Service bean using ***RestTemplateBuilder*** and ***RestTemplate***
- Write the tests to make the calls

Lab7 : *SpringSecurity*

7.1 Configuration

Add Spring Security in dependencies of previous web project

Test application access

Enable debug traces for security

View the configured filter ***springSecurityFilterChain***, perform the authentication sequence and observe the messages on the console

Add a Configuration Class which extends `WebSecurityConfigurerAdapter` :

Override appropriate methodes to :

On the MVC web application

- Allows access to the home page
- Require authentication for all other pages
- On a successful logout, return to the home page

On the REST back-end

- Allow access to swagger documentation
- Requires authentication for API GET methods
- Requires ADMIN role for all other methods

7.2 Custom authentification

Set up a class implementing ***UserDetailService***

The class will be based on the *MemberRepository* bean developed in the previous labs

Also ensure that passwords are encrypted in the database.

7.3 OpenId / oAuth2 : Authentication and authorization

- Apply https://www.baeldung.com/spring-security-5-oauth2-login to log with an oAuth2 provider
- Set up a specific page
- In addition to the login proposal, add an authentication form, allowing authentication with the DB

Lab8 : Testing with SpringBoot

We use the same project and work in the ***src/test*** hierarchy

Each time the tests are set up, it is recommended to have the Spring application context (*ApplicationContext*) injected in order to inspect the configured beans.

8.1 Auto-configured tests

@DataJpaTest

Write a test class verifying the correct behaviour of the method *findByOwner*

@JsonTest

Write a test class verifying the correct behaviour of the serialization/deserialization of the *Member* class

@WebMVCTest

Use ***@WebMVCTest*** to:

- Test *MemberRestController* with a mockMVC
- Teste the home page with *HtmlUnit*

8.2 System tests (with security)

Use annotation ***SpringBootTest*** to create a web environment starting on a random port

Use ***@WithMockUser*** to simulate an authenticated user

Lab9 : Messaging

We set up a Publish/Subscribe type communication

ProductService publishes the TICKET_READY event to a topic

DeliveryService reacts by creating a delivery

9.1 Démarrage du Message Broker

Retrieve the docker-compose.yml file to start the Kafka message broker and the necessary ZooKeeper process

```
docker-compose up -d
```

Declare kafka in the host file

```
127.0.0.1 kafka
```

9.2 Producer

Add the starter ***spring-kafka*** to the project *ProductService*

Implement a Service bean that

- Create a ticket
- Publish a READY_TO_PICK event on the Kafka topic ***tickets-status***, the message key is the ***Ticket id***, the message body is an instance of the ***ChangeStatusEvent*** class

9.3 Consumer

Add the starter *spring-kafka* to the project *DeliveryService*

Implement a Service bean that

- Listen to the topic ***ticket-status***
- If event is READY_TO_PICK, create a Livraison

You can test everything with the provided JMeter script

Lab10 : Spring Data Réactive

10.1. Reactive Mongo DB

Start a Spring Starter Project and choose atarters ***reactive-mongodb*** et ***embedded Mongo***

Retrieve the provided model class

10.2 Repository classes

Create a Repository interface inheriting from `ReactiveCrudRepository<Account, String>`

Défine 2 reactive methods :

- A method to retrieve all Account classes via their value attribute
- A method to retrieve the first Account class via the owner attribute

Implement the CommandLineRunner interface in your main class and run code adding *Account* classes to the database, then performing the queries defined by the Repository class

2.2 ReactiveMongoTemplate

Create a configuration class creating a bean of type ***ReactiveMongoTemplate*** like following :

```
@Configuration
public class ReactiveMongoConfig {

    @Autowired
    MongoClient mongoClient;

    @Bean
    public ReactiveMongoTemplate reactiveMongoTemplate() {
        return new ReactiveMongoTemplate(mongoClient, "test");
    }
}
```

Then create a Service class exposing a business method for managing Account classes using the template.

Test via the main class or a test class.

Lab11 : Spring Web Flux

11.1. Controller approach

The objective of this part is to compare the performance and the scaling of the blocking model with respect to the non-blocking model.

Retrieve the provided web project (blocking model)

Create a Spring Starter Project and choose the ***web-reactive*** starter

Implement a controller equivalent to that of the blocking model.

Using the provided JMeter script, perform shots with the following parameters: NB_USERS=100, PAUSE=1000

At the end of the result, note:

- The test execution time
- The throughput

Perform multiple shots by increasing the number of users

11.2 Functional endpoints

The objective is to offer a Rest API for the management of the Mongo database of the previous TP

Resume the previous lab and add the WebReactif starter and delete the old main class

Create a Handler class grouping the methods allowing to define the following HandlerFunctions :

- "GET /accounts": Retrieve all accounts
- "GET /accounts/{id}": Retrieve an account by an id
- "POST /accounts": Create an account

Create the WebFlux configuration class declaring the endpoints of our application.

Use the provided JMeter script to test your implementation.

Lab12: Towards Production

12.1 Actuator

Add dependency to Actuator,

View default endpoints

Configure the application so that all endpoints are activated and accessible

View endpoints

Set up a production profile:

- Using Postgres database
- Choice for the security profile
- Disabling swagger-ui

12.2 Artefact for production

Modify the build so that build information is added

Configure the build so that the generated artefact is executable and has the project name

Start the app and access actuator endpoints

Optionally, build a Docker image

Optionally deploy to Heroku