Labs : Spring and SpringBoot

Requirements

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

Lab1 : Spring, First configuration

The first practical lab uses the example presented in the slides.
We want to develop a business service able to list all the films of a specific director. The films being stored in a repository (A simple text file for the moment).

Therefore, the service will expose the following method:

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

It can rely on a repository interface which allow to retreive all the movies store in a particular repository.

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

An implementation of this interface for a csv file (*org.formation.dao.FileDAO*).

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

1.1 XML Configuration

1. Set up the project
Import the provided Maven project.

2. Implement org.formation.service.MovieLister
Implement the business methid and either the constructor or the setters allowing dependency injection.

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

4. Execute the test
Execute the test ***org.formation.service.MovieListerTest*** to validate your implementation

Lab2. Configuration via annotations

2.1 Configuration via annotations

Start from the previous lab

Add the dependency ***javax.annotation :javax.annotation-api :1.3.2*** to use *@PostConstruct* and *@PreDestroy* annotations

Perform a configuration equivalent to the previous lab only via annotations.

In particular, you will use:

- *@Configuration* and stereotype annotations
- The location of the file containing the movies will be externalized in a *application.properties file*

2.2 Profiles

We want to run the test in 2 separate profiles "***csv***" 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 DB server and a pgAdmin instance via docker-compose :
docker-compose -f postgres-docker-compose.yml up -d

Connect to *pgAdmin* (localhost:81) with ***admin@admin.com/admin***
Declarer the DB server with the following connection properties :
movies-postgresql postgres/postgres
Create a database named ***movies*** and execute the provided initialisation script.

Retrieve the provided JDBC classes and annotate them correctly.

Rewrite the test class by defining 2 methods:
- 1 performing the test in the *csv* profile
- the other in the *jdbc* profile

Lab3 : Getting started with SpringBoot

- Create a Spring project (*New → SpringStarter Project)*
  - Add the starter web
  - Execute the application, (*Run As → Spring Boot App*)
  - Access to ***http://localhost:8080***
  - In *Run → Configurations*, override the properties *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 implementing a REST endpoint:

```
@RestController
public class HelloController {

	@Autowired
	HelloProperties props;

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

Add a dependency in the pom.xml to process the *@ConfigurationProperties* annotation during the build phase and create the necessary files so that Spring Tool takes into account the new properties:

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

- •Then test the completion in the property editor

- • Test the application

Auto configuration

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

Plug-in Maven

Start the application from the command line after performing a build (*mvn package* or *gradlew bootJar*)

Development starters

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

Activate ***devtools***

Observe the consequences of devtools: Automatic restart when changing Java code

Properties via le format .yml

Rename *application.properties* to *application.yml*

Set the following application properties, these properties are encapsulated in the configuration properties class *HelloProperties* :

- ◦ *hello.greeting* (not empty without default value) : The way to say *hello*
- ◦ *hello.styleCase* (Upper, Lower or Camel) : The way to write the name
- ◦ *hello.position* (0 ou 1) : The name or the greeting in first position

Add validation constraints in *HelloProperties*

Profiles

Define a different port for the prod profile

Activate Profile:

- Via the IDE
- Via the CLI, after having generated the *fat jar*

DEBUG mode and trace configuration

Enable the ***-debug*** option on boot

Modify the configuration in order to generate a server log file

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

Lab4 : SpringData

4.1 Spring Data JPA

Default configuration of Spring JPA

Create a project with

- a dependency on the starter ***Spring JPA***
- a dependency on ***H2***

Retrieve the provided model classes.

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 SQL statements

Retrieve the supplied ***import.sql*** script and place it correctly in the project hierarchy so that it inserts test data into the database

Start the application and verify that insertions take place

Interfaces JpaRepository

Define Repository interfaces that implement the following functionalities:

- CRUD on Member and Documents
- Search all documents
- Find members with a particular email
- Find the member for a given email and password
- All members whose first or last 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

Tests

Use the provided test class to validate queries for the *Member* entity.

Write on the same model a test class validating queries on Document

Optional:

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

Datasource configuration and connexion pool

Add a dependency on the PostgresQL driver

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

Create a runtime configuration that enables this profil.

Service implementation

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

Test the method

Lab4B (Optional) : MongoDB

Add dependency on MongoDB and on Mongo embedeed server

Declare a model class as follows:

```
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 interface that allows searching for *Customer* by ***firstName*** or ***lastName*** attributes

Write a test class which will contain the following code:

```
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);
		}

	}
```

Lab5 : Restful with SpringBoot and SpringMVC

5.1 Controllers

Create a Rest controller, *MemberRestController*, to:

- Display all members
- To display members whose full name (first name + last name) containing a particular string
- Perform all CRUD methods on a member
- Retrieve all documents of a given member
- To add a document to a member

When returning a list of members the returned json will only contain simple Member's attributes

When returning a single member, the json will also contain the associated documents.

Some methods may throw "*MemberNotFoundException*" business exceptions

Disable "Open Session in View" Pattern

Test GET URLs

5.2 CORS and Exceptions handling

Configure the CORS, to allow requests from all origins

Add a *ControllerAdvice* which centralize exception handling *MemberNotFoundException*

5.3 OpenAPI and Swagger

Add following dependency to get OpenApi 3.0 documentation

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

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

Add OpenAPI annotations to perfect documentation

Lab6 : RESTful call

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

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

Steps :
Create a project with the reactive-web starter
Disable tomcat startup with property:
spring.main.web-application-type=none

Implement a Service bean using ***WebClient.Builder*** and ***WebClient***
Write the tests to make the calls

Lab7 : Messaging

We set up a Publish/Subscribe type communication

ProductService publishes the TICKET_READY event to a topic

DeliveryService reacts by creating a delivery in its own database

7.1 Starting the 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 local host file (/etc/hosts)

```
127.0.0.1 kafka
```

7.2 Producer

Add ***spring-kafka*** starters on *ProductService*

Implement a service which :

- Creates a ticket
- Publishes an event *READY_TO_PICK* on the kafka topic ***tickets-status***, the key of the message is the Ticket id and the body an instance of *ChangeStatusEvent*

Publish some messages by creating ticket

7.3 Consumer

Add ***spring-kafka*** starters on *DeliveryService*

Implement a Service which

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

You can test everything with the provided JMeter script

Lab8 : *SpringSecurity*

8.1 Configuration

Add Spring Security in dependencies of previous web project

Test application access

Enable debug traces for security

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

Add a configuration class of `WebSecurityConfigurerAdapter whœich :`

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

8.2 Memory realm

Define in-memory users to verify previous ACL rules

What filters are enabled in the filter chain?

Enable debugging and make requestss

8.3 Custom authentication

Set up a class implementing *UserDetailService*, configure authentication to use this class.

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

Also ensure that passwords are encrypted in the database.

8.4 (Optional) Authentification via OpenId/oAuth2

- Apply https://www.baeldung.com/spring-security-5-oauth2-login to our project
- Set up a specific page

  In addition to the login proposal, add an authentication form, allowing authentication with the DB

8.5 (Optional) KeyCloak integration

1. Set up the token provider : KeyCloak

Start a KeyCloak server via Docker:

```
docker run -e KEYCLOAK_USER=admin -e KEYCLOAK_PASSWORD=admin -p
8089:8080 --name keycloak jboss/keycloak
```

- Connect to its administration console with admin/admin
- Create a Realm ***MemberRealm***
- Under this realm, create one client : ***member-app***
- Set access-type to confidential and Valid Redirect Uri to *
- Then create a user with credentials **user/secret**

Obtaining tokens:

Grant type *password* for the client *store-app*

```
curl -XPOST http://localhost:8089/auth/realms/MemberRealm/protocol/openid-
connect/token -d grant_type=password -d client_id=member-app -d
client_secret=<client_secret> -d username=user -d password=secret
```

Get a refresh token using a POST request with:

```
client_id:member-app
client_secret:<client_secret>
'refresh_token': refresh_token_requete_précédente,
grant_type:refresh_token
```

2. OAuth2 login et sécurité stateful

Ajouter les starters ***`security`*** ***et*** ***`oauth2-client`*** au projet Members

Configurer la sécurité en indiquant le oAuth2Login()

Exemple code Spring WebFlux :

```
@Bean
public SecurityWebFilterChain securitygWebFilterChain(
  ServerHttpSecurity http) {
    return http.authorizeExchange()
            .pathMatchers("/actuator/**").permitAll()
            .pathMatchers("/auth/**").permitAll()
      .anyExchange().authenticated()
      .and().
      oauth2Login().csrf().disable().build();
}
```

Configure Keycloak's oauth2 client as follows:

```
spring :
  security:
    oauth2:
      client:
        provider:
          keycloak:
            token-uri:
http://localhost:8089/auth/realms/MemberRealm/protocol/openid-connect/token
            authorization-uri:
http://localhost:8089/auth/realms/MemberRealm/protocol/openid-connect/auth
            user-info-uri:
http://localhost:8089/auth/realms/MemberRealm/protocol/openid-connect/userinfo
            user-name-attribute: preferred_username
        registration:
          member-app:
            provider: keycloak
            client-id: store-app
            client-secret: 57abb4f6-5130-4c73-9545-6d377dd947cf
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
```

Then access the API via a browser

To better understand, you can see the filter configured in the console and also augment the traces via:

```
logging:
  level:
    org.springframework.security: debug
```

3. ACL and resource server

Add the starter `oauth2-resource-server`

Specify `spring.security.oauth2.resourceserver.jwr.issuer-uri`

Update the security configuration to check ACL via le the token

Here is a sample

```
    @Bean
    public SecurityWebFilterChain securitygWebFilterChain(
      ServerHttpSecurity http) {
        return http.authorizeExchange()
                .pathMatchers("/actuator/**").permitAll()
                .pathMatchers("/auth/**").permitAll()
            .anyExchange().authenticated()
            .and().
            oauth2ResourceServer(v -> v.jwt()).csrf().disable().build();
    }
```

Test your configuration with the provided JMeter scenario

Lab 9 : SpringBootTest and auto-configured tests

For each test, it is recommended to print the Spring application context (ApplicationContext) in order to inspect the auto-configured beans.

@DataJpaTest

Write a test class which tests the *findByOwner* method

@JsonTest

Write a test class which tests the serialization/deserialization of the Member class

@WebMVCTest

Use ***@WebMVCTest*** to test MemberRestController using a mockMVC

Lab10: Toward Production

10.1 Actuator

Add a dependency to *Actuator*,

View default endpoints

Configure the application so that all endpoints are activated and accessible

View endpoints

10.2 Production profile

Set up a production profile:

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

10.3 Artefact for production

Modify the build so that build information is added

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

Start the app and access actuator endpoints

Optionally build a Docker image and boot it from Docker

Lab : Reactor

1.1 Reactor API

Create a project with the following dependencies

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>{reactor-version}</version>
</dependency>

            <dependency>
                  <groupId>ch.qos.logback</groupId>
                  <artifactId>logback-classic</artifactId>
                  <version>{latest}</version>
            </dependency>
```

A Main class

1.1.1 Method 1: Lambda Expression

In a first method,

- create a Feed that publishes 10 integers,
- a subscriber as a lambda expression that adds each event to a list, use the log() operator on the stream to see the sequence of events.

1.1.2 Method 2: Subscriber class

Get the same result using an inner Subscriber class

1.1.3 Method 3: Back pressure

Then modify the Subscriber so that it requests the events 2 by 2

1.1.4 Operator chaining

Use the following operator string:

- Multiply each integer by 3
- Filter odd numbers
- For each integer, generate its value and its opposite
- Logger
- Make the sum

Retrieve the result and display it.

1.1.5 Flow Combination

Combine 2 integer streams with zipWith

1.1.6 Timing

Generate a stream generating a counter every second, display the first 5 elements

1.2 Threads

Take one of the previous examples and use the subscribeOn method on a Scheduler. Observe the effects on the logs.

Add in the chain (in the middle of the chain) a call to publishOn
Observe the effects on the logs.

1.3 Gestion des erreurs

Go back to the previous examples.

Adding to streams produces an error event

1.3.1 Default behavior
Observe the default behavior.

1.3.2 onError method
Then implement the onError() method in the subscriber

1.3.3 onResume method
Finally, use an onErrorResume method to generate a Fallback Flow in case of an error

1.3.4 Retry
Give it a chance by giving it another try!

1.4 Test et debug

Ajouter les dépendances ***JUnit5*** et ***reactor-test***
Écrire une classe de test vérifiant la séquence d'événements d'une des méthode précédente.

Mettre en place le debug et ré-exécuter les exemples précédents avec les événements d'erreur

1.5 Contexte

Écrire une pipeline :

- générant 10 entiers
- les transformant en 10 String, en préfixant chaque entier par une valeur lue dans le contexte

Utiliser un pool de threads pour exécuter 5 fois cette pipeline
Vérifier que la valeur lue dans le contexte et bien propagée

Lab : Spring Data Réactive

2.1 Reactive Mongo DB

Start a Spring Starter Project and choose ***reactive-mongodb***, ***embedded Mongo*** and lombok starters

Retrieve the provided model class

2.1.1 Class Repository

Create a Repository interface inheriting from ***ReactiveMongoRepository<Account, String>***

Define 2 responsive methods:

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

Implement a test class that checks the addition of *Account* instances in the database, as well as the queries defined by the Repository class

2.1.2 ReactiveMongoTemplate

Create a configuration class creating a bean of type ***ReactiveMongoTemplate*** as follows:

```
@Configuration
public class ReactiveMongoConfig {

    @Autowired
    MongoClient mongoClient;

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

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

Write a test class of this Service from the main class.

2.2 R2DBC

Start a Spring Starter Project and choose ***SpringData-r2dbc*** and H2 starters

Take the Account model class from the previous workshop and modify it if necessary.

Also copy the database initialization file src/main/resources/schema.sql

2.2.1 Repository class

Add a bean to create Connections to the database:

```
        @Bean
    ConnectionFactoryInitializer initializer(ConnectionFactory connectionFactory) {

        ConnectionFactoryInitializer initializer = new ConnectionFactoryInitializer();
        initializer.setConnectionFactory(connectionFactory);
        initializer.setDatabasePopulator(new ResourceDatabasePopulator(new
ClassPathResource("schema.sql")));

        return initializer;
    }
```

Create a Repository interface inheriting from ***ReactiveCrudRepository<Account, Long>***

Define 2 responsive methods:

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

Implement a test class that checks the addition of Account instances in the database, as well as the queries defined by the Repository class

2.2.2 R2DBCEntityTemplate

Create a Service bean exposing a business interface for managing Account classes and using a *R2DBCEntityTemplate*.

Service methods will expose:

- `public Mono<Account> findById(String id) ;`
- `public Flux<Account> findAll() ;`
- `public Mono<Account> save(Account account) ;`
- `public Mono<Account> first() ;`

Implement a test class that tests these methods.

Lab : Spring Web Flux

3.1. Approche Contrôleur

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 flow

Perform multiple shots by increasing the number of users.

Observe Threads

3.2 Functional endpoints

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

Take the previous lab and add the WebReactif starter

Create a Handler class grouping the methods 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.

Lab : WebClient

4.1. WebClient

Create a new SpringBoot application with the webflux starter

Modify the main class main method so that it does not start a web server:

```
SpringApplication app = new SpringApplication(WebclientApplication.class);
// prevent SpringBoot from starting a web server
app.setWebApplicationType(WebApplicationType.NONE);
app.run(args);
```

Write a Webclient that uses the REST service from the previous lab.

The client will make the following requests:

- Creation of a new Account
- Retrieval of the account via its ID
- Retrieval of the first element returned by the list of all accounts