133 – Réaliser des applications web en session-handling

Rapport personnel

Version 1 du 24.03.25

Créé le 17.03.25

Blanchard Yann

|  |  |
| --- | --- |
|  | Module du 17.03.2025 au 15.04.2025 |

Table des matières

[1 Introduction 4](#_Toc193708734)

[2 Tests technologiques selon les exercices 5](#_Toc193708735)

[2.1 Installation et Hello World 5](#_Toc193708736)

[2.1.1 Prérequis 5](#_Toc193708737)

[2.1.2 Extensions VS code 5](#_Toc193708738)

[2.1.3 Test Hello world 5](#_Toc193708739)

[2.1.4 Observation 6](#_Toc193708740)

[2.2 Conteneurisation 7](#_Toc193708741)

[2.2.1 Réalisation 7](#_Toc193708742)

[2.2.2 Observation 7](#_Toc193708743)

[2.3 Création d'un projet Spring Boot 8](#_Toc193708744)

[2.3.1 Réalisation 8](#_Toc193708745)

[2.3.2 Observation 8](#_Toc193708746)

[2.4 Connexion à la DB JDBC 9](#_Toc193708747)

[2.4.1 Réalisation 9](#_Toc193708748)

[2.4.2 Observation 9](#_Toc193708749)

[2.5 Connexion à la DB JPA 9](#_Toc193708750)

[2.5.1 Réalisation 9](#_Toc193708751)

[2.5.2 Observation 9](#_Toc193708752)

[2.6 Connexion à la DB JPA avec DTO 11](#_Toc193708753)

[2.6.1 Réalisation 11](#_Toc193708754)

[2.6.2 Observation 11](#_Toc193708755)

[2.7 Gestion des sessions 12](#_Toc193708756)

[2.7.1 Réalisation 12](#_Toc193708757)

[2.7.2 Observation 13](#_Toc193708758)

[2.8 Documentation API avec Swagger 15](#_Toc193708759)

[2.8.1 Réalisation 15](#_Toc193708760)

[2.8.2 Observation 16](#_Toc193708761)

[2.9 Hébergement 16](#_Toc193708762)

[2.9.1 Réalisation 16](#_Toc193708763)

[3 Auto-évaluations et conclusions 18](#_Toc193708764)



# Introduction

Côté client

* La liaison au métier par protocole http (XML et JSON)
* HTML5-JS o Pattern MVC
* Contrôle des évènements produits par l’utilisateur

Côté serveur

* Métier en JAVA (servlets et JSP)
* Création et utilisation de WebServices o L’application Web en JSP
* Pattern MVC
* Contrôle des évènements produits par les appels des clients
* Connexion au serveur MySQL

En global

* Hébergement APACHE et TOMCAT
* Documentation et JAVADocTests

# technologiques selon les exercices

## Installation et Hello World

### Prérequis

Il faut installer WSL ainsi que docker via les liens suivants :

* <https://learn.microsoft.com/fr-fr/windows/wsl/install>
* <https://docs.docker.com/desktop/wsl/#download>

### Extensions VS code

Pour que tous fonctionnent correctement il faut que les extensions suivantes soit installé et fonctionnelle.

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.Une image contenant texte, capture d’écran, Police, logo

Le contenu généré par l’IA peut être incorrect.Une image contenant texte, capture d’écran, Police, Graphique

Le contenu généré par l’IA peut être incorrect.

### Test Hello world

Pour créer une première application java, nous allons utiliser l’exemple Spring.

Il faut donc cloner le projet disponible sur GitHub : <https://spring.io/guides/gs/guides-with-vscode>

Ensuite depuis le terminal sur le WSL il faut faire les commandes suivantes :

sudo apt install openjdk-17-jdk -y

echo 'export JAVA\_HOME=/usr/lib/jvm/java-17-openjdk-amd64' >> ~/.bashrc

source ~/.bashrc

echo $JAVA\_HOME

S’il y a une erreur lors de l’exécution de la première ligne il faut faire la commande suivante, cela peut régler le problème

sudo apt update

Cela a permis d’installer le Jdk 17, une fois cela terminé il faut lancer le programme. Si tout se passe bien 2 pages web devrait être accessible à ces endroits :

* [localhost:8080](http://localhost:8080/)
* [localhost:8080/greeting](http://localhost:8080/greeting)

### Observation

**Observez la console pour comprendre comment le projet est lancé et comment il tourne ?**

L'application Spring Boot démarre avec Java 17.

Elle utilise Tomcat 10.1.24 comme serveur web embarqué.

**C'est quoi le build et le run de Java ? Quel outil a-t-on utiliser pour build le projet ?**

Elle écoute sur le port 8080 et utilise le profil "default".

**Y a-t-il un serveur web ?**

Oui, un serveur Tomcat est intégré à l'application.

**Quelle version de java est utilisée ?**

Java 17.0.14

**Si il y a un serveur web, quelle version utilise-t-il ?**

Tomcat 10.1.24.

## Conteneurisation

### Réalisation

Tout d’abord il faut ajouter son fichier docker à son projet, pour ce faire il faut faire CTRL + SHIFT + P puis rechercher Add Docker Files to Workspace

Ensuite il faut choisir java, le fichier pom.xml et le port 8080

Le Dockerfile devrait ressembler à cela. Si c’est le cas il faut dans le terminal se déplacer dans complete et taper les commandes suivantes :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

cd complete

sudo apt install maven

mvn clean package

Une fois cela effectué il faut dans docker faire un click droit dessus et sélectionner run

Une image contenant texte, capture d’écran, logiciel, Police

Le contenu généré par l’IA peut être incorrect.

### Observation

**Pourquoi faire un container pour une application Java ?**

* Permet de standardiser l’environnement d’exécution (même JDK, mêmes dépendances).
* Facilite le déploiement et l’évolutivité sur différents environnements
* Isolation du processus pour éviter les conflits avec d’autres applications.
* Simplifie la gestion des dépendances et des configurations.

**Y a-t-il un serveur web ? Où se trouve-t-il ?**

Oui, l’application utilise un serveur Tomcat embarqué dans Spring Boot.

Il est inclus dans le .jar de l’application et s’exécute à l’intérieur du container Docker.

Le serveur écoute sur le port 8080 dans le container et est exposé pour être accessible depuis l’extérieur.

**À quoi faut-il faire attention (pensez aux versions !) ?**

* Vérifier que la version du JDK utilisée dans le Dockerfile (openjdk:17-jdk-alpine) est bien compatible avec l’application.
* S’assurer que la version de Spring Boot fonctionne correctement avec Tomcat.
* Toujours builder le projet avant de copier le .jar dans le container.
* Vérifier que le port 8080 est bien exposé pour éviter les problèmes d’accès.

## Création d'un projet Spring Boot

### Réalisation

Il faut créer un projet en faisant CTRL + SHIFT + P et écrire Spring Initializr

Une image contenant texte, capture d’écran, Police, Bleu électrique

Le contenu généré par l’IA peut être incorrect.

Ensuite il faut ajouter Spring Web et Lombok

Pour finir il faut créer un contrôleur avec les méthodes permettant le get, post, etc, puis créer le dockerfile afin de mettre sur son docker.

Une fois cela terminé il faut vérifier si ses méthodes sont accessibles via postman.

### Observation

* @RestController

🡪 Indique que la classe est un contrôleur Spring qui gère des requêtes HTTP et retourne des données JSON.

* @GetMapping("/getExample")

🡪Associe la méthode à une requête GET sur l’URL /getExample.

* @PostMapping("/postExample")

🡪Associe la méthode à une requête POST sur l’URL /postExample.

* @PutMapping("/putExample")

🡪Associe la méthode à une requête PUT sur l’URL /putExample.

* @RequestParam

🡪Permet de récupérer un paramètre de requête dans l’URL (?param=valeur).

* @RequestBody

🡪Indique que le corps de la requête doit être converti en objet Java (ici, String).

## Connexion à la DB JDBC

### Réalisation

Tout d’abord il faut créer un container de son docker puis le démarrer :

#Création du répertoire sur la machine locale qui contiendra les données de MySQL

mkdir -p /opt/mysql

#Démarrage du container MySQL

docker run --name mysql -d -p 3308:3306 -e MYSQL\_ROOT\_HOST=% -e MYSQL\_ROOT\_PASSWORD=emf123 -v /opt/mysql:/var/lib/mysql mysql/mysql-server:8.0

Ensuite il faut récupérer le WrkDB et modifié les informations concernant la db ainsi qu’importé les tables dans sa db

Ensuite il faut tester via posteman si les datas sont bien récupérés

### Observation

Il est possible de réaliser les requêtes en passant par le local host ou par le container. Le port a été modifié afin de contrer des soucis de port déjà écouté.

## Connexion à la DB JPA

### Réalisation

Pour cet exercice il fallait modifier un fichier précis.

Voici la configuration du fichier :

spring.application.name=ex5

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.properties.hibernate.format\_sql=true

spring.jpa.properties.hibernate.jdbc.batch\_size=20

spring.datasource.url=jdbc:mysql://localhost:3308/133ex5

spring.datasource.username=root

spring.datasource.password=emf123

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.hikari.maximum-pool-size=10

spring.datasource.hikari.minimum-idle=2

spring.jpa.properties.hibernate.cache.use\_second\_level\_cache=false

spring.jpa.properties.hibernate.cache.use\_query\_cache=false

logging.level.org.hibernate.SQL=DEBUG

logging.level.org.hibernate.orm.jdbc.bind=TRACE

### Observation

* @Autowired

🡪 Permet à Spring d’injecter automatiquement un composant (comme un Repository) dans une classe sans avoir à l’instancier manuellement.

* @ManyToOne

🡪 Indique une relation entre plusieurs entités de type source (par exemple Skieur) et une seule entité cible (comme Pays). Utilisée pour définir une relation en base de données.

* fetch = FetchType.EAGER

🡪Spécifie que l'entité liée (ici, le pays) doit être chargée immédiatement avec l'entité principale (le skieur).

* fetch = FetchType.LAZY

🡪Spécifie que l'entité liée ne sera chargée que lorsqu’elle est explicitement demandée. Cela permet d’économiser des ressources, mais peut provoquer une erreur si l’accès se fait hors d’un contexte transactionnel.

## Connexion à la DB JPA avec DTO

### Réalisation

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.Il suffit juste de mettre en place la structure :

### Observation

* **Model**

🡪 Représente les données (ex. Skieur avec nom, âge, etc.).

* **Repository**

🡪Sert à accéder à la base de données (ex. SkieurRepository pour CRUD).

* **DTO** (**Data Transfer Object)**

🡪 Sert à transférer uniquement les données utiles sans exposer toute l'entité.

* **Service**

🡪 Contient la logique métier, utilise les repositories pour manipuler les données.

* **Contrôleur**

🡪 Reçoit les requêtes, appelle les services et renvoie les réponses.

Le pays est juste un champ simple dans Skieur, il n'est pas manipulé séparément, donc pas besoin de créer un PaysDTO.

## Gestion des sessions

### Réalisation

Cet exercice est déjà donné, le but est de comprendre le fonctionnement. Nous n’avons donc rien eu a faire à par le mettre sur notre docker puis tester via postman

package com.example.ex6.controller;

import jakarta.servlet.http.HttpSession;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/user")

public class UserController {

    @PostMapping("/login")

    public ResponseEntity<String> login(HttpSession session, @RequestParam String username, @RequestParam String password) {

        // Ici, vérifiez les identifiants de l'utilisateur

        // Ceci est juste un exemple simple, ne stockez jamais le mot de passe en clair comme ceci dans la session

        if ("user".equals(username) && "pass".equals(password)) {

            session.setAttribute("username", username);

            session.setAttribute("visites", 0);

            return ResponseEntity.ok("Connecté");

        } else {

            return ResponseEntity.badRequest().body("Échec de la connexion");

    @asdfas   }

    }

    @GetMapping("/visites")

    public ResponseEntity<String> visites(HttpSession session) {

        if (session.getAttribute("username") != null) {

            Integer visites = (Integer) session.getAttribute("visites");

            visites++;

            session.setAttribute("visites", visites);

            return ResponseEntity.ok("Nombre de visites: " + visites);

        } else {

            return ResponseEntity.badRequest().body("Non connecté");

        }

    }

    @PostMapping("/logout")

    public ResponseEntity<String> logout(HttpSession session) {

        session.invalidate();

        return ResponseEntity.ok("Déconnecté");

    }

}

### Observation

Utilisation de la méthode setAttribute pour garder des informations sous format clé valeur. Cela permet de garder un username, un id, un nombre de visite par exemple.

 session.setAttribute("username", username);

 session.setAttribute("visites", 0);

Ici nous nous connectons grâce au login ce qui va créer le champ visites cotés client ainsi que le statu de connecté.

Une image contenant texte, capture d’écran, logiciel, nombre

Le contenu généré par l’IA peut être incorrect.

Ensuite il est possible de récupérer ces informations via la méthode getAttribut, pour ce faire il faut avoir la bonne clé mais cela permet de récupérer des informations du client connecté.

session.getAttribute("username")

Ici nous pouvons récupérer l’informations du client via le username stockés puis récupérer le nombre de visites effectuées.

Une image contenant texte, capture d’écran, logiciel, nombre

Le contenu généré par l’IA peut être incorrect.

Pour terminer une session il faut utiliser la méthode invalidate.

 session.invalidate();

Ici la session est détruite

Une image contenant texte, capture d’écran, logiciel, affichage

Le contenu généré par l’IA peut être incorrect.

## Documentation API avec Swagger

### Réalisation

Pour cet exercice il n’y a pas grand-chose à faire il suffit d’ajouter le bloc suivant dans le fichier pom.xml

   <dependency>

      <groupId>org.springdoc</groupId>

      <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

      <version>2.3.0</version>

   </dependency>

### Observation

Nous pouvons constater que la documentation s’est réalisée toute seul par rapport à notre API

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Le contenu généré par l’IA peut être incorrect.

## Hébergement

### Réalisation

Tout d’abord il faut créer un repository via le site suivant : <https://hub.docker.com>

Une image contenant texte, capture d’écran, Police, conception

Le contenu généré par l’IA peut être incorrect.

Ensuite il faut créer son token : <https://hub.docker.com/settings/security> puis dans son terminal il réaliser les actions données après la création

Une image contenant texte, capture d’écran, Police, logiciel

Le contenu généré par l’IA peut être incorrect.

Voici le message sensé s’affiché si l’opération s’est déroulée correctement

Une image contenant texte, capture d’écran, Police, ligne

Le contenu généré par l’IA peut être incorrect.

Pour finir il faut ajouter ses lignes dans le terminal

docker tag nomImageLocale:latest username/nomRepo:latest

docker push username/nomRepo:latest

Ce qui est sensé donné ceci :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

# Auto-évaluations et conclusions

Durant le module nous avons appris à utiliser plus en profondeur docker, j’ai eu un du mal a me mettre dans le module et bien comprendre la matière. Docker reste quelque chose de nouveau pour nous et je n’étais pas du tout habitué à son utilisation. De plus je trouve que les consignes d’exercices étaient assez vagues par moment ce qui cumulé à docker rend les choses complexes.

De manière générale je suis plus à l’aise avec l’utilisation de docker et la compréhension générale des taches demandés et les outils à disposition.