133 – Développer des applications WEB

Rapport personnel

Version 1 du 15.04.2025

Créé le 17.03.2025

Borgeat Nathan

|  |  |
| --- | --- |
| Une image contenant texte  Description générée automatiquement | Module du 13.03.2025 au 15.04.2025 |

[1 Introduction 4](#_Toc195608120)

[2 Tests technologiques selon les exercices 5](#_Toc195608121)

[2.1 Installation et Hello World 5](#_Toc195608122)

[2.2 Conteneurisation 5](#_Toc195608123)

[2.3 Création d'un projet Spring Boot 5](#_Toc195608124)

[2.4 Connexion à la DB JDBC 6](#_Toc195608125)

[2.5 Connexion à la DB JPA 6](#_Toc195608126)

[2.6 Connexion à la DB JPA avec DTO 6](#_Toc195608127)

[2.7 Gestion des sessions 7](#_Toc195608128)

[2.8 Documentation API avec Swagger 7](#_Toc195608129)

[2.9 Hébergement 7](#_Toc195608130)

[3 Auto-évaluations et conclusions 9](#_Toc195608131)

# Introduction

Voici les objectifs du module :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

# Tests technologiques selon les exercices

## Installation et Hello World

Voici un extrait de la console lors du démarrage du projet :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

La première ligne nous dit où nous nous trouvons actuellement.

Ensuite nous pouvons voir que Tomcat est initialisé et qu’il tourne sur le port 8080.

Avant d’arriver à ce point, il a fallu build Maven avec la commande « mvn clean package ». Le build permet de compiler le projet, ce qui le rend exécutable.

Le run permet d’exécuter le projet compilé.

La version de java se trouve aussi dans la capture d’écran (17.0.14).

Le serveur web Tomcat utilise quant à lui la version 10.1.24.

## Conteneurisation

Un conteneur contient tout ce dont a besoin un micro-service pour fonctionner.

Le serveur web se trouve par exemple dans ce conteneur.

Avec Java, il faut faire attention d’utiliser au minimum la version 11 car Spring Boot a été introduit dans cette version.

## Création d'un projet Spring Boot

Voici les annotations utilisées dans cet exercice :

@RestController : Indique que la classe est un contrôleur Spring et que ses méthodes renverront directement des réponses HTTP (JSON, texte, etc.)

@GetMapping : Associe une méthode à une requête HTTP GET sur une URL spécifique.

@PostMapping : Associe une méthode à une requête HTTP POST, utilisée pour envoyer des données au serveur.

@PutMapping : Associe une méthode à une requête HTTP PUT, souvent utilisée pour mettre à jour des ressources.

@RequestParam : Extrait un paramètre de la requête (GET) et le lie à un argument de méthode. Il permet aussi de définir une valeur par défaut.

@RequestBody : Indique que le corps de la requête HTTP (JSON, texte brut…) doit être converti en un objet Java et injecté dans la méthode.

## Connexion à la DB JDBC

Grâce à JDBC, nous pouvons désormais connecter une base de données à notre programme.

Nous avons besoin d’un conteneur créé au moyen de cette commande :

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

Cela va créer un conteneur qui va être disponible sur le port 3308 pour éviter les conflits avec une autre instance de mysql qui écoute déjà le port 3306.

Pour utiliser MySQL il ne faut pas oublier d’ajouter le driver dans le fichier pom.xml.

Dans le cas d’une configuration avec des conteneurs, il faut lier la base de données sous cette forme :

final String url = "jdbc:mysql://host.docker.internal:" + port + "/" + dbName

Voici une image résumant le fonctionnement de cette connexion :

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

Le contenu généré par l’IA peut être incorrect.

## Connexion à la DB JPA

L’annotation « @Autowired » permet de détecter automatiquement le Repository et l’initialiser automatiquement sans qu’on doive le gérer.

L’annotation « @ManyToOne » permet quant à elle de créer une relation n à 1 avec une table.

Dans notre cas, ce « @ManyToOne » est utilisé avec le FetchType « EAGER ». Cela permet de charger immédiatement l’entité liée, alors que le « LAZY » charge l’entité uniquement quand cela est explicitement demandé.

## Connexion à la DB JPA avec DTO

Dans cet exercice, il y a un SkieurDTO et pas de PaysDTO car on ne veut pas récupérer les pays, ce qui veut dire qu’il n’y a pas besoin de protéger les pays d’un potentiel vol d’informations importantes.

Voici maintenant une explication de chaque package :

* Model : Relie une table de la DB à une classe (par exemple Skieur ou Pays)
* Repository : Implémente les fonctionnalités CRUD pour chaque model.
* Dto : Classe réduite d’un model pour protéger du vol de données.
* Services : Classe qui contient les méthodes spécialement créées.
* Controllers : EndPoint du programme qui va appeler les méthodes services.

## Gestion des sessions

Les sessions permettent de séparer les différentes connexions et permettre aux différents utilisateurs d’avoir une session personnalisée. Dans notre exemple un utilisateur connecté pourra obtenir son nombre de connexions.

Voici les différents résultats possibles lors de l’url : localhost:8080/user/visites

Un utilisateur non connecté obtiendra ce résultat :



Tandis qu’un utilisateur connecté obtiendra ce genre de résultat :



## Documentation API avec Swagger

Swagger est une dépendance qui permet de générer automatiquement la documentation d’une API. Par défaut, l’url est la suivante (si fait en local) :

<http://localhost:8080/swagger-ui/index.html>

Voici un exemple de cette page :

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

Le contenu généré par l’IA peut être incorrect.

Il suffit d’ajouter la dépendance Swagger dans le fichier pom.xml et refaire une image pour que cette page apparaisse.

## Hébergement

Tout d’abord, il ne faut pas oublier de préparer les variables pour que les conteneurs accèdent aux bonnes URL (il ne faut pas mettre localhost tout le temps).

Ensuite, pour l’hébergement il y deux façons de faire, la première avec dockerhub et l’autre avec un serveur avec docker d’installé (module 239)

Pour dockerhub il faut créer un compte, puis un repository puis un token. Il faut bien enregistrer le mot de passe de ce token car il ne sera plus jamais disponible.

Une fois ces trois étapes faites, il faut aller dans un terminal sur VSCode et écrire la commande suivante :

docker login -u username

Avant d’entrer le token obtenu précédemment.

Comme dernière étape, il ne faut plus que faire les deux commandes suivantes pour tag et push l’image sur dockerhub :

docker tag nomImageLocale:latest username/nomRepo:latest

docker push username/nomRepo:latest

Pour l’hébergement web, il suffit d’avoir un serveur avoir docker d’installé, créer les conteneurs d’après les images pour finalement faire les liens pour rendre le tout accessible.

Cette partie concerne le module 239 c’est pourquoi je n’irai pas plus dans les détails.

# Auto-évaluations et conclusions

Personnellement, je pense avoir bien compris la matière dans sa globalité. D’après moi, je n’ai pas vraiment de points faibles pour ce module, j’ai eu 2-3 difficultés au début mais maintenant j’ai tout compris.

Ce module est une bonne comparaison avec le module 151 du bloc précédent. J’ai donc pu comparer PHP à Java et voir les avantages et inconvénients des deux langages.