Une image contenant texte

Description générée automatiquements

133 – Réaliser des applications Web en Sessions-Handling

Rapport personnel

Noam Bourqui

Version : 1 du 14.04.2025

Date de création : 17.03.25

Module du 17.03.2025 au 15.04.2025.

Tables des matières

[1 Introduction 6](#_Toc196227450)

[2 Tests technologiques selon les exercices 7](#_Toc196227451)

[2.1 Installation et Hello World 7](#_Toc196227452)

[2.1.1 Observez la console pour comprendre comment le projet est lancé et comment il tourne ? 7](#_Toc196227453)

[2.1.2 C'est quoi le build et le run de Java ? Quel outil a-t-on utiliser pour build le projet ? 7](#_Toc196227454)

[2.1.3 Y a-t-il un serveur web ? 7](#_Toc196227455)

[2.1.4 Quelle version de java est utilisée ? 7](#_Toc196227456)

[2.1.5 Si il y a un serveur web, quelle version utilise-t-il ? 7](#_Toc196227457)

[2.2 Conteneurisation 7](#_Toc196227458)

[2.2.1 Implémentation 7](#_Toc196227459)

[2.2.2 Mise en place 7](#_Toc196227460)

[2.2.3 Pourquoi faire un container pour une application Java ? 8](#_Toc196227461)

[2.2.4 Y a-t-il un serveur web ? Ou se trouve t'il ? 8](#_Toc196227462)

[2.2.5 A quoi faut-il faire attention (pensez au versions!) ? 8](#_Toc196227463)

[2.3 Création d'un projet Spring Boot 9](#_Toc196227464)

[2.3.1 Classe Controller 9](#_Toc196227465)

[2.3.2 @RestController 9](#_Toc196227466)

[2.3.3 @GetMapping 9](#_Toc196227467)

[2.3.4 @PostMapping 9](#_Toc196227468)

[2.3.5 @PutMapping 10](#_Toc196227469)

[2.3.6 @RequestParam 10](#_Toc196227470)

[2.3.7 @RequestBody 10](#_Toc196227471)

[2.3.8 Résumé des annotations utilisées 10](#_Toc196227472)

[2.4 Connexion à la DB JDBC 11](#_Toc196227473)

[2.4.1 Implémentation 11](#_Toc196227474)

[2.5 Connexion à la DB JPA 13](#_Toc196227475)

[2.5.1 Implémentation 13](#_Toc196227476)

[2.5.2 Questions 14](#_Toc196227477)

[2.6 Connexion à la DB JPA avec DTO 14](#_Toc196227478)

[2.6.1 Pourquoi un SkieurDTO et pas de PaysDTO ? 14](#_Toc196227479)

[2.6.2 Explication des différentes couches : 14](#_Toc196227480)

[2.7 Gestion des sessions 14](#_Toc196227481)

[2.8 1. Se connecter (POST /login) : 14](#_Toc196227482)

[2.8.1 Vérification des visites (GET /visites) : 15](#_Toc196227483)

[2.8.2 Se déconnecter (POST /logout) : 16](#_Toc196227484)

[2.9 Documentation API avec Swagger 16](#_Toc196227485)

[2.10 Hébergement 16](#_Toc196227486)

[2.10.1 Préparation des Variables dans le DockerFile 16](#_Toc196227487)

[2.10.2 Modification du fichier application.properties 17](#_Toc196227488)

[2.10.3 Modification du Dockerfile 17](#_Toc196227489)

[2.10.4 Envoyer l'image Docker sur DockerHub 17](#_Toc196227490)

[3 Auto-évaluation et conclusion 18](#_Toc196227491)

# Introduction

Ce projet consiste à développer une fonctionnalité spécifique pour une application web, en tenant compte des exigences fonctionnelles et de sécurité. L'objectif est de comprendre comment analyser les données, concevoir les fonctionnalités, puis les implémenter en utilisant un langage de programmation. Cela inclut des éléments comme la gestion des sessions, l'authentification des utilisateurs et la vérification des formulaires. Enfin, l'application sera testée pour vérifier son bon fonctionnement et sa sécurité. Ce processus permet de garantir que l'application soit à la fois efficace et protégée contre les vulnérabilités.

# Tests technologiques selon les exercices

## Installation et Hello World

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

Voici comment le projet est lancé et fonctionne :

1. **Lancement du projet** : Le projet est lancé en exécutant la classe principale RestServiceApplication avec la commande suivante :

/usr/bin/env /usr/lib/jvm/java-17-openjdk-amd64/bin/java @/tmp/cp\_32cvcyr1yeuehiodxfebd8v54.argfile com.example.restservice.RestServiceApplication

Cela démarre l'application Spring Boot.

1. **Exécution** : L'application utilise **Java 17** pour démarrer, comme l'indique la ligne :

Starting RestServiceApplication using Java 17.0.14

1. **Serveur web embarqué** : Le serveur web **Tomcat** est initialisé à port 8080, ce qui est confirmé par la ligne :

Tomcat started on port 8080 (http) with context path '/'

1. **Processus** : L'application démarre et le processus de Spring Boot est exécuté, prêt à recevoir des requêtes via le serveur web embarqué.

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

Le projet est construit via Maven, car je l’ai choisi au départ de l’application.

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

Oui, un Tomcat

### Quelle version de java est utilisée ?

La version de Java utilisé est la 17.0.14.

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

La version utilisée est la 10.1.24.

## Conteneurisation

### Implémentation

### Mise en place

Pour commencer, on crée un fichier DockerFile : CTRL+SHIFT+P -> Docker: Add Docker Files To Workspace…

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

Le contenu généré par l’IA peut être incorrect.

Ensuite on choisit : Java, complete/pom.xml, port 8080

Puis on doit avoir un DockerFiles ressemblant à :

# Utilisation de l'image openjdk

FROM openjdk:17-jdk-alpine

VOLUME /tmp

ARG JAVA\_OPTS

ENV JAVA\_OPTS=$JAVA\_OPTS

COPY complete/target/rest-service-complete-0.0.1-SNAPSHOT.jar complete.jar

EXPOSE 8080

ENTRYPOINT exec java $JAVA\_OPTS -jar complete.jar

Attention : vérifiez bien le chemin d’accès, dans la consigne c’est target/… mais le dossier target ce trouve dans complete/.

Après, Il faut installer maven sur notre WSL :

Sudo apt install maven

Et pour finir Build le projet avec maven pour avoir le .jar du Dockerfile :

cd /complete #projet à build

mvn clean package

On fait un clic droit sur DockerFile -> Build Images

Dans l'onglet Docker de VSCode : faite un clic droit sur l'image -> "Run"

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

Le contenu généré par l’IA peut être incorrect.

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

* Portabilité : Permet de déployer l'application dans différents environnements sans souci de configuration locale.
* Isolation : L'application et ses dépendances sont isolées, évitant les conflits de versions.
* Facilité de déploiement : Docker simplifie l'automatisation du déploiement et l'intégration continue.
* Optimisation des ressources : Moins de consommation comparée aux machines virtuelles.

### Y a-t-il un serveur web ? Ou se trouve t'il ?

* Oui : L'application Spring Boot inclut un serveur web embarqué (comme Tomcat).
* Emplacement : Le serveur est intégré directement dans l'application via le JAR, donc il n'y a pas de serveur web séparé.

### A quoi faut-il faire attention (pensez au versions!) ?

* Versions de Java : Assurez-vous que la version de Java dans le Dockerfile correspond à celle de l'application.
* Dockerfile : Le fichier doit être correctement structuré (copier le JAR, définir les variables d'environnement, exposer le port).
* Compatibilité des dépendances : Assurez-vous que toutes les dépendances nécessaires sont incluses.
* Gestion des ports : Exposez correctement le port pour accéder à l'application (ici le port 8080).
* Mise à jour des images Docker : Utilisez des images sécurisées et régulièrement mises à jour.

## Création d'un projet Spring Boot

### Classe Controller

@RestController

public class Controller {

   // Handler pour GET

   @GetMapping("/getExample")

   public String getExample(@RequestParam(value = "name", defaultValue = "World") String name) {

       return String.format("Hello, %s!", name);

   }

   // Handler pour POST

   @PostMapping("/postExample")

   public String postExample(@RequestBody String body) {

       return "Received POST request with body: " + body;

   }

   // Handler pour PUT

   @PutMapping("/putExample")

   public String putExample(@RequestBody String body) {

       return "Received PUT request with body: " + body;

   }

}

### @RestController

Cette annotation indique que la classe est un contrôleur REST. Elle est une combinaison de @Controller et @ResponseBody, ce qui signifie que les méthodes de cette classe retournent directement les réponses HTTP (souvent au format JSON ou texte), sans avoir besoin de passer par des vues (comme avec un contrôleur MVC traditionnel).

@RestController

public class Controller { ... }

### @GetMapping

Cette annotation est utilisée pour marquer une méthode comme un handler pour les requêtes HTTP GET. Elle est une version spécialisée de l'annotation @RequestMapping qui est utilisée pour gérer les requêtes GET sur un chemin spécifique.

@GetMapping("/getExample")

public String getExample(@RequestParam(value = "name", defaultValue = "World") String name) { ... }

### @PostMapping

Cette annotation marque la méthode comme étant un handler pour les requêtes HTTP POST. Elle permet de gérer les requêtes POST envoyées vers un chemin spécifique.

@PostMapping("/postExample")

public String postExample(@RequestBody String body) { ... }

### @PutMapping

Cette annotation marque la méthode comme étant un handler pour les requêtes HTTP PUT. Elle permet de gérer les requêtes PUT envoyées vers un chemin spécifique.

@PutMapping("/putExample")

public String putExample(@RequestBody String body) { ... }

### @RequestParam

Cette annotation permet d'extraire les paramètres de la requête GET. Elle est utilisée pour accéder aux paramètres de la requête HTTP envoyée. Ici, elle permet de récupérer le paramètre name de l'URL. Si ce paramètre est absent, la valeur par défaut "World" est utilisée.

@RequestParam(value = "name", defaultValue = "World") String name

### @RequestBody

Cette annotation est utilisée pour extraire le corps de la requête HTTP dans les méthodes POST et PUT. Elle permet de lier directement le corps de la requête au paramètre de la méthode. Dans ton exemple, elle est utilisée pour récupérer le corps du POST ou PUT envoyer dans la requête HTTP.

@RequestBody String body

### Résumé des annotations utilisées

* @RestController : Indique que la classe est un contrôleur REST.
* @GetMapping : Gère les requêtes HTTP GET.
* @PostMapping : Gère les requêtes HTTP POST.
* @PutMapping : Gère les requêtes HTTP PUT.
* @RequestParam : Extrait un paramètre de la requête GET.
* @RequestBody : Extrait le corps de la requête HTTP (POST ou PUT).

Ces annotations permettent de facilement gérer les différentes méthodes HTTP et les paramètres associés dans un contrôleur Spring.

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

Le contenu généré par l’IA peut être incorrect.

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

Le contenu généré par l’IA peut être incorrect.

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

Le contenu généré par l’IA peut être incorrect.

## Connexion à la DB JDBC

### Implémentation

Pour commencer on créer un contenaire MySql pour notre base de données :

#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 on récupère la DB et on la met dans WorkBench et on ajoute dans notre contrôleur une fonctionnalité getPays pour récupérer les pays (grâce à la logique dans WrkDB :

Controller :

@GetMapping("/getPays")

   public List<String> getPays() {

       return wrkDB.getPays();

   }

WrkDB ; on ajoute les identifications de connexion :

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

final String user = "root";

final String pw =  "emf123";//

boolean result = false;

Requête exécuter pour récupérer les pays (getPays) :

ps = dbConnexion.prepareStatement("SELECT Nom FROM t\_pays");

Ensuite on ajout des drivers JDBC dans le pom.xml avec un clic droit sur le pom.xml et « Add starter », puis MySQL :

Une image contenant texte, capture d’écran, conduire, mémoire flash

Le contenu généré par l’IA peut être incorrect.

Ceci a dû ajouter au fichier pom.xml le dependency suivante :

<dependency>

<groupId>com.mysql</groupId>

<artifactId>mysql-connector-j</artifactId>

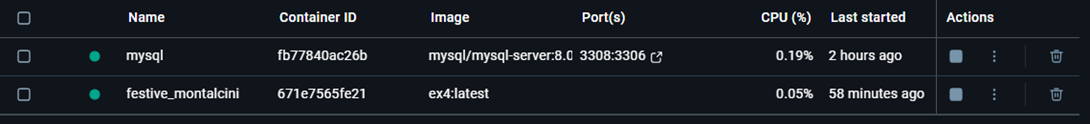
<scope>runtime</scope>

</dependency>

Dans le backend, vous pouvez maintenant vous connecter à la base de données en utilisant les credentials :

'mysql:host=localhost;port=3308;dbname=nomDB;charset=utf8', 'root', 'emf123'

Pour finir il faut créer une connexion docker en y ajoutant notre container MySQL et celui de notre exercice (création du dockerFile -> voir étape de l’exercice 4). Pour savoir le nom il faut aller dans docker :



Et exécuter les commandes suivantes dans un terminal :

docker network create asgard

docker network connect asgard mysql

docker network connect asgard nomContainerBackend

Pour finir on test avec Postman notre getPays :

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

Le contenu généré par l’IA peut être incorrect.

## Connexion à la DB JPA

### Implémentation

Pour la réalisation de cet exercice, j’ai dû modifier le fichier .properties et le pom.xml

.properties

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

### Questions

* **@Autowired** : Cette annotation permet à Spring d’injecter automatiquement le bean SkieurRepository dans le contrôleur, ce qui simplifie l'accès aux méthodes de gestion des entités Skieur.
* **@ManyToOne** : Cette annotation définit une relation entre l'entité Skieur et l'entité Pays. Un skieur appartient à un pays.
  + **FetchType.EAGER** : Cela signifie que les pays associés aux skieurs seront toujours récupérés en même temps que les skieurs. Si vous testez avec **FetchType.LAZY**, le pays ne sera récupéré que lorsqu'il est explicitement demandé.

## Connexion à la DB JPA avec DTO

### Pourquoi un SkieurDTO et pas de PaysDTO ?

* **SkieurDTO** est nécessaire car nous exposons directement les informations des skieurs dans l'API, et ces informations peuvent être manipulées ou filtrées avant d'être envoyées au client.
* **PaysDTO** n'est pas nécessaire, car le pays d'un skieur est uniquement une référence dans l'entité Skieur et nous n'avons pas besoin d'exposer l'entité Pays entière dans l'API. Nous envoyons simplement le nom du pays dans le SkieurDTO.

### Explication des différentes couches :

* **Modèles** : Ce sont des classes qui représentent les entités de la base de données, comme Skieur et Pays.
* **Repository** : Ce sont des interfaces qui permettent de gérer les opérations CRUD (Create, Read, Update, Delete) sur les entités en utilisant Spring Data JPA.
* **DTO** : Ce sont des objets qui sont utilisés pour transférer des données entre différentes couches de l'application. Les DTOs sont souvent utilisés pour ne transmettre que les données nécessaires et éviter l'exposition directe des entités.
* **Services** : Les services contiennent la logique métier. Ils gèrent les interactions entre le contrôleur et les repositories, et la transformation des données entre entités et DTOs.
* **Contrôleurs** : Les contrôleurs sont responsables de la gestion des requêtes HTTP. Ils exposent les API REST et appellent les services pour traiter les données.

## Gestion des sessions

## 1. Se connecter (POST /login) :

Dans la méthode login, j'ai mis en place la logique pour permettre à un utilisateur de se connecter. Lorsque l'utilisateur fait une requête POST à l'endpoint /login avec son nom d'utilisateur et son mot de passe, je vérifie si ces informations correspondent aux valeurs par défaut ("user" et "pass").

Si la connexion est réussie, je crée une session et y stocke deux informations essentielles :

Le nom d'utilisateur (username), pour pouvoir savoir qui est connecté.

Un compteur de visites (visites), initialisé à 0, pour pouvoir suivre combien de fois l'utilisateur a accédé à l'application.

Voici le code :

@PostMapping("/login")

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

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

session.setAttribute("username", username); // Stocke le nom d'utilisateur dans la session

session.setAttribute("visites", 0); // Initialise le compteur de visites à 0

return ResponseEntity.ok("Connecté");

} else {

return ResponseEntity.badRequest().body("Échec de la connexion"); // Renvoie une erreur si les identifiants sont incorrects

}

}

Ce que cela fait :

Si les identifiants sont corrects, je crée une session et j'y associe l'utilisateur et un compteur de visites.

Une fois connecté, la session est active et pourra être utilisée lors des prochaines requêtes.

### Vérification des visites (GET /visites) :

Pour cette fonctionnalité, je veux suivre le nombre de visites de l'utilisateur connecté. Chaque fois que l'utilisateur fait une requête GET à l'endpoint /visites, je vérifie d'abord si le nom d'utilisateur (username) est bien présent dans la session. Si c'est le cas, cela signifie que l'utilisateur est connecté, et je récupère son compteur de visites. Je l'incrémente, puis je mets à jour la session avec la nouvelle valeur.

Voici le code pour cette méthode :

@GetMapping("/visites")

public ResponseEntity<String> visites(HttpSession session) {

if (session.getAttribute("username") != null) { // Vérifie si l'utilisateur est connecté

Integer visites = (Integer) session.getAttribute("visites"); // Récupère le compteur de visites

visites++; // Incrémente le compteur de visites

session.setAttribute("visites", visites); // Mette à jour le compteur dans la session

return ResponseEntity.ok("Nombre de visites: " + visites); // Retourne la nouvelle valeur du compteur

} else {

return ResponseEntity.badRequest().body("Non connecté"); // Si l'utilisateur n'est pas connecté, renvoie une erreur

}

}

Ce que cela fait :

Si l'utilisateur est connecté, je récupère son nombre de visites et l'incrémente à chaque appel à /visites.

Si l'utilisateur n'est pas connecté, je renvoie une erreur pour lui signaler qu'il doit être connecté pour accéder à cette information.

### Se déconnecter (POST /logout) :

Dans cette méthode, j'ai mis en place la déconnexion de l'utilisateur. Quand l'utilisateur fait une requête POST à l'endpoint /logout, je valide la session en utilisant session.invalidate(). Cela supprime tous les attributs stockés dans la session, ce qui signifie que l'utilisateur est complètement déconnecté.

Voici le code pour cette méthode :

@PostMapping("/logout")

public ResponseEntity<String> logout(HttpSession session) {

session.invalidate(); // Invalide la session, supprimant ainsi toutes les informations stockées

return ResponseEntity.ok("Déconnecté"); // Renvoie un message de confirmation

}

Ce que cela fait :

En invalidant la session, je m'assure que toutes les informations (comme le nom d'utilisateur et le compteur de visites) sont effacées, et l'utilisateur est déconnecté.

La réponse Déconnecté est renvoyée pour informer l'utilisateur qu'il a bien été déconnecté.

## Documentation API avec Swagger

Dans cet exercice, nous avons dû ajouter une dépendance Swagger qui vient de Spring-boot-

Pour cela, j’ai simplement ajouté la dépendance nécessaire :

<dependency>

<groupId>org.springdoc</groupId>

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

<version>2.3.0</version>

</dependency>

Une image contenant texte, capture d’écran, logiciel, Page web

Le contenu généré par l’IA peut être incorrect.

## Hébergement

### Préparation des Variables dans le DockerFile

Pour rendre mon application plus flexible et adaptable aux environnements différents (développement, production), j'ai paramétré certaines valeurs de configuration à l'aide de variables d'environnement.

### Modification du fichier application.properties

J'ai modifié le fichier application.properties de l'application Spring afin de ne pas hardcoder l'URL de la base de données. J'ai utilisé la syntaxe suivante :

spring.datasource.url=${DATABASE\_URL:jdbc:mysql://localhost:3306/dbprojet}

Cela permet à l'application de chercher d'abord une variable d'environnement nommée DATABASE\_URL. Si elle n'est pas trouvée, la valeur par défaut jdbc:mysql://localhost:3306/dbprojet est utilisée.

### Modification du Dockerfile

Dans mon Dockerfile, j'ai ajouté une ligne pour définir la variable d'environnement DATABASE\_URL. Voici ce que j'ai ajouté :

ENV DATABASE\_URL=jdbc:mysql://host.docker.internal:3306/dbprojet

Cela permet à mon application Dockerisée de se connecter à la base de données en utilisant l'URL correcte, même si l'application est déployée sur un serveur de production.

### Envoyer l'image Docker sur DockerHub

Une fois que mon application a été conteneurisée avec Docker, j'ai voulu la mettre en ligne pour pouvoir la déployer facilement sur un serveur de production. Voici comment je l'ai fait.

#### Créer un compte sur DockerHub

1. Je me suis rendu sur [DockerHub](https://hub.docker.com) et j'ai créé un compte.

#### Créer un repository sur DockerHub

1. Une fois connecté à DockerHub, j'ai créé un repository public pour héberger l'image Docker de mon application.

#### Créer un token DockerHub

1. Dans les paramètres de sécurité de DockerHub, j'ai généré un **Access Token** pour pouvoir me connecter en ligne de commande sans utiliser mon mot de passe.
2. J'ai copié ce token, car je devais l'utiliser pour m'authentifier dans le terminal.

#### Se connecter à DockerHub

Dans le terminal, je me suis connecté à DockerHub en exécutant la commande suivante :

docker login -u username

On m'a demandé d'entrer mon **token** en tant que mot de passe, et une fois connecté, j'étais prêt à envoyer mon image Docker.

**Taguer et pousser l'image vers DockerHub**

1. J'ai ensuite tagué mon image locale pour qu'elle soit liée à mon repository DockerHub :

docker tag nomImageLocale:latest username/nomRepo:latest

1. Ensuite, j'ai poussé l'image vers DockerHub avec la commande suivante :

docker push username/nomRepo:latest

Cela a mis mon image Docker en ligne, disponible pour le déploiement sur d'autres serveurs.

# Auto-évaluation et conclusion

Ce module m'a permis d'apprendre énormément sur le développement et le déploiement d'applications web en Java. J'ai commencé par configurer mon environnement de travail avec WSL et Docker, ce qui m'a aidé à mieux comprendre comment créer et gérer des containers. Ensuite, j'ai exploré la gestion des sessions avec Spring Boot, ce qui m'a montré comment stocker des données utilisateur, comme un compteur de visites, dans une session HTTP. J'ai aussi travaillé avec JDBC et JPA pour connecter mon application à une base de données, ce qui m'a permis de comprendre l'importance de l'interaction entre l'application Java et la base de données. Enfin, j'ai appris à utiliser Swagger pour documenter mes API, et à déployer mes applications sur un serveur Docker. Ce module m'a donné des bases solides pour développer des applications web robustes et bien structurées, tout en me préparant à les déployer sur des serveurs de production.