

BDD et Web

JAVA & SPRING BOOT

LES APIS REST

Partie 1 - Introduction à Java

- ▶ 1. Présentation de Java et des bases de la POO (Programmation Orientée Objet)
- ▶ 2. Gestion des types de données et des collections
- ▶ 3. Introduction aux interfaces et classes abstraites
- ▶ 4. Gestion des exceptions
- ▶ 5. Rappels sur les packages et organisation du code
- ▶ 6. Introduction à Maven et aux dépendances


1. Présentation de Java et des bases de la POO (Programmation Orientée Objet)

Concepts fondamentaux

- ▶ **Classes** : Une classe est un modèle, ou "template", qui définit les caractéristiques (attributs) et les comportements (méthodes) des objets qui en seront issus.
- ▶ **Objets** : Instances concrètes d'une classe, chaque objet a son propre état et peut utiliser les méthodes définies dans sa classe.
- ▶ **Encapsulation** : Principe de cacher les détails d'implémentation internes d'une classe et d'exposer uniquement ce qui est nécessaire via des méthodes publiques.
- ▶ **Héritage** : Permet de créer de nouvelles classes à partir de classes existantes (ex : une classe *Pomme* peut hériter de la classe *Fruit*)
- ▶ **Polymorphisme** : Capacité d'utiliser des objets de différentes classes à travers une interface commune.


Exemple : Classe Person avec des attributs et méthodes.

java

 Copier le code

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

java

 Copier le code

```
public class Main {  
    public static void main(String[] args) {  
        Person person = new Person("Alice", 30);  
        System.out.println("Name: " + person.getName());  
        person.setAge(31);  
        System.out.println("New Age: " + person.getAge());  
    }  
}
```

2. Gestion des types de données et des collections


► Types de bases :

- int
- double / float
- String
- Boolean

► Tableaux et collections

- List
- Set (List avec unicité)
- Map

java

 Copier le code

```
List<String> names = new ArrayList<>();  
names.add("Alice");  
names.add("Bob");  
  
Map<Integer, String> students = new HashMap<>();  
students.put(1, "Alice");  
students.put(2, "Bob");
```

```
List<Person> people = new ArrayList<>();  
people.add(new Person("Alice", 30));  
people.add(new Person("Bob", 25));  
  
for (Person p : people) {  
    System.out.println(p.getName());  
}
```

3. Introduction aux interfaces et classes abstraites

Une **interface** déclare des méthodes sans les implémenter.

Une interface est destinée à être implémentée par une classe, on peut la définir comme un contrat.

Pourquoi utiliser des interface ?

Fournir une structure commune pour des objets différents sans relation d'héritage (ex : tous les objets "vivants" respirent, mais ils ne descendent pas d'une même classe).


Différence avec une classe abstraite :

Une **classe abstraite** peut avoir des méthodes avec et sans implémentation, et est destinée à être étendue. On peut la définir comme un squelette.

3. Introduction aux interfaces et classes abstraites

Exemple d'interface :

java

 Copier le code

```
interface LivingBeing {  
    void breathe();  
}  
  
class Person implements LivingBeing {  
    public void breathe() {  
        System.out.println("Person is breathing.");  
    }  
}
```

3. Introduction aux interfaces et classes abstraites

Exemple de classe abstraite :


java

```
// Classe abstraite Animal
public abstract class Animal {

    // Méthode abstraite (sans implémentation)
    public abstract void makeSound();

    // Méthode concrète (avec implémentation)
    public void eat() {
        System.out.println("L'animal mange.");
    }
}
```

java

 Copier le code

```
// Sous-classe Dog
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Le chien aboie : Woof Woof !");
    }
}

// Sous-classe Cat
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Le chat miaule : Meow !");
    }
}
```



4. Gestion des exceptions

Une **exception** sert à gérer les erreurs au sein du programme pour **éviter les crash** et **faciliter le débogage**.

Types d'exceptions courantes :

- ▶ **NullPointerException** : lorsqu'on tente d'utiliser un objet *null*
- ▶ **ArrayIndexOutOfBoundsException** : provoquée par un index non valide pour un tableau

java

 Copier le code

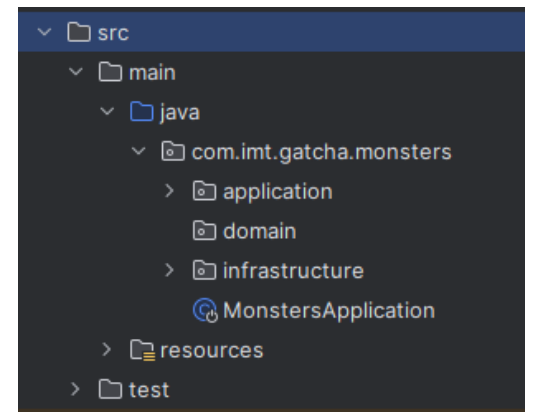
```
public class Main {  
    public static void main(String[] args) {  
        try {  
            int[] numbers = {1, 2, 3};  
            System.out.println(numbers[5]); // Génère une exception  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Index hors limites : " + e.getMessage());  
        }  
    }  
}
```

5. Rappels sur les packages et organisation du code

On organise les classes par package pour structurer le code et éviter les conflits de noms.

Par exemple :

- ▶ `com.example.model` pour les classes comme *Person*
- ▶ `com.example.controller` pour les contrôleurs



Le bon nommage des packages est important, il existe plein de méthodes d'architecture du code différentes, dont l'objectif est généralement d'en optimiser la maintenabilité.

Pour utiliser du code d'autres packages, on va devoir importer les classes via la commande *import*.

```
import java.io.FileWriter;
```

6. Introduction à Maven et aux dépendances

Maven est un outil de gestion de projet qui facilite la gestion des dépendances et la configuration de projets Java.

Il utilise pour cela un fichier connu : *pom.xml*

xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.6.3</version>
  </dependency>
</dependencies>
```

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.2.2</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.imt.gatcha</groupId>
<artifactId>monsters</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>monsters</name>
<description>Monsters API</description>
<properties>
  <java.version>21</java.version>
</properties>
```

Partie 2 - Introduction à Spring Boot

- ▶ 1. Introduction détaillée à Spring Boot
- ▶ 2. Structure et configuration d'un projet Spring Boot
- ▶ 3. Annotations Spring Boot et leurs rôles
- ▶ 4. Gestion des exceptions et validation des données
- ▶ 5. Documentation et tests de l'API avec Spring Boot

1. Introduction à Spring Boot

Spring Framework : À l'origine, Spring est un framework pour simplifier le développement Java, mais il nécessitait beaucoup de configuration (XML) et une gestion complexe des dépendances.

Spring Boot est né pour éliminer la complexité en introduisant :

- ▶ Réduction des configurations
- ▶ Démarrage rapide
- ▶ Serveur embarqué
- ▶ Pré-configuration des dépendances

1. Introduction à Spring Boot

Focus sur la pré-configuration

- Au lieu de spécifier manuellement les librairies on ajoute un starter approprié.

Par exemple, pour une application Web, au lieu de chercher et d'ajouter plusieurs dépendances pour configurer un serveur Tomcat, gérer les requêtes HTTP, ou encore transformer les données en JSON, on peut simplement ajouter le starter *spring-boot-starter-web*.

On récupère ainsi les bibliothèques comme Tomcat, Spring MVC et Jackson.

1. Introduction à Spring Boot

Focus sur la pré-configuration | Exemple

Disons que l'on souhaite créer une API REST avec une base de données. Avec Spring Boot :

- ▶ *spring-boot-starter-web* pour avoir une API REST prête avec Tomcat embarqué et Jackson
- ▶ *spring-boot-starter-data-jpa* pour configurer l'accès à une base de données
- ▶ *spring-boot-starter-h2* pour que Spring Boot configure automatiquement la BDD en mémoire pour les tests, et y connecte JPA

1. Introduction à Spring Boot

Focus sur la pré-configuration

Spring Boot gère aussi les versions compatibles entre les bibliothèques.

Les starters utilisent des versions qui sont connues pour bien fonctionner ensemble, ce qui réduit le risque de conflits entre bibliothèques et rend la configuration stable.

Spring Boot est régulièrement mis à jour pour suivre les nouvelles versions de bibliothèques populaires tout en assurant leur compatibilité dans un même projet.

1. Introduction à Spring Boot

Focus sur la pré-configuration | Avantages

- ▶ **Gain de temps** : Les développeurs n'ont pas à rechercher et configurer manuellement chaque bibliothèque nécessaire.
- ▶ **Moins de risques d'erreurs** : Évite les erreurs liées aux versions incompatibles.
- ▶ **Simplification** : Permet de se concentrer sur le code métier au lieu des configurations techniques.

Ce système de dépendances de démarrage et d'auto-configuration est une des raisons principales pour lesquelles Spring Boot est si populaire pour le développement d'applications Java modernes.

2. Structure et configuration d'un projet Spring Boot

- Tout d'abord pour générer le projet : Spring Initializr

Outil en ligne qui permet d'initialiser un projet Spring Boot avec les dépendances demandées.

On pourra y choisir également le langage (java, kotlin..), la version, et le gestionnaire de dépendances (maven ou gradle).

2. Structure et configuration d'un projet Spring Boot

On retrouvera à l'intérieur du projet généré :

- ▶ *src/main/java* : contient le code source Java
- ▶ *src/main/ressources* : contient les ressources (template, fichiers de configuration, fichiers statiques)
- ▶ *application.properties* : fichier de configuration principal pour l'application

On pourra y changer le port de l'application et ajouter les informations de connexion à la BDD par exemple.

3. Annotations Spring Boot et leurs rôles

- ▶ 1. `@SpringBootApplication`
- ▶ 2. `@RestController`
- ▶ 3. `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`
- ▶ 4. `@PathVariable` et `@RequestParam`
- ▶ 5. `@Service`
- ▶ 6. `@Repository`
- ▶ 7. `@Autowired`
- ▶ 8. `@Configuration` et `@Bean`
- ▶ 9. `@ControllerAdvice` et `@ExceptionHandler`
- ▶ 10. `@Profile`

@SpringBootApplication

Cette annotation se place sur la classe principale de l'application, celle qui contient la méthode *main*. Elle combine trois autres annotations :

- ▶ **@Configuration** : indique que la classe contient des *beans* à configurer.
- ▶ **@EnableAutoConfiguration** : active l'auto-configuration Spring Boot en fonction des dépendances présentes.
- ▶ **@ComponentScan** : scanne automatiquement le package actuel et ses sous-packages pour trouver les composants Spring (@Component, @Service, @Repository, etc...)


@SpringBootApplication

Exemple :

Ici @SpringBootApplication rend la classe prête à lancer l'application Spring Boot.

Inutile de déclarer manuellement des configurations supplémentaires pour activer l'auto-configuration.

java

 Copier le code

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

@RestController

Utilisée pour les classes qui définissent des **contrôleurs REST**. Elle combine `@Controller` et `@ResponseBody`.

Elle indique que chaque méthode de la classe renverra directement les données (généralement au format JSON ou XML) en réponse HTTP.

```
java Copier le code

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello, World!";
    }
}
```

Avec `@RestController`, la méthode `sayHello` renvoie directement la chaîne « Hello World! » en tant que réponse HTTP sans passer par quelque vue.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping

Ces annotations facilitent la gestion des requêtes HTTP spécifiques.

Elles sont des raccourcis pour `@RequestMapping` avec des méthodes HTTP prédéfinies.

Dans l'exemple à droite, ces annotations rendent le code plus lisible en explicitant le type de requête associé à chaque méthode.

```
java Copier le code

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

@RestController
@RequestMapping("/api/users")
public class UserController {

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        // Logique pour récupérer un utilisateur par son ID
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        // Logique pour créer un nouvel utilisateur
    }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User user) {
        // Logique pour mettre à jour un utilisateur existant
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        // Logique pour supprimer un utilisateur par ID
    }
}
```



@PathVariable et @RequestParam

@PathVariable : extrait une variable de l'URL.

@RequestParam : extrait un paramètre de requête dans l'URL.

@PathVariable est utilisé pour obtenir une valeur dans l'URL dynamique, alors que @RequestParam est utilisé pour les paramètres de requête qui suivent le point d'interrogation dans l'URL.

java

 Copier le code

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

@RestController
public class ProductController {


    @GetMapping("/products/{id}")
    public Product getProductById(@PathVariable Long id) {
        // Logique pour récupérer un produit par ID
    }

    @GetMapping("/products")
    public List<Product> getProductsByCategory(@RequestParam String category) {
        // Logique pour récupérer des produits par catégorie
    }
}
```

@Service

@Service est utilisée pour annoter les classes qui contiennent la logique métier. Elle indique à Spring que cette classe doit être gérée en tant que service, et elle sera injectée dans d'autres composants avec **@Autowired**.

java

 Copier le code

```
import org.springframework.stereotype.Service;

@Service
public class UserService {


    public User findUserById(Long id) {
        // Logique pour trouver un utilisateur
    }
}
```

@Service clarifie l'intention de la classe et permet à Spring de la gérer comme un composant logique injectable.

@Repository

@Repository est utilisée pour les classes qui gèrent la persistance des données (généralement en accédant à une base de données). Elle permet également une gestion simplifiée des exceptions liées à la base de données.

java

 Copier le code

```
import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // Méthodes spécifiques de requêtes peuvent être ajoutées ici
}
```

@Repository permet à Spring de reconnaître cette interface comme un composant de gestion de données.

@Autowired

@Autowired est utilisée pour injecter automatiquement des dépendances gérées par Spring, comme des services ou des repositories.

```
java Copier le code

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class OrderService {

    private final UserRepository userRepository;

    @Autowired
    public OrderService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```


Ici, **@Autowired** injecte une instance de *UserRepository* dans *OrderService*, ce qui permet de l'utiliser sans avoir à l'instancier manuellement.

@Configuration et @Bean

@Configuration indique que la classe contient des définitions de beans.

@Bean indique qu'une méthode retourne un bean géré par Spring.

java

 Copier le code

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```


Dans cet exemple, *PasswordEncoder* sera disponible pour être injecté dans d'autres composants.

@ControllerAdvice et @ExceptionHandler

@ControllerAdvice permet de définir une gestion globale des exceptions pour tous les contrôleurs.

@ExceptionHandler : Gère une exception spécifique et permet de renvoyer une réponse adaptée.

java

 Copier le code

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public String handleResourceNotFound(ResourceNotFoundException ex) {
        return ex.getMessage();
    }
}
```

@ControllerAdvice et @ExceptionHandler permettent ici de catcher les erreurs *ResourceNotFoundException* et de renvoyer une réponse **404 NOT_FOUND** à l'utilisateur.


On pourra centraliser toutes les erreurs qu'on veut gérer dans cette classe.

@Profile

@Profile active des configurations spécifiques selon l'environnement (dev, qual, prod, test...).

Le profile est paramétrable dans le fichier de configuration de l'application Spring Boot (application.yml) ou dans les arguments de JVM pour lancer l'application.

java

 Copier le code

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
public class DataSourceConfig {

    @Bean
    @Profile("dev")
    public DataSource devDataSource() {
        // Retourne une source de données pour l'environnement de développement
    }

    @Bean
    @Profile("prod")
    public DataSource prodDataSource() {
        // Retourne une source de données pour l'environnement de production
    }
}
```

3. Annotations Spring Boot et leurs rôles

Conclusion :

Ces annotations permettent une **grande flexibilité** et une **configuration simplifiée** dans Spring Boot, tout en respectant des principes comme **l'injection de dépendances** et la **séparation des responsabilités**. En utilisant les annotations Spring Boot, les développeurs peuvent facilement **structurer leur code**, éviter des configurations complexes, et garder une application bien organisée.

4. Gestion des exceptions et validation des données


Gestion des exceptions dans Spring Boot :

La gestion des exceptions dans Spring Boot repose principalement sur les deux annotations vu précédemment :

- ▶ @ControllerAdvice
- ▶ @ExceptionHandler

On peut créer des exception personnalisées pour représenter des erreurs spécifiques.

java

 Copier le code

```
public class ResourceNotFoundException extends RuntimeException {  
    public ResourceNotFoundException(String message) {  
        super(message);  
    }  
}
```

4. Gestion des exceptions et validation des données

handleResourceNotFound() gère spécifiquement les **ResourceNotFoundException**.

On renvoie alors une **404 NOT_FOUND**.

handleIllegalArgument() gère lui les erreurs de validation en renvoyant une réponse **400 BAD_REQUEST**.

Ainsi, toutes les exceptions de ces types seront capturées par ce contrôleur d'exception global, rendant le code plus propre et facilitant la gestion des erreurs.

```
@RestControllerAdvice // Utilise RestControllerAdvice pour les APIs REST
public class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND) // Définit le statut HTTP 404
    public ErrorResponse handleResourceNotFound(ResourceNotFoundException ex) {
        return new ErrorResponse("Not Found", ex.getMessage());
    }

    @ExceptionHandler(IllegalArgumentException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ErrorResponse handleIllegalArgument(IllegalArgumentException ex) {
        return new ErrorResponse("Bad Request", ex.getMessage());
    }
}

class ErrorResponse {
    private String error;
    private String message;

    // Constructeurs, getters et setters

    public ErrorResponse(String error, String message) {
        this.error = error;
        this.message = message;
    }

    // Getters et setters
}
```

4. Gestion des exceptions et validation des données

Validation des données avec Spring Boot

Spring Boot permet de valider les données des requêtes HTTP via des annotations de validation.

Les annotations de validation peuvent être appliquées sur les champs des objets transmis, souvent utilisés dans les contrôleurs REST avec @RequestBody.

4. Gestion des exceptions et validation des données

Les annotations les plus courantes :

- ▶ **@NotNull** : Le champ ne doit pas être nul.
- ▶ **@NotBlank** : Le champ ne doit pas être vide (spécifique pour les chaînes de caractères).
- ▶ **@Size(min=, max=)** : Définit une taille minimale et/ou maximale.
- ▶ **@Email** : Vérifie si le champ contient une adresse email valide.
- ▶ **@Min et @Max** : Spécifie des valeurs minimales et maximales pour les nombres.

4. Gestion des exceptions et validation des données

Les annotations les plus courantes | Exemple :

```
java Copier le code

import javax.validation.constraints.*;

public class UserDTO {
    @NotNull(message = "Le nom ne doit pas être nul")
    @Size(min = 2, max = 30, message = "Le nom doit avoir entre 2 et 30 caractères")
    private String name;

    @NotBlank(message = "L'email est obligatoire")
    @Email(message = "L'email doit être valide")
    private String email;

    @Min(value = 18, message = "L'âge doit être au moins de 18 ans")
    private int age;

    // Getters et setters
}
```

4. Gestion des exceptions et validation des données


Validation dans les contrôleurs :

Pour appliquer la validation, il suffit d'annoter le paramètre d'entrée avec **@Valid** dans le contrôleur.

En cas d'échec, Spring Boot déclenche automatiquement une *MethodArgumentNotValidException*.

L'objectif sera de la catch dans le gestionnaire d'exception global.

java

 Copier le code

```
import org.springframework.validation.annotation.Validated;
import javax.validation.Valid;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public User createUser(@Valid @RequestBody UserDTO userDTO) {
        // Logique de création d'utilisateur
    }
}
```

4. Gestion des exceptions et validation des données

Validation dans les contrôleurs :

Pour appliquer la validation, il suffit d'annoter le paramètre d'entrée avec **@Valid** dans le contrôleur.

En cas d'échec, Spring Boot déclenche automatiquement une *MethodArgumentNotValidException*.

L'objectif sera de la catch dans le gestionnaire d'exception global.

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;

import java.util.HashMap;
import java.util.Map;

@RestControllerAdvice
public class ValidationExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public Map<String, String> handleValidationExceptions(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach((error) -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage);
        });
        return errors;
    }
}
```

4. Gestion des exceptions et validation des données

Conclusion en bonnes pratiques :

- ▶ Gestion centralisée des exceptions avec @ControllerAdvice et @ExceptionHandler
- ▶ Utiliser des exceptions personnalisées
- ▶ Validation des données avec Bean Validation
 - ▶ Sur les modèles de données
 - ▶ Sur les entrées de contrôleur
- ▶ Personnaliser les messages d'erreurs

Ces pratiques garantissent une gestion propre, cohérente et sécurisée des erreurs dans les applications Spring Boot. Les utilisateurs recevront des messages clairs, tandis que le code reste structuré et maintenable

5. Documentation et tests

Documentation :

Automatique avec **Swagger** : il suffit d'ajouter la dépendance **springdoc-openapi-ui**.

Accessible via l'endpoint `/swagger-ui.html`.

Tests unitaires :

Junit pour tester unitairement le code

@WebMvcTest pour tester les contrôleurs

@DataJpaTest pour tester la persistance (database)

```
1 openapi: 3.1.0
2 info:
3   title: Swagger API
4   version: 1.0.0
5 servers:
6   - url: https://api.swagger.io
7 paths:
8   /resources:
9     get:
10      tags:
11        - API Resources
12      summary: Retrieve all resources based on filter criteria
13      description: Fetches a list of all available resources.
14      responses:
15        '200':
16          description: A
17          content:
18            application/json:
19              schema:
20                type: array
21                items:
22                  $ref: '#/components/schemas/Resource'
23      post:
24        tags:
25          - API Resources
26        summary: Create a new resource
27        description: Create a new resource
28        requestBody:
29          required: true
30          content:
31            application/json:
32              schema:
33                $ref: '#/components/schemas/Resource'
```

Servers

https://api.swagger.io

API Resources

GET /resources Retrieve all resources based on filter criteria

POST /resources Create a new resource

GET /resources/{resourceId} Retrieve a resource

PUT /resources/{resourceId} Update a resource

PATCH /resources/{resourceId} Partially update a resource

DELETE /resources/{resourceId} Delete a resource

OPTIONS /resources/{resourceId} Retrieve options

HEAD /resources/{resourceId} Check resource presence

Partie 3 - Introduction à Spring Data JPA

- ▶ 1. Introduction à Spring Data JPA
- ▶ 2. Configuration de Spring Data JPA dans Spring Boot
- ▶ 3. Définition des Entités
- ▶ 4. Les Repositories Spring Data
- ▶ 5. Requêtes avancées avec Spring Data JPA
- ▶ 6. Pagination et tri
- ▶ 7. Transactions et gestion de la persistance
- ▶ 8. Gestion des exceptions dans JPA

1. Introduction à Spring Data JPA

Qu'est-ce que JPA ? : Java Persistence API (JPA) est une spécification Java pour la gestion de la persistance des données (par exemple, le stockage et la récupération d'objets dans une base de données relationnelle). JPA simplifie l'interaction avec la base de données en représentant les tables comme des objets Java.


Pourquoi Spring Data JPA ? : Spring Data JPA simplifie encore plus l'utilisation de JPA en fournissant une infrastructure prête à l'emploi pour créer des repositories, ce qui rend les opérations de persistance très faciles et maintenables.

2. Configuration de Spring Data JPA dans Spring Boot

Datasource et propriétés : Par défaut, Spring Boot détecte automatiquement les dépendances JPA et configure Hibernate (un framework ORM populaire).

Configuration dans *application.properties* :

properties

 Copier le code

```
spring.datasource.url=jdbc:mysql://localhost:3306/nom_de_la_base  
spring.datasource.username=utilisateur  
spring.datasource.password=mot_de_passe  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.show-sql=true
```

La propriété *ddl-auto=update* permet de mettre à jour automatiquement le schéma de la base de données en fonction des entités.

3. Définition des entités

Création d'une Entité : Une entité représente une table dans la base de données.

```
java Copier le code

import javax.persistence.*;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;


    @Column(nullable = false, unique = true)
    private String email;

    // Getters et setters
}
```

3. Définition des entités

Mapping des relations : Spring Data JPA permet aussi de gérer les relations entre tables (ex. : **@OneToMany**, **@ManyToOne**).

java

 Copier le code

```
@OneToMany(mappedBy = "user")  
private List<Order> orders;
```

4. Les Repositories Spring Data

Création d'un Repository : En étendant *JpaRepository*, Spring génère automatiquement des méthodes CRUD pour manipuler les données de la table.

```
java Copier le code

import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    // Méthodes de recherche personnalisées peuvent être ajoutées ici
    User findByEmail(String email);
}
```

Recherche personnalisée : Par convention de nommage, Spring génère des requêtes dynamiques, comme *findByEmail*.

5. Requêtes Avancées avec Spring Data JPA

Requêtes JPQL avec @Query : utiliser **@Query** pour définir les requêtes JPQL.

```
java Copier le code  
  
import org.springframework.data.jpa.repository.Query;  
  
@Query("SELECT u FROM User u WHERE u.email = ?1")  
User findByEmailAddress(String email);
```

Requêtes natives : On peut également exécuter des requêtes SQL directement :

```
java Copier le code  
  
@Query(value = "SELECT * FROM users WHERE email = ?1", nativeQuery = true)  
User findByEmailNative(String email);
```


6. Pagination et Tri

Utilisation de *Pageable* et *Sort* : Spring Data JPA permet de retourner des pages de résultats et d'ajouter un tri simple.

```
java Copier le code  
  
import org.springframework.data.domain.Page;  
import org.springframework.data.domain.Pageable;  
  
Page<User> findAll(Pageable pageable);
```

Pagination dans le contrôleur :

```
java Copier le code  
  
@GetMapping("/users")  
public Page<User> getUsers(Pageable pageable) {  
    return userRepository.findAll(pageable);  
}
```

7. Transactions et gestion de la persistance

Annotation **@Transactional** : Permet de gérer les transactions pour garantir la cohérence des données.

```
java Copier le code

import org.springframework.transaction.annotation.Transactional;

@Service
public class UserService {

    @Transactional
    public void registerUser(User user) {
        // Logique de création de l'utilisateur
    }
}
```

Explication : Les méthodes annotées avec **@Transactional** seront exécutées dans une transaction, et en cas d'exception, la transaction sera annulée.

7. Transactions et gestion de la persistance

Une **transaction** en informatique (et particulièrement dans les bases de données) est une **unité de travail logique** qui regroupe une ou plusieurs opérations (comme des requêtes SQL) exécutées de manière atomique.

Cela signifie que toutes les opérations d'une transaction doivent être exécutées avec succès pour que les changements soient appliqués dans la base de données. Si une seule des opérations échoue, toutes les autres doivent être annulées (rollback), laissant la base dans un état cohérent.

7. Transactions et gestion de la persistance

Caractéristiques d'une transaction

Une transaction respecte les propriétés ACID :

1. **Atomicité** : Toutes les opérations dans une transaction sont réalisées complètement ou pas du tout. Si une partie échoue, tout est annulé.
2. **Cohérence** : La transaction garantit que la base de données passe d'un état cohérent à un autre état cohérent.
3. **Isolation** : Les transactions en cours sont isolées les unes des autres, empêchant les modifications non validées d'affecter d'autres transactions.
4. **Durabilité** : Une fois qu'une transaction est validée (commit), ses modifications sont permanentes, même en cas de panne.

7. Transactions et gestion de la persistance

Caractéristiques d'une transaction

Une transaction respecte les propriétés ACID :

1. **Atomicité** : Toutes les opérations dans une transaction sont réalisées complètement ou pas du tout. Si une partie échoue, tout est annulé.
2. **Cohérence** : La transaction garantit que la base de données passe d'un état cohérent à un autre état cohérent.
3. **Isolation** : Les transactions en cours sont isolées les unes des autres, empêchant les modifications non validées d'affecter d'autres transactions.
4. **Durabilité** : Une fois qu'une transaction est validée (commit), ses modifications sont permanentes, même en cas de panne.

7. Transactions et gestion de la persistance

Caractéristiques d'une transaction

Une transaction respecte les propriétés ACID :


1. **Atomicité** : Toutes les opérations dans une transaction sont réalisées complètement ou pas du tout. Si une partie échoue, tout est annulé.
2. **Cohérence** : La transaction garantit que la base de données passe d'un état cohérent à un autre état cohérent.
3. **Isolation** : Les transactions en cours sont isolées les unes des autres, empêchant les modifications non validées d'affecter d'autres transactions.
4. **Durabilité** : Une fois qu'une transaction est validée (commit), ses modifications sont permanentes, même en cas de panne.

8. Gestion des exceptions dans JPA

Exceptions spécifiques : En cas de violation d'une contrainte de base de données, Spring JPA peut lancer des exceptions (par exemple *DataIntegrityViolationException*).

Gestion des exceptions :

java

 Copier le code

```
@ExceptionHandler(DataIntegrityViolationException.class)
@ResponseStatus(HttpStatus.CONFLICT)
public String handleDataIntegrityViolation(DataIntegrityViolationException ex) {
    return "Contrainte de base de données violée : " + ex.getMessage();
}
```

Exemple global :

Création d'une entité User (JPA) :

```
java Copier le code

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    // Getters et setters
}
```


Exemple global :

Création d'une entité User (Mongo) :

```
@Document(collection = "car")
public class Car {


    @MongoId
    private UUID id;
    private String brand;
    private String color;
    private int horsepower;

}
```

Exemple global :

Définition du Repository (JPA) :

java

 Copier le code

```
public interface UserRepository extends JpaRepository<User, Long> {  
    User findByEmail(String email);  
}
```

Exemple global :

Définition du Repository (Mongo) :

```
@Repository
public interface CarDao extends MongoRepository<Car, UUID> {

    List<Car> findAllByBrand(String brand);

}
```

Exemple global :

Création du Service :

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public User createUser(User user) {
        return userRepository.save(user);
    }


    public User getUserById(Long id) {
        return userRepository.findById(id).orElseThrow(() -> new ResourceNotFoundException("User not found"));
    }

    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

Exemple global :

Création du Controller :

java

 Copier le code

```
@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }

    @GetMapping("/{id}")
    public User getUser(@PathVariable Long id) {
        return userService.getUserById(id);
    }
}
```

Conclusion

Spring Data JPA est un module essentiel de l'écosystème Spring, conçu pour simplifier la gestion de la persistance dans les applications utilisant des bases de données relationnelles. Grâce à son intégration avec JPA et Hibernate, il offre :

Simplicité et efficacité : Les interfaces comme *JpaRepository* permettent de réaliser des opérations CRUD et des requêtes complexes avec un minimum de code.

Réduction du code boilerplate : Les conventions de nommage et l'auto-génération des requêtes éliminent le besoin d'écrire des DAO manuellement.

Puissance et flexibilité : Les annotations JPA, les requêtes personnalisées (JPQL ou SQL natif), et les fonctionnalités avancées comme la pagination, le tri, et la gestion des relations rendent le module très complet.

Intégration Spring Boot : Avec l'auto-configuration, la configuration des bases de données et des transactions est pratiquement automatisée, ce qui accélère le développement.

Conclusion

En résumé, **Spring Data JPA** est une solution robuste et productive qui répond aux besoins des applications modernes en matière d'accès aux données.

Sa courbe d'apprentissage est rapide, surtout grâce à l'uniformité de l'approche Spring, et il s'adapte aussi bien aux petites applications qu'aux projets complexes nécessitant des fonctionnalités avancées.

Il existe évidemment des solutions équivalentes pour les autres systèmes de BDD comme par exemple **Spring Data MongoDB** qui fonctionne de la même manière avec quelques spécificités liées à la différence de technologie, mais facile à prendre en main quand on connaît **Spring Data JPA**.

Partie 4 – Les tests

- ▶ 1. Introduction à la stratégie de tests
- ▶ 2. Types de tests applicables à une application Spring Boot
- ▶ 3. Configuration des tests dans Spring Boot
- ▶ 4. Tests unitaires des couches JPA
- ▶ 5. Tests d'intégration pour la persistance des données
- ▶ 6. Utilisation de bases de données en mémoire pour les tests
- ▶ 7. Bonnes pratiques pour les tests liés à Spring Boot et JPA

1. Introduction à la stratégie de tests

Pourquoi tester ?

- Garantir le bon fonctionnement de l'application.
- Prévenir les régressions lors des évolutions du code.
- Assurer la qualité du code.

Spécificités des tests pour JPA :

- Valider les opérations CRUD.
- Vérifier le bon fonctionnement des requêtes personnalisées.
- Tester la gestion des transactions et des relations.

2. Types de tests applicables

Tests unitaires :

- Tester des méthodes spécifiques en isolant leur logique.
- Pas de dépendance avec une base de données réelle.

Tests d'intégration :

- Vérifier l'interaction entre les différentes couches (service, repository, base de données).
- Nécessite une base de données en mémoire ou réelle.


Tests de bout en bout (end-to-end) :

- Tester l'application complète via des API REST.
- Simule un scénario utilisateur du début à la fin.

3. Configuration des tests dans Spring Boot

Spring Boot simplifie les tests grâce à ses annotations et fonctionnalités intégrées.

xml

 Copier le code

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>test</scope>
</dependency>
```

3. Configuration des tests dans Spring Boot

Spring Boot simplifie les tests grâce à ses annotations et fonctionnalités intégrées.

Quelques annotations importantes :

@SpringBootTest : Charge le contexte Spring complet pour les tests d'intégration.

@DataJpaTest : Fournit une configuration minimale pour tester la couche JPA.

@MockBean : Permet de simuler des dépendances dans les tests.

4. Tests unitaires des couches JPA

Tester un Repository

Les tests unitaires des repositories peuvent être réalisés avec des bases en mémoire comme H2.

```
@DataJpaTest
public class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    void testSaveAndFindUser() {
        User user = new User();
        user.setName("John Doe");
        user.setEmail("john.doe@example.com");

        userRepository.save(user);

        User found = userRepository.findByEmail("john.doe@example.com");
        assertNotNull(found);
        assertEquals(found.getName(), "John Doe");
    }
}
```

4. Tests unitaires des couches JPA

Mocking avec Mockito

Les tests unitaires peuvent aussi simuler les interactions sans accéder à une base de données réelle.

```
public class UserServiceTest {

    private final UserRepository userRepository = Mockito.mock(UserRepository.class);
    private final UserService userService = new UserService(userRepository);

    @Test
    void testFindUserByEmail() {
        User user = new User();
        user.setName("Jane Doe");
        user.setEmail("jane.doe@example.com");

        when(userRepository.findByEmail("jane.doe@example.com")).thenReturn(user);


        User result = userService.findUserByEmail("jane.doe@example.com");
        assertThat(result.getName()).isEqualTo("Jane Doe");
    }
}
```

5. Tests d'intégration pour la persistance

Charger une base en mémoire

Configure H2 comme base pour les tests dans *application.properties* :

properties

 Copier le code

```
spring.datasource.url=jdbc:h2:mem:testdb  
spring.datasource.driver-class-name=org.h2.Driver  
spring.jpa.hibernate.ddl-auto=create-drop
```

On pourra aussi utiliser Testcontainers avec Docker par exemple afin d'utiliser des bases « réelles ».

5. Tests d'intégration pour la persistance

```
@SpringBootTest
@Transactional
public class UserIntegrationTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    void testUserPersistence() {
        User user = new User();
        user.setName("Alice");
        user.setEmail("alice@example.com");

        userRepository.save(user);

        User found = userRepository.findByEmail("alice@example.com");
        assertThat(found).isNotNull();
        assertThat(found.getName()).isEqualTo("Alice");
    }
}
```


6. Bonnes pratiques pour les tests Spring Boot et JPA

Isoler les tests :

- Les tests doivent être indépendants les uns des autres.
- Utiliser des bases en mémoire ou des conteneurs (comme Testcontainers).

Mocker intelligemment :

- Simuler des dépendances avec Mockito pour les tests unitaires.
- Ne pas utiliser de mocks dans les tests d'intégration.

Nettoyer les bases :

- Utiliser @Transactional pour que les données soient annulées après chaque test.
- Préférer des bases temporaires pour éviter d'impacter une base réelle.

Utiliser des jeux de données :

- Charger des données initiales pour des tests reproductibles (*data.sql* ou @Sql)

Couvrir tous les cas critiques :

- Tester les requêtes complexes, les transactions, et les erreurs attendues.

6. Bonnes pratiques pour les tests Spring Boot et JPA

Mais aussi :

- Prioriser les tests unitaires pour des feedbacks rapides.
- Isoler les tests en utilisant des slices comme **@DataJpaTest** ou **@WebMvcTest**.
- Utiliser des bases de données légères comme H2 pour les tests locaux, mais valider aussi avec des bases proches de la production.
- Tester les scénarios d'échec (ex : contraintes, violations de relations).

Conclusion

Les tests sont une partie essentielle du développement d'une application Spring Boot avec JPA.

En combinant des tests unitaires et d'intégration, on peut garantir la fiabilité de la couche de persistance.

Spring Boot offre des outils et des annotations puissantes pour simplifier la configuration et l'exécution des tests.

Une stratégie bien pensée améliore la qualité du code, tout en réduisant le risque de bugs en production.

Partie 4.1 – Bonus – le BDD, Gherkin & Cucumber

- ▶ 1. Introduction au BDD (Behavior-Driven Development)
- ▶ 2. Présentation du langage Gherkin
- ▶ 3. Introduction à Cucumber
- ▶ 4. Configurer Cucumber dans un projet Spring Boot
- ▶ 5. Écrire et exécuter des tests avec Cucumber
- ▶ 6. Bonnes pratiques pour les tests BDD avec Gherkin et Cucumber

1. Introduction au BDD

Behavior-Driven Development (BDD) :

Approche de développement orientée sur le comportement fonctionnel attendu.

Implique la collaboration entre développeurs, testeurs, et parties prenantes (ex. : PO, business analysts).

Pourquoi BDD ?

- ▶ Rendre les spécifications compréhensibles par tous.
- ▶ Se concentrer sur les fonctionnalités critiques.
- ▶ Faciliter la validation par les utilisateurs.

Outils clés du BDD : **Gherkin** (pour décrire) et **Cucumber** (pour exécuter).

2. Présentation du langage Gherkin

Qu'est-ce que Gherkin ?


- ▶ Langage simple et structuré pour écrire des spécifications sous forme de scénarios.
- ▶ Utilise une syntaxe proche du langage naturel.
- ▶ Multilingue, supporte plus de 70 langues.

Structure d'un scénario Gherkin :

- ▶ **Feature** : Décrit une fonctionnalité à tester.
- ▶ **Scenario** : Définit un cas précis d'utilisation ou de test.
- ▶ **Given/When/Then** : Étapes du scénario.

2. Présentation du langage Gherkin

gherkin

 Copier le code

Feature: Gestion des utilisateurs

As an admin

I want to manage users

So that I can keep the application secure

Scenario: Ajouter un nouvel utilisateur

Given un administrateur est connecté

When l'administrateur crée un utilisateur avec le nom "John Doe"

Then l'utilisateur "John Doe" est enregistré dans le système

Given : Décrit le contexte ou l'état initial.

When : Décrit une action ou un événement déclencheur.

Then : Décrit le résultat ou l'état attendu.

3. Introduction à Cucumber

Qu'est-ce que Cucumber ?

- ▶ Outil open-source pour exécuter des scénarios écrits en Gherkin.
- ▶ Relie les scénarios fonctionnels à des étapes de test automatisées (mappings entre Gherkin et Java).

Cycle d'utilisation :

- ▶ Écrire un scénario en Gherkin.
- ▶ Associer chaque étape (*Given*, *When*, *Then*) à une implémentation en java.
- ▶ Exécuter les scénarios avec Cucumber pour valider les fonctionnalités.

4. Configurer Cucumber dans un projet Spring Boot

Ajouter les dépendances :

```
xml Copier le code

<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>7.14.0</version>
</dependency>
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-spring</artifactId>
  <version>7.14.0</version>
</dependency>
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-junit</artifactId>
  <version>7.14.0</version>
  <scope>test</scope>
</dependency>
```

4. Configurer Cucumber dans un projet Spring Boot

Configurer les tests

Crée un fichier de test Cucumber, qui servira de point d'entrée.

```
import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;

import static io.cucumber.junit.platform.engine.Constants.GLUE_PROPERTY_NAME;


@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features") // Dossier contenant les fichiers .feature
@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value = "com.example.steps") // Package des étapes
public class CucumberTest {
}
```

5. Écrire et exécuter des tests avec Cucumber

Créer un fichier .feature

Dans `src/test/resources/features`, créer un fichier `user-management.feature`.

gherkin

 Copier le code

Feature: Gestion des utilisateurs

Scenario: Ajouter un utilisateur

Given un administrateur est connecté

When l'administrateur crée un utilisateur avec le nom "John Doe"

Then l'utilisateur "John Doe" est enregistré dans le système

5. Écrire et exécuter des tests avec Cucumber

Créer des étapes en Java

Dans `src/test/java/com/example/steps`, implémenter les étapes définies dans le scénario.

```
gherkin Copier le code

Feature: Gestion des utilisateurs

Scenario: Ajouter un utilisateur
  Given un administrateur est connecté
  When l'administrateur crée un utilisateur avec le nom "John Doe"
  Then l'utilisateur "John Doe" est enregistré dans le système
```

En lançant les tests comme des tests unitaires, Cucumber détecte automatiquement les fichiers et exécute les étapes correspondantes.

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.springframework.beans.factory.annotation.Autowired;
import static org.assertj.core.api.Assertions.assertThat;

public class UserSteps {

    @Autowired
    private UserService userService;

    private User createdUser;

    @Given("un administrateur est connecté")
    public void unAdministrateurEstConnecte() {
        // Simuler la connexion de l'administrateur
        userService.setAdminLoggedIn(true);
    }

    @When("l'administrateur crée un utilisateur avec le nom {string}")
    public void createNewUser(String userName) {
        createdUserName = new User(userName);
        userService.createUser(userName);
    }

    @Then("l'utilisateur {string} est enregistré dans le système")
    public void verifyUserExists(String userName) {
        assertThat(userService.findUserByName(userName)).isEqualTo(createdUser);
    }
}
```

6. Bonnes pratiques pour Gherkin et Cucumber

Langage clair et compréhensible :

- ▶ Rédiger les scénarios pour qu'ils soient lisibles par des non-développeurs.
- ▶ Limiter la longueur des scénarios pour éviter la confusion.

Réutilisation des étapes :

- ▶ Conserver des étapes génériques pour éviter les duplications (*Given, When, Then*)
- ▶ Regrouper les étapes par domaine fonctionnel dans des classes distinctes.

Couvrir des cas critiques :

- ▶ Commencer par tester les fonctionnalités principales.
- ▶ Ajouter des scénarios pour les cas limites et les erreurs attendues.

Automatiser dès le début :

- ▶ Vérifier que chaque scénario a une implémentation correspondante en Java.

Conclusion

Le langage **Gherkin** et l'outil **Cucumber** permettent de lier spécifications fonctionnelles et tests automatisés, favorisant une collaboration efficace entre les équipes techniques et métier.

En combinant les scénarios lisibles de Gherkin avec l'automatisation de Cucumber, les applications Spring Boot peuvent être testées de manière robuste, assurant la qualité des fonctionnalités tout en facilitant leur validation.

A vos devs !