

# Introduction

---

## Spring

Développé depuis plus de 10 ans, [Spring](#) est le framework le plus connu dans le développement d'applications d'entreprise du monde Java.

Historiquement, et ce qui est encore le coeur du framework, est l'implémentation du design pattern **Inversion Of Control** ou en français, **Injection de dépendances**

Ce design pattern a pour but de déléguer l'instanciation des classes de services au framework Spring. Ainsi, ce n'est plus vous qui créez les instances des classes par le mot clé **new**. Votre rôle consistera à créer les classes de services, à indiquer les relations entre elles (association). Le framework Spring s'occupera de créer les instances et d'injecter automatiquement les instances nécessaires à vos objects. Tel un maître d'orchestre, il donnera vie à votre programme.

A partir de ce principe, tout un écosystème est disponible et permet de couvrir quasiment tout le périmètre nécessaire au développement. Cet écosystème est matérialisé par un nombre important de bibliothèques.

Par exemple :

- Spring MVC pour le développement d'applications Web
- Spring Security pour la sécurisation
- Spring Data pour faciliter l'accès aux données
- Spring Batch pour le développement Batch
- Spring Intégration pour gérer les patterns d'entreprises tel que l'Event Driven Architecture
- ...

## Spring Boot

Le développement d'applications d'entreprise nécessitant le plus souvent l'utilisation combinée de plusieurs bibliothèques **Spring**. Comment garantir que la version des bibliothèques peuvent fonctionner correctement ? Comment permettre une configuration facile de celles-ci ?


[Spring Boot](#) a été conçu dès 2014 pour répondre à ces problématiques.

Il permet ainsi :

- de garantir une compatibilité entre les versions des bibliothèques
- de fournir un mécanisme de configuration facilement la paramétrage
- de permettre le développement d'applications dites **standalone**. L'application Java développée a ainsi tous les éléments pour fonctionner de manière autonome.

## Spring Initializr

[Spring Initializr](#) est un service internet permettant de sélectionner vos besoins et de générer automatiquement un projet Java prêt à développer.

 **spring** initializr

**Project**

☒ Gradle - Groovy

☐ Gradle - Kotlin

☐ Maven

**Language**

☒ Java

☐ Kotlin

☐ Groovy

**Spring Boot**

☐ 3.4.0 (SNAPSHOT)

☐ 3.4.0 (RC1)

☐ 3.3.6 (SNAPSHOT)

☒ 3.3.5

☐ 3.2.12 (SNAPSHOT)

☐ 3.2.11

**Project Metadata**

Group

com.example

Artifact

demo

Name

demo

Description

Demo project for Spring Boot

Package name

com.example.demo

Packaging

☒ Jar

☐ War

Java

☐ 23

☐ 21

☒ 17

**Dependencies**

ADD DEPENDENCIES... ⌘ + B

**Spring Web** WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Spring Data JPA** SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

**PostgreSQL Driver** SQL

A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

GENERATE ⌘ + ↵

EXPLORE CTRL + SPACE

SHARE...

Spring Initializr est directement intégré dans la version IntelliJ Ultimate

2 / 15

# Les Bases

---

Le but de Spring est d'instancier lui-même les classes de services et d'associer celles-ci entre.

Pour détecter qu'une classe doit être gérée par Spring, il est nécessaire de déclarer des annotations.

En partant d'un exemple, nous allons découvrir les annotations à utiliser.

Les premières versions de Spring, avant l'avènement des annotations, utilisaient des fichiers xml pour associer les instances entre elles.

## Les annotations @Service, @Component et @Repository

Positionnées au-dessus du nom de la classe, elle indique à Spring que la classe doit être instanciée et qu'elle sera utilisée au sein du contexte de l'application.

Fonctionnement quasiment identiques, la différence de noms est surtout présente pour indiquer le rôle que la classe va jouer.

- **@Component** : composant de base. Assure un fonctionnement global à l'application
- **@Service** : composant indiquant que cette classe va intégrer des règles de gestion liées au périmètre couvert par l'application
- **@Repository** : composant dont l'objectif est de fournir des accès aux données.

La classe **GareRepository** a pour objectif de charger à partir d'un fichier CSV une liste de gares en France. Associée à la restitution de données, cette classe est annotée avec l'annotation **@Repository**.

```
@Repository
public class GareRepository {

    private List<Gare> gares;

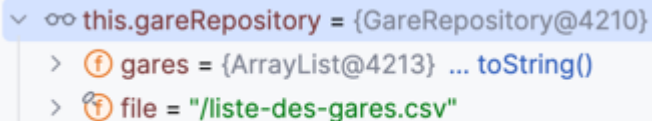
    private final String file;

    public GareRepository(@Value("${demo.file}") String file) {
        this.file = file;
    }

    @PostConstruct
    public void initialize() {
        try (var reader = new BufferedReader(new
InputStreamReader(GareRepository.class.getResourceAsStream(this.file)))) {
            var csvReader = new CsvToBeanBuilder<Gare>(reader)
                .withType(Gare.class)
                .withSeparator(';')
                .build();
            this.gares = csvReader.parse();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
    public Optional<Gare> findGareByVille(String ville) {  
        return gares.stream().filter(gare ->  
            gare.getLibelle().startsWith(ville)).findFirst();  
    }  
}
```

En mettant l'application en debug, nous constatons qu'une instance à cette classe a été créée par Spring.



```
▼ this.gareRepository = {GareRepository@4210}  
  > f gares = {ArrayList@4213} ... toString()  
  > i file = "/liste-des-gares.csv"
```


Quant à la classe **GareService**, elle représente la classe de service de l'application de gestion des gares. Cette classe a besoin d'accès aux données associées aux gares. Ainsi, lors du chargement de spring, l'instance de la classe **GareService** aura besoin de la classe **GareRepository** pour être pleinement fonctionnel. Nous déclarons ainsi un constructeur prenant en paramètre une instance de cette classe.

```
@Service  
public class GareService {  
  
    private static final Logger LOGGER =  
        LoggerFactory.getLogger(GareService.class);  
  
    private GareRepository gareRepository;  
  
    public GareService(GareRepository gareRepository) {  
        this.gareRepository = gareRepository;  
    }  
  
    public Optional<Gare> findGareByVille(String ville) {  
        LOGGER.debug("Recherche de la gare commençant par {}", ville);  
        return this.gareRepository.findGareByVille(ville);  
    }  
}
```

Lorsque Spring créera l'instance de la classe **GareService**, il réalisera les étapes suivantes :

1. Chargement de la classe **GareService**
2. Detection du constructeur necessitant en paramètre à la classe **GareRepository**
3. Création d'une instance à la classe **GareRepository**
4. Création d'une instance à la classe **GareService** en invoquant le constructeur **GareRepository**

Si nous nous mettons en debug, nous constaterons que l'instance à **GareService** possède bien une instance à la classe **GareRepository**.



## L'annotation @Autowired

L'annotation **@Autowired** indique à Spring que la propriété doit être renseignée (setlée) avec une instance de classe gérée par Spring.

```
@SpringBootApplication
public class Application implements CommandLineRunner {

    @Autowired
    GareService gareService;

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Recherche d'une gare à lille : ");

        System.out.println(this.gareService.findGareByVille("Lille").orElseThrow()
            .toString());
    }

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

Dans la classe **Application**, classe de lancement de notre application, l'instance de cette classe utilisera l'instance de la classe **GareService** pour invoquer la méthode **findGareByVille**.

Ainsi, en mettant l'annotation **@Autowired** sur cette propriété, Spring injectera l'instance la classe de service.

```
@Autowired
GareService gareService;
```

## Spring Boot et la configuration

Spring Boot fournit un mécanisme avancé de configuration d'applications à travers des fichiers de configuration, des paramètres d'environnement...

Chaque librairie associée à un domaine technique fournit par défaut un ensemble de variables afin de faciliter son fonctionnement.

Par exemple, la configuration des logs est facilité par des variables spécifique :

Fichier `application.yml`

```
logging:
  level:
    root: warn
    org.springframework.web: debug
    fr.epsi.spring: debug
```

Il est également possible d'ajouter ses propres variables :

```
demo.file: /liste-des-gares.csv
```

Pour utiliser ces variables, l'annotation `@Value` indiquera à Spring à injecter la valeur associée en allant chercher la valeur dans les éléments de configuration.

```
@Repository
public class GareRepository {

    private List<Gare> gares;

    private final String file;

    public GareRepository(@Value("${demo.file}") String file) {
        this.file = file;
    }
}
```

## L'annotation `@SpringBootApplication`

L'annotation `@SpringBootApplication` indique à Spring que la classe de chargement de l'application. On y trouve généralement la méthode **main** utilisée pour lancer l'application.

```
@SpringBootApplication
public class Application implements CommandLineRunner {

    @Autowired
    GareService gareService;

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Recherche d'une gare à Lille : ");

        System.out.println(this.gareService.findGareByVille("Lille").orElseThrow()
            .toString());
    }
}
```

```

    }

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

```

Il est impossible de positionner du code dans la méthode `main` pour utiliser les services que nous venons de créer. Dans notre exemple, la classe implémente l'interface **CommandLineRunner**.

Cette interface propose la méthode *run* à implémenter. Après le chargement de **Spring**, la méthode sera invoquée par l'application en passant en paramètre les arguments passés lors du changement de l'application (ie les mêmes que ceux de la méthode *main*)

Ainsi, l'exécution de cette classe lancera le programme :

```
mvn --quiet compile exec:java -
Dexec.mainClass=fr.epsi.spring.base.Application
/\ \ / ___' _ __ _(_)__ __ _ \ \ \ \ 
( ( )\___|'_||_||'_|\/_`_|\\ \\ \ 
\\ / ____)| |_| | || | || (_| | ) ) ) 
'|____|. _|| | | | |\__, | / / / / 
=====|_|=====|___/=/_/_/_/

:: Spring Boot ::                (v3.3.0)

2024-11-06T23:33:13.427+01:00 INFO 6471 --- [main]
fr.epsi.spring.base.Application : Starting Application using Java
21.0.2 with PID 6471 (/Users/nrousseau1/Documents/03.sources/epsi/java-introduction/20-Spring/22-Base/target/classes started by nrousseau1 in
/Users/nrousseau1/Documents/03.sources/epsi/java-introduction)
2024-11-06T23:33:13.430+01:00 DEBUG 6471 --- [main]
fr.epsi.spring.base.Application : Running with Spring Boot
v3.3.0, Spring v6.1.8
2024-11-06T23:33:13.430+01:00 INFO 6471 --- [main]
fr.epsi.spring.base.Application : No active profile set, falling
back to 1 default profile: "default"
2024-11-06T23:33:14.125+01:00 INFO 6471 --- [main]
fr.epsi.spring.base.Application : Started Application in 1.045
seconds (process running for 1.373)
Recherche d'une gare à lille :
2024-11-06T23:33:14.127+01:00 DEBUG 6471 --- [main]
f.epsi.spring.base.service.GareService : Recherche de la gare
commençant par Lille
Gare[adresse=fr.epsi.spring.base.domain.Adresse@3a2b2322, id=0,
libelle='Lille-St-Sauveur', supportFret=true, supportVoyageurs=false,
codeLigne='277100']
```

# Spring MVC

---

## Dépendance

Afin d'utiliser Spring MVC, il est nécessaire d'ajouter la dépendance suivante dans le fichier **pom.xml**.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>${spring.boot.version}</version>
</dependency>
```

## Les contrôleurs

Les classes interceptant les appels à http sont souvent appelées des *contrôleurs*.

Les annotations **@RestController** ou **Controller** sont à positionner sur les classes ayant ce rôle.

```
@RestController
public class GareController {

}
```

Ces classes interceptant des appels http doivent être associées à des urls. L'annotation **@RequestMapping** permet de définir l'url associée et également le contenu produit et/ou intercepté.

L'exemple précédent est complété afin d'indiquer que la classe *GareController* interceptera les urls de type **/v1/gares** et produira du contenu json.

```
@RestController
@RequestMapping(value = "/v1/gares", produces =
  MediaType.APPLICATION_JSON_VALUE)
public class GareController {

}
```

## Les méthodes

Les méthodes des classes contrôleurs vont être associées à des appels http. Pour chaque type de méthodes http, une annotation spécifique y sera associée.

Annotation	Méthode Http	Rôle
@GetMapping	GET	Récupérer un element ou une liste d'éléments



Annotation	Méthode Http	Rôle
@PostMapping	POST	Créer / modifier un élément
@PutMapping	PUT	Modifier une partie d'un élément
@DeleteMapping	DELETE	Supprimer un élément
@HeadMapping	HEAD	Verifier si un élément existe. Aucun contenu n'est retourné

La méthode **search** permet de récupérer un ensemble de gares.

```
@GetMapping
Page<Gare> search(Pageable page) {
    return gareService.findAll(page);
}
```

La méthode **searchById** permet de rechercher une gare à partir de son identifiant. L'url sera de la forme "{id}", l'id représentant l'identifiant de la gare. Spring va automatiquement extraire l'identifiant de l'url. Par exemple, à partir de l'url `http://localhost:8080/v1/gares/87741132`, Spring va extraire la valeur **87741132** et l'associer à la variable interceptant la variable du chemin id.

```
@GetMapping("/{id}")
Gare searchById(@PathVariable("id") Long id) {
    return gareService.findById(id).orElse(null);
}
```

Des annotations sont prévues pour chaque type de variables :

Annotation	Type de variable	exemple
@PathVariable	variable dans l'url	/v1/gares/12446
@RequestParam	paramètres de la requete	/v1/gares?page=2
@RequestBody	corps de l'appel	
@RequestHeader	Paramètre dans l'entête	

L'annotation doit être positionnée avant le paramètre de la méthode en indiquant si nécessaire le nom de la valeur extrait de l'appel.

Par défaut l'application s'exécute sur le port 8080. La variable **server.port** permet de changer le port. Cette variable peut être modifiée dans le fichier `application.yml` ou passée en paramètre

## Jouer avec les codes HTTP

Réaliser des API Rest suppose de respecter 2 règles :

- utiliser correctement les méthodes HTTP

- s'appuyer sur les codes HTTP pour donner un sens à la réponse.

Par exemple, si un élément n'existe pas, le service retournera le code http 404

Lors de la création d'un élément, le service retournera le code http 401 (CREATED) avec l'url afin de récupérer la ressource.

Si une erreur de données, l'application retournera un code du type 4XX.

Pour une erreur serveur non gérée, on sera le plus fréquemment sur une erreur du type 500.

Pour personnaliser les codes retournés, nous avons à notre possession:

- l'utilisation de la classe **ResponseEntity**
- l'utilisation de l'annotation **@ResponseStatus** lors de la déclaration des exceptions.

## La classe **ResponseEntity**

Au lieu de retourner directement l'objet en retour de la méthode interceptant une url, il est possible de retourner une instance de la classe **ResponseEntity** et en personnalisant :

- le code http
- le contenu retourné
- la personnalisation des entêtes http

Par exemple, le code suivant indique une manière pour la création d'une nouvelle ressource.

```
@PostMapping
ResponseEntity<Void> create(@RequestBody Gare gare) {
    this.gareService.save(gare);
    var createdObjectRequest =
    UriComponentsBuilder.fromPath("/v1/gares/{id}")
        .build(gare.getId());
    return ResponseEntity.created(createdObjectRequest).build();
}
```

Si nous créons la nouvelle gare,

```
curl -X 'POST' \
  'http://localhost:8080/v1/gares' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "libelle": "Nouvelle Gare",
    "supportFret": true,
    "supportVoyageurs": true,
    "codeLigne": "TER1",
    "adresse": {
      "commune": "EPSI City",
      "departement": "Nord"
    }
  }'
```

```
}  
}' -v
```

nous obtenons en réponse :

```
< HTTP/1.1 201  
< Location: /v1/gares/87988720  
< Content-Length: 0  
< Date: Tue, 12 Nov 2024 21:26:50 GMT
```

## L'annotation **@ResponseStatus**

L'annotation **@ResponseStatus** positionnée sur la classe d'une exception permet de définir le code http et le message en cas de la levée de l'exception.

Créons une nouvelle classe d'exception **NotFoundException** qui sera utilisée lorsqu'une gare ne sera pas trouvée.

```
@ResponseStatus(HttpStatus.NOT_FOUND)  
public class NotFoundException extends Exception{  
}
```

En modifiant la méthode **findById**, nous indiquons que la méthode peut lever l'exception :

```
@Operation(description = "Permet de rechercher une gare à partir de son  
id")  
@ApiResponses({  
    @ApiResponse(responseCode = "200",  
        content = {@Content(mediaType = "application/json",  
            schema = @Schema(implementation = Gare.class))},  
    @ApiResponse(responseCode = "404", description = "gare non  
trouvée", content = @Content())  
})  
@GetMapping("/{id}")  
Gare searchById(@PathVariable("id") Long id) throws NotFoundException{  
    return gareService.findById(id).orElseThrow(()-> new  
NotFoundException());  
}
```

En testant avec un identifiant de gara inconnu, nous obtenons l'erreur 404.

```
curl -X 'GET' \  
  'http://localhost:8080/v1/gares/9999999' \  
  -H 'accept: application/json' \  
  -v
```

```
< HTTP/1.1 404
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Tue, 12 Nov 2024 21:44:40 GMT
<
* Connection #0 to host localhost left intact
{"timestamp":"2024-11-12T21:44:40.988+00:00","status":404,"error":"Not Found","path":"/v1/gares/9999999"}%
```

Afin de réaliser un swagger contenant le plus d'informations sur les services, il est parfois nécessaire d'enrichir la description avec des annotations. Par exemple, l'annotation `@ApiResponse` permet de décrire les codes http pouvant être retournés.

L'annotation `@ResponseStatus` permet d' gérer des cas simples de retour http. Il est également possible d'utiliser les annotations `@ExceptionHandler` et `@ControllerAdvice` qui permettent de personnaliser plus finement le code http et le contenu retournés.

## swagger-ui

Les applications Java sont essentiellement utilisées pour créer des API Rest. Pour définir les contrats de services, c'est à dire les urls prises en charge par l'application, des documents swagger sont le plus souvent proposés.. Ces documents sont au format Json.

Il est assez facile d'intégrer des librairies permettant :

- de générer le document Json
- de fournir un visuel type swagger-ui

Pour spring boot 3, ajouter la dépendance :

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.5.0</version>
</dependency>
```

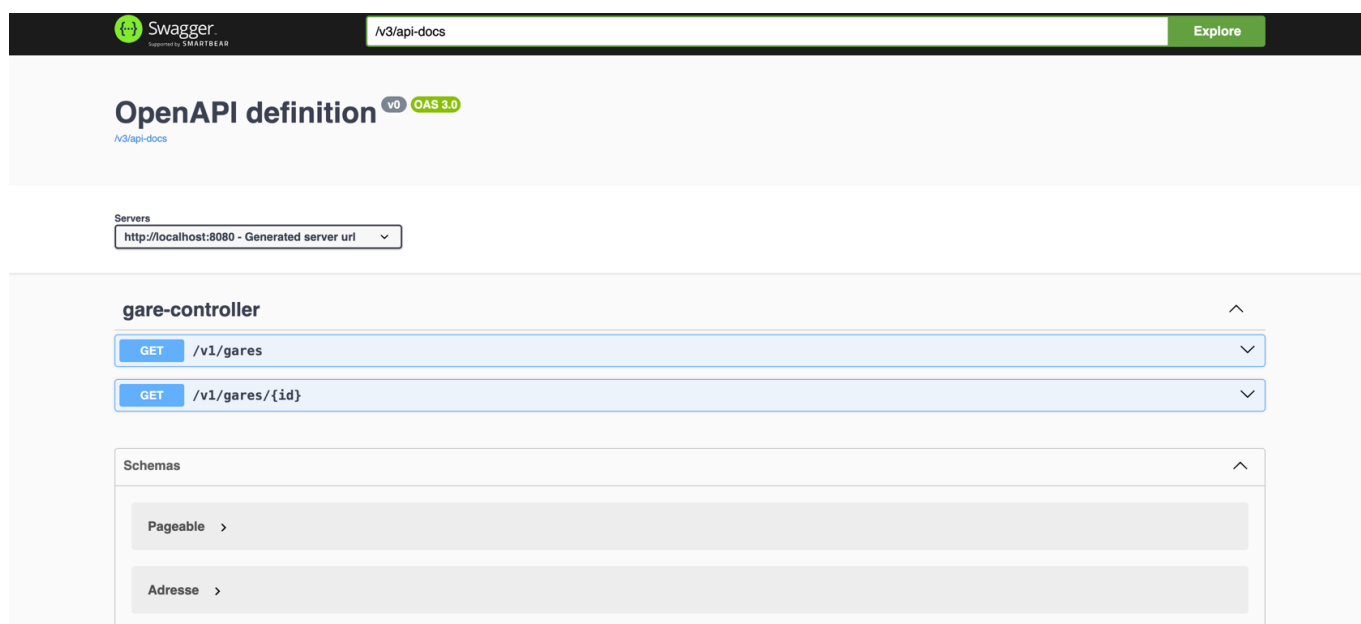
Au lancement de l'application, de nouveaux endpoints seront disponibles :

L'url `http://localhost:8080/v3/api-docs` permet de visualiser le document JSON de définitions des services :

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "OpenAPI definition",
    "version": "v0"
  },
  ...
}
```

```
"servers": [
  {
    "url": "http://localhost:8080",
    "description": "Generated server url"
  }
],
"paths": {
  "/v1/gares": {
    "get": {
      "tags": [
        "gare-controller"
      ],
      "description": "Permet de rechercher des gares",
      "...": "..."
    }
  }
}
```

L'url <http://localhost:8080/swagger-ui/index.html> permet de visualiser une page type swagger-ui



## Créer une application web autonome

Un des objectifs de spring boot est fournir une application au format jar totalement autonome ayant tout le nécessaire pour s'exécuter.

En exécutant la commande suivante, un fichier au format jar contenant tout le nécessaire au fonctionnement sera réalisé à la racine du répertoire **target**

```
mvn clean package spring-boot:repackage
```

La commande java permettra de lancer l'application :

```
java -jar target/23-SpringMVC-1.0-SNAPSHOT.jar  
fr.epsi.spring.demo.Application
```

