



SPRING BOOT

Découverte du Framework

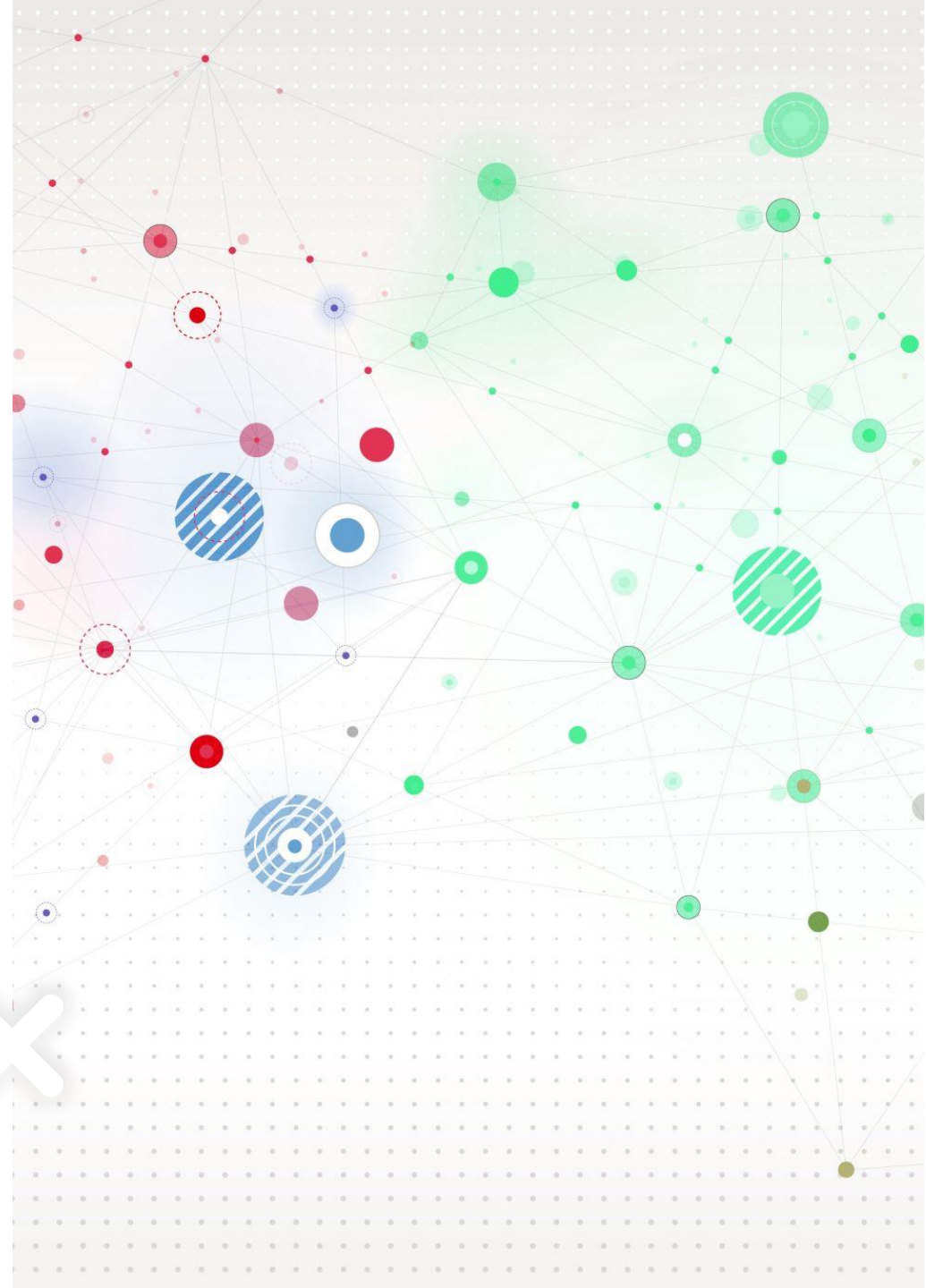


Table des matières.

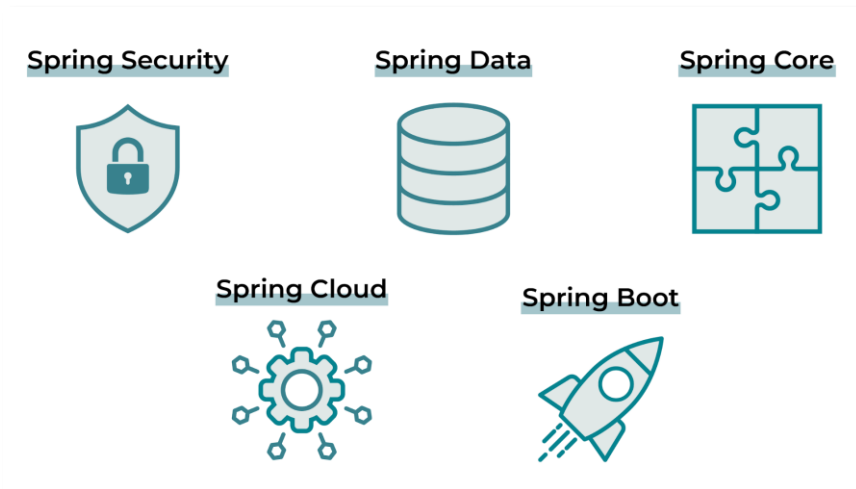
1. Présentation de Spring.
2. Avantages de Spring Boot.
3. Création d'un projet.
4. Composition du projet.
5. Structure du projet.
6. Paramétrages.
7. Coder le projet.
8. Tester et déployer.
9. Projet API.
10. Projet Web.
11. Thymeleaf.
12. Tester le projet.
13. Conclusion.

Présentation de Spring

Spring est un **framework**, c'est-à-dire un cadre de travail existant que les développeurs peuvent utiliser.

Il y a de nombreux composants au sein du **framework**, et après, à nous de faire l'assemblage.

<https://spring.io/projects/>



- Spring Security
 - Ce composant est l'un des plus critiques de Spring Framework. Il permet de gérer l'authentification, l'autorisation, mais aussi la sécurité des API.
 - <https://spring.io/projects/spring-security/>
- Spring Data
 - Ce composant permet de communiquer avec de nombreux types de bases de données.
 - <https://spring.io/projects/spring-data/>
- Spring Core
 - Ce composant est la base de l'écosystème de Spring.
 - <https://spring.io/projects/spring-framework/>
- Spring Cloud
 - Ce composant permet de répondre aux contraintes de l'architecture microservice.
 - <https://spring.io/projects/spring-cloud/>
- Spring Boot
 - Ce composant permet la mise en œuvre des autres composants (autoconfiguration, statuts de dépendances et endpoints Actuator)
 - <https://spring.io/projects/spring-boot/>

Avantages de Spring Boot

Spring est un écosystème avec un grand E ! À tel point que parfois ce framework peut même sembler trop rigide, trop encombrant ou trop complexe.

De plus, il contient de nombreux composants, et ces derniers ne s'utilisent pas de façon exclusive : dans la très grande majorité des projets, vous devrez utiliser plusieurs composants de Spring simultanément.

Par voie de conséquence, l'intégration de plusieurs composants Spring pour un même projet ajoute de la complexité.

Complexité qui sera croissante plus le projet prendra de l'importance !

- + **Spring Boot** ! Ce composant de Spring a été créé pour nous aider à utiliser Spring Framework.
 - C'est un composant au service des autres composants.
- + **Spring Boot** nous met à disposition les bons composants, nous permettant ainsi de les faire fonctionner ensemble.
 - Optimisation de la gestion des dépendances déduites avec **la version du framework utilisée**.
 - Automatisation de la configuration -> Autoconfiguration avec l'annotation **@SpringBootApplication**.
 - Gestion des propriétés avec **applications.properties**.
 - Monitoring et gestion du programme.
 - Déploiement simplifié car pas d'installation d'un serveur web comme Tomcat.

Pattern Dependency Injection

Spring Framework s'occupe de tout grâce à son **IoC container** !

Ce dernier est aussi appelé le **contexte Spring**. Il vous permettra de créer des objets dynamiquement, et de les injecter dans d'autres objets.

De plus, on pourra facilement modifier l'implémentation d'un objet, avec quasiment zéro impact sur les objets qui utilisent ce dernier.

Les mécanismes de l'**IoC container** rendront **votre code évolutif, performant et robuste** !

- + **IoC** est le sigle de **Inversion of Control**.
- + Cette expression indique un principe de programmation qui correspond au fait de déléguer à un framework le flux de construction et d'appels des objets.
- + La construction des objets de notre application va être déléguée à ce composant que l'on appelle un conteneur IoC (IoC container). Ce conteneur accueille un ensemble d'objets dont il a la responsabilité de gérer le cycle de vie.
- + **Le Spring Framework est avant tout un conteneur IoC.**
- + On peut résumer le rôle du Spring Framework en disant qu'il est responsable de la création des objets qui constituent l'ossature de notre application et qu'il s'assure que les dépendances entre eux sont correctement créées.
- + Comme son principe reste général, il va être possible d'utiliser le Spring Framework pour des types d'application très divers. Il faut simplement pouvoir laisser au Spring Framework la possibilité d'initialiser l'application au lancement en créant le conteneur IoC.
- + Dans la terminologie du Spring Framework, le conteneur IoC est constitué d'un ou de plusieurs contextes d'application.

Projet Spring Boot

Création d'un premier projet simple.

Projet Hello World

Voici les objectifs du projet :

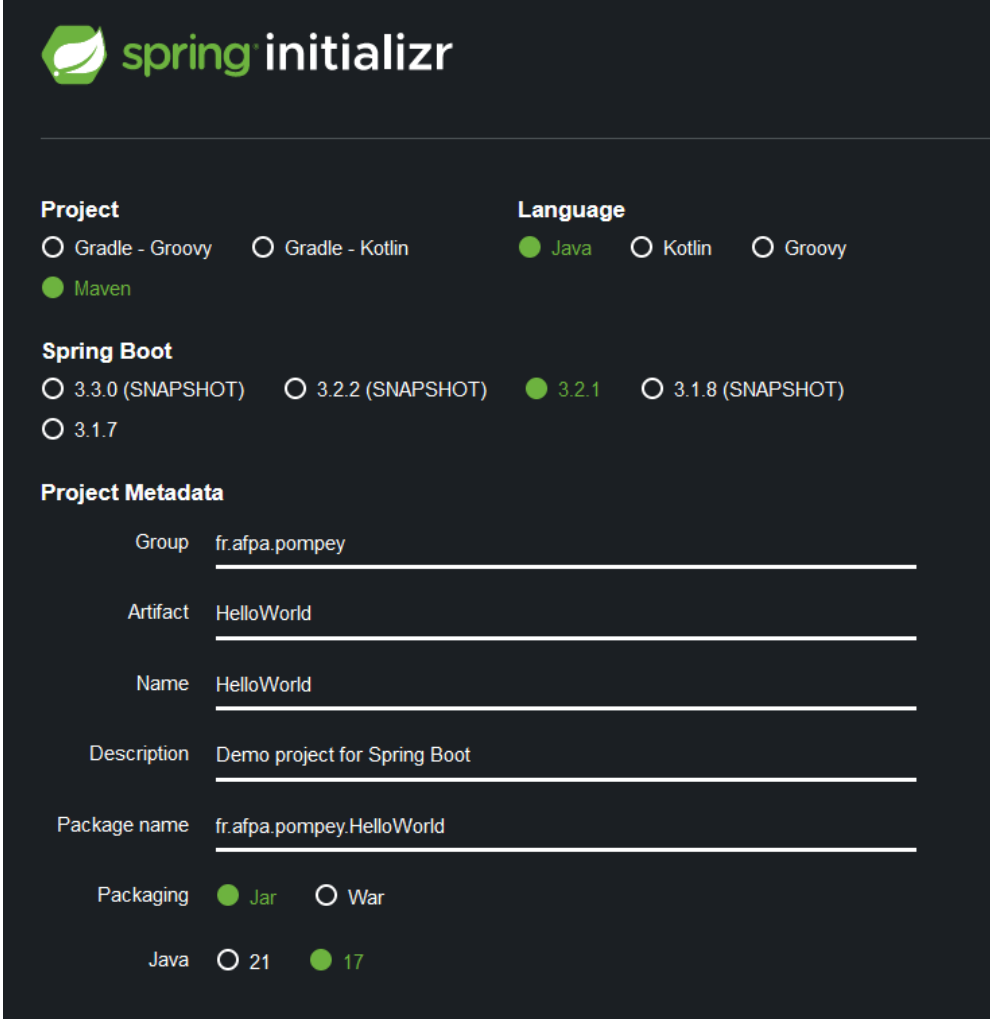
1. Créer le projet, c'est-à-dire générer la structure minimale.
2. Structurer et configurer le projet.
3. Écrire le code.
4. Tester et déployer.

- + La première étape implique de générer la base de votre projet.
- + Spring Boot nous fournit une base de travail que l'on peut nommer la structure minimale.
- + Spring Boot va vous demander un certain nombre d'informations, comme :
 - La version de Java,
 - Maven ou Gradle,
 - La version de Spring Boot,
 - des informations sur le projet (groupId, artifactId, nom du package),
 - Les dépendances.
- + Les dépendances sont gérées via des starters de dépendances qui sont des kits de dépendances.
 - Les starters sont préfixés par **spring-boot-starter**
 - *spring-boot-starter-core ;*
 - *spring-boot-starter-data-jpa ;*
 - *spring-boot-starter-security ;*
 - *spring-boot-starter-test ;*
 - *spring-boot-starter-web.*

Projet Hello World

Pour la création du projet, il existe plusieurs méthodes.

- Avec [Spring Initializr](https://start.spring.io/)
 - <https://start.spring.io/>
 - Depuis l'ide IntelliJ
- Avec Spring Tool Suite
 - [Spring Tools 4 for Eclipse](#)



The screenshot shows the Spring Initializr web interface. It has a dark theme with green accents. The 'Project' section has radio buttons for 'Gradle - Groovy', 'Gradle - Kotlin', 'Maven' (selected), and 'Groovy'. The 'Language' section has radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'. The 'Spring Boot' section has radio buttons for '3.3.0 (SNAPSHOT)', '3.2.2 (SNAPSHOT)', '3.2.1' (selected), and '3.1.8 (SNAPSHOT)'. Below these is a radio button for '3.1.7'. The 'Project Metadata' section contains text input fields for 'Group' (fr.afpa.pompey), 'Artifact' (HelloWorld), 'Name' (HelloWorld), 'Description' (Demo project for Spring Boot), and 'Package name' (fr.afpa.pompey.HelloWorld). At the bottom, there are radio buttons for 'Packaging' ('Jar' selected, 'War' unselected) and 'Java' ('21' unselected, '17' selected).

Projet Hello World

Depuis le site web de [Spring Initializr](#)

- Par défaut pas de dépendances car [spring-boot-starter](#) contient toutes les fonctionnalités de base.
- Génère un zip contenant votre projet qu'il vous suffit de placer dans votre espace de travail.
- Dans le pom.xml de votre projet, vous pouvez voir les dépendances par défaut que l'outil à installer.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.2.1</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>fr.afpa.pompey</groupId>
<artifactId>HelloWorld</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>HelloWorld</name>
<description>Demo project for Spring Boot</description>
<properties>
  <java.version>17</java.version>
</properties>
<dependencies> Edit Starters...
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

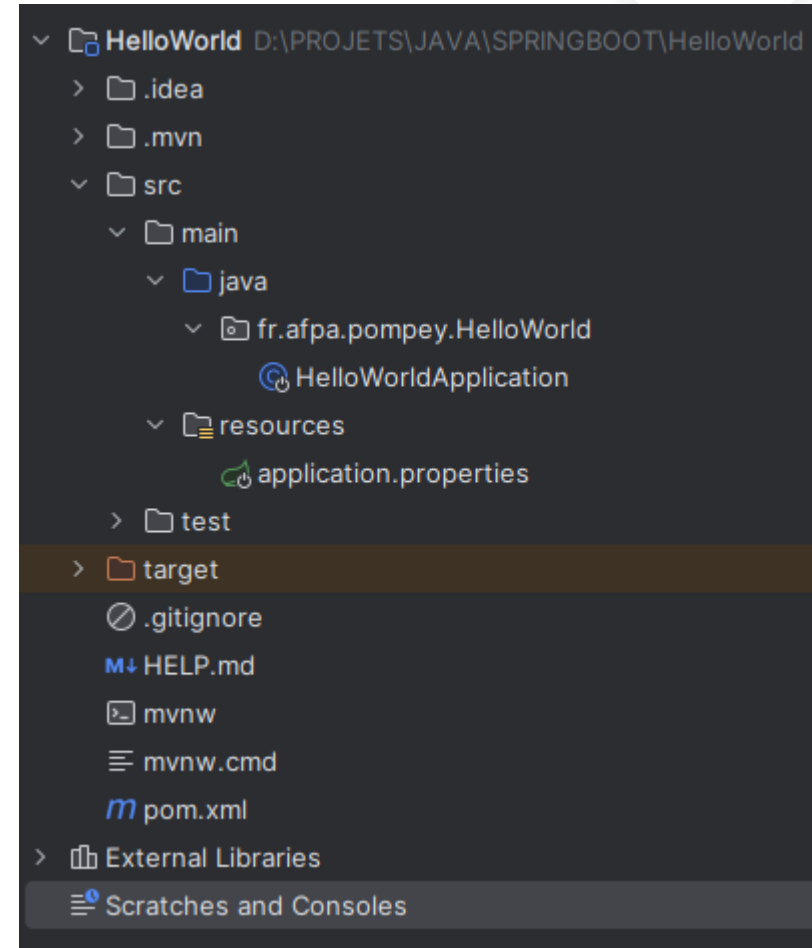
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Projet Hello World

L'étape 2 correspond au fait de structurer et configurer notre projet.

La structure minimale nous offre déjà une structure. Voici quelques points à noter :

- Nous retrouvons les fichiers liés à Maven (pom.xml, mvnw et mvnw.cmd).
- Nos sources sont organisées suivant le schéma standard :
 - `src/main/java` : contient les packages et les classes Java ;
 - `src/main/resources` : contient les fichiers ressources, tels que les fichiers de propriétés ou des templates (HTML, par exemple) ;
 - `src/test/java` : contient les classes Java de test.
- Comme tout projet Java/Maven, on retrouve également la JRE et les Maven Dependencies.



Composition du projet

Il y a deux classes Java et un fichier Propriétés.

- Le fichier de propriétés `application.properties` créé par défaut, mais vide pour le moment.
- `HelloWorldApplicationTests.java`, qui est la classe de test.
- `HelloWorldApplication.java`, qui est la classe principale de votre projet. dont 2 choses importantes :
 - La méthode `main`.
 - `@SpringBootApplication` qui est critique !

- + Une annotation, c'est-à-dire `@[nom de l'annotation]`, peut être ajoutée à une classe, une méthode, un attribut.
 - L'annotation influe sur le comportement du programme car elle fournit des métadonnées lors de la compilation.
 - Ces mêmes métadonnées seront utilisées lors de l'exécution.
- + Depuis la version 2.5 de Spring, de nombreuses annotations sont fournies, dont le but est de :
 - Permettre à l'IOC container d'utiliser nos classes.
 - Influencer sur le comportement de Spring.
- + L'annotation `@SpringBootApplication` va permettre à l'IOC container d'utiliser nos classes et d'influencer sur le comportement de Spring. `@SpringBootApplication` est un composé de 3 autres annotations :
 - `@SpringBootConfiguration` : la classe sera utilisée comme une classe de configuration.
 - `@EnableAutoConfiguration` : active la fameuse fonctionnalité d'autoconfiguration de Spring Boot.
 - `@ComponentScan` : active le "scanning" de classes dans le package de la classe et dans ses sous-packages. Sans cette annotation, l'IOC container ne tiendra pas compte de vos classes, même si vous avez ajouté une annotation sur celles-ci.

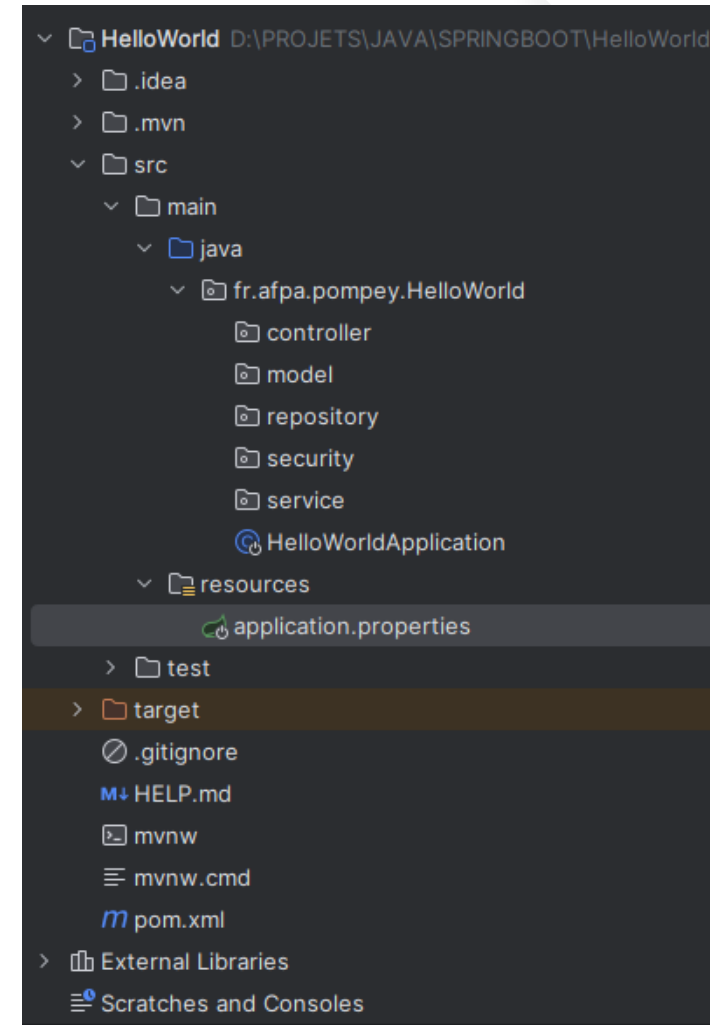
Structure du projet

Ici, nous respectons les principes acquis tout au long de la formation.

Package nos futures classes dans les packages dédiés.

Si on se base sur le modèle MVC par exemple, nous pourrions avoir :

- Un package Model
- Un package Controller
- Une package Vue
- Etc...



Paramétrages du projet

Une application Spring doit être paramétrable, c'est-à-dire que son comportement peut changer en fonction des paramètres fournis.

Pour rendre paramétrable une application, elle doit donc être capable de lire ces paramètres.

Ces paramètres sont dans des "sources de propriétés" (property sources), et sont définis par les gestionnaires de ces sources.

- + Spring est capable de lire ces sources de propriétés (sans interaction de notre part), et de rendre les propriétés disponibles sous forme de beans au sein du contexte Spring.
- + Parmi les sources de propriétés, il y a :
 - les propriétés de la JVM ;
 - les variables d'environnements du système d'exploitation ;
 - les arguments passés par la ligne de commande ;
 - les fichiers .properties ou .yaml (comme application.properties).
- <https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html>

```
# nom de l'application
spring.application.name=HelloWorld
# affiche uniquement les logs de niveau ERROR
# pour le package org.springframework
logging.level.org.springframework=error
```

Coder le projet

Après avoir structuré et configuré le projet, nous pouvons passer à l'étape de codage du projet.

La méthode main sera théoriquement là où on écrirait notre code dans un programme Java simple.

Mais en l'occurrence, cette dernière contient `SpringApplication.run(HelloWorldApplication.class, args)`

Cette instruction permet de démarrer notre application, et **ce n'est pas une bonne pratique d'ajouter autre chose dans la méthode main.**

- + Spring Boot fournit une interface nommée "CommandLineRunner".
- + En implémentant cette interface, la classe sera obligée de déclarer la méthode `public void run(String... args) throws Exception`.
- + À partir de là, si la classe est un bean (un bean est une classe au sein du contexte Spring), Spring Boot exécutera la méthode run à l'exécution du programme.
 - soit modifier la classe `HelloWorldApplication` afin qu'elle implémente `CommandLineRunner` et la méthode run, avec comme corps de méthode un `System.out.println("Hello World!");`
 - soit créer une nouvelle classe qui implémente `CommandLineRunner`, la méthode run (même corps de méthode), et qui aura une annotation `@Component` (au-dessus du nom de la classe).

```
@SpringBootApplication
public class HelloWorldApplication implements CommandLineRunner {

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

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Hello World");
    }
}
```

Manipulez des beans

1. Création d'un objet métier `HelloWorld` contenant un attribut `value`
2. Création d'un service `BusinessService` qui va instancier l'objet Métier et le retourner.
3. Modifier la classe `HelloWorldApplication` pour faire appel à nos classes.

Importants à noter :

1. `@Component` déclare la classe `BusinessService` comme Bean dans le context Spring

`@Autowired` permet l'injection de dépendances par l'IoC Container de Spring. Spring va chercher au sein de son contexte s'il existe un bean de type `BusinessService`.

- S'il le trouve, il va alors instancier la classe de ce bean et injecter cette instance dans l'attribut.
- S'il ne trouve pas de bean de ce type, Spring génère une erreur.

```
5 usages
public class HelloWorld {

    no usages
    private String value = "Hello World";

    @Override
    public String toString() {
        return this.value;
    }
}
```

```
2 usages
@Component
public class BusinessService {

    1 usage
    public HelloWorld getHelloWorld() {
        return new HelloWorld();
    }
}
```

```
@SpringBootApplication
public class HelloWorldApplication implements ApplicationRunner {

    @Autowired
    private BusinessService bs;

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

    @Override
    public void run(String... args) throws Exception {

        // simple écriture dans la console
        System.out.println("Hello World");

        HelloWorld hw = bs.getHelloWorld();
        System.out.println(hw);
    }
}
```

SPRING-BOOT - JBOV1

Tester et déployer

Pour conclure notre application HelloWorld, il nous reste 2 choses à faire :

- Tester notre application.
- Déployer notre application.

- + **Pour les tests**, il s'agit simplement d'écrire nos différents comme on le ferait en Java. Vous pouvez constater les dépendances de Junit5 dans les librairies installées.

```
> [Maven] org.junit.jupiter:junit-jupiter:5.10.1
> [Maven] org.junit.jupiter:junit-jupiter-api:5.10.1
> [Maven] org.junit.jupiter:junit-jupiter-engine:5.10.1
> [Maven] org.junit.jupiter:junit-jupiter-params:5.10.1
```

- + **Pour le déploiement**, l'environnement de Spring Boot suffit à lui-même.
- + Rappelons que parmi les avantages de Spring Boot, il y a sa facilité de déploiement car le JAR qui résulte de la compilation embarque tout.
- + Même le serveur Tomcat qui permet d'exécuter une application web est embarqué. Nul besoin d'installer un serveur Tomcat, il est déjà là !

Projet API

Créer une API avec Spring Boot



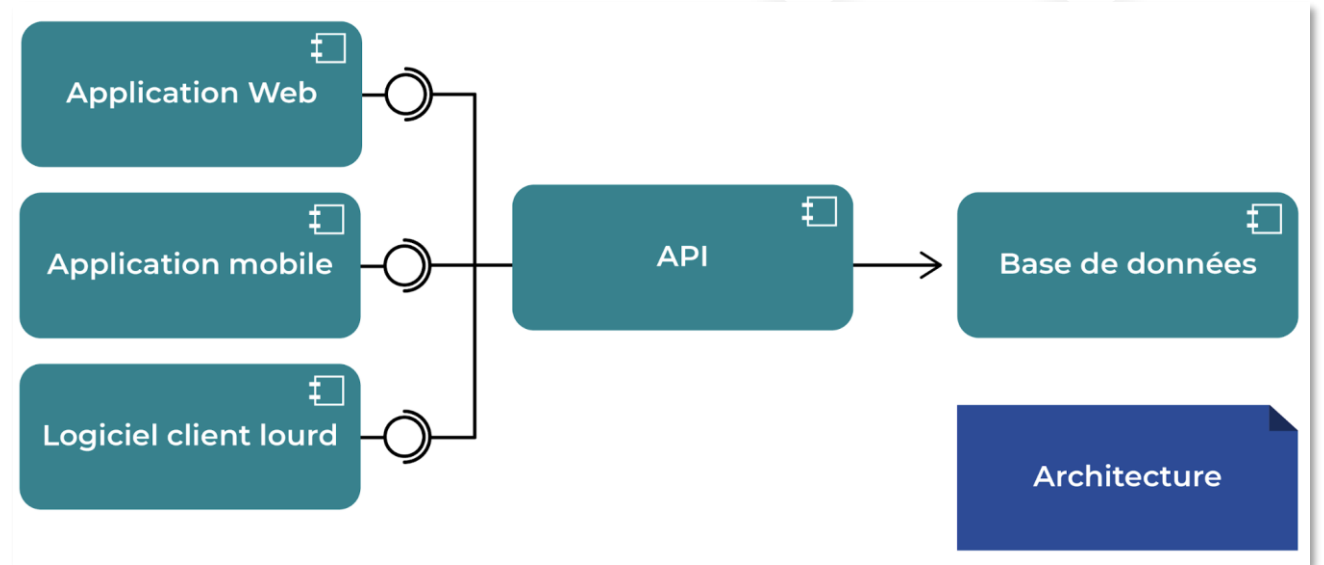
Objectifs

On souhaite mettre à disposition une API qui permettra à toutes les autres applications d'accéder aux mêmes données.

L'idée étant de gérer **des personnes**, l'API devra donc offrir un **CRUD** (Create, Read, Update, Delete) pour les données des personnes.

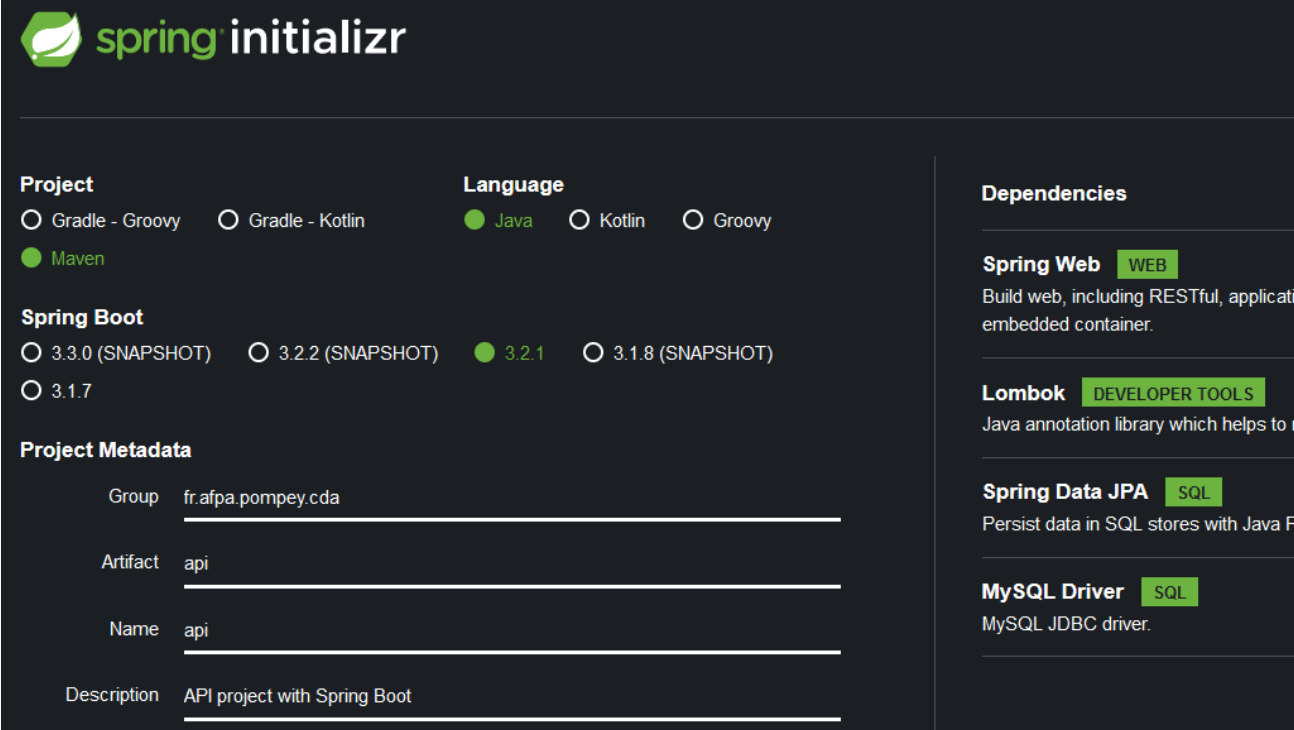
Notre API devra donc **exposer des endpoints** correspondant aux actions du CRUD, et **communiquer avec la base de données** pour récupérer ou modifier les informations des patients.

À noter que l'API sera de type **REST**.



Créer le projet

- Pour les dépendances :
 - **Spring Web** : comme la description l'indique, permet de faire du RESTful, ce qui correspond à notre API pour exposer des endpoints.
 - **Lombok** : librairie pour optimiser certaines classes.
 - **MySQL Driver** : pour les drivers MySQL.
 - **Spring Data JPA** : permet de gérer la persistance des données avec la base de données.



The screenshot shows the Spring Initializr web form with the following configuration:

- Project:** ☒ Maven
- Language:** ☒ Java
- Spring Boot:** ☒ 3.2.1
- Project Metadata:**
 - Group: fr.afpa.pompey.cda
 - Artifact: api
 - Name: api
 - Description: API project with Spring Boot
- Dependencies:**
 - ☒ Spring Web (WEB)
 - ☒ Lombok (DEVELOPER TOOLS)
 - ☒ Spring Data JPA (SQL)
 - ☒ MySQL Driver (SQL)

Configurer et structurer le projet

Notre structure minimale étant prête, il nous faut désormais :

- Structurer avec des packages,
- Configurer notre application.
- <https://spring.io/guides/gs/accessing-data-mysql/>

Couche	Objectif
controller	Réceptionner la requête et fournir la réponse
service	Exécuter les traitements métiers
repository	Communiquer avec la source de données
model	Contenir les objets métiers

```
spring.application.name=API

# port par défaut de tomcat
server.port=9000

# niveau de level logging
logging.level.root=error
# pour mon package
logging.level.fr.afpa.pompey.cda=info
# pour la BDD
logging.level.org.springframework.data=INFO
logging.level.org.springframework.jdbc.core.JdbcTemplate=DEBUG
# niveau de level logging pour TOMCAT
logging.level.org.springframework.boot.web.embedded.tomcat=INFO

# paramétrage de la BDD
spring.datasource.url=jdbc:mysql://localhost:3306/sparadrap
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
# modifie la base de données en fonction des entity pour le mode développement
# en production remettre à none
spring.jpa.hibernate.ddl-auto=update
# hibernate
# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
```

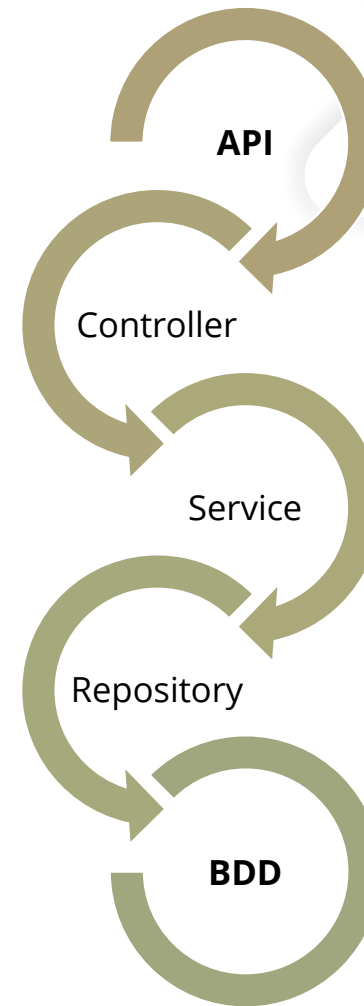
Interaction entre les couches

Au lancement de l'API, le contrôleur est chargé.

En fonction du mapping mis en place, il fait appel au Service.

Le service se charge alors d'orienter la demande vers le repository en fonction de la demande faite à l'API.

Enfin le repository contacte la Base de données pour récupérer la ou les données.



Création d'un contrôleur REST pour gérer les données

1. Création de l'entité Person

- `@Data` est une annotation Lombok se chargeant des getter et setter
- `@Entity` est une annotation qui indique que la classe correspond à une table de la base de données.
- `@Table(name="person")` indique le nom de la table associée.
- L'attribut id correspond à la clé primaire de la table, et est donc annoté `@Id` et est auto-incrémenté, par l'annotation `@GeneratedValue(strategy = GenerationType.AUTO)`.
- Id, est annoté avec `@Column` pour faire le lien avec le nom du champ de la table qui `idperson`.

```
18 usages  neojero *
@Data
@Entity
@Table(name= "Person")
public class Person {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name= "idperson")
    private int id;

    // @Column(name="firstname")
    private String firstname;

    // @Column(name="lastname")
    private String lastname;

}
```

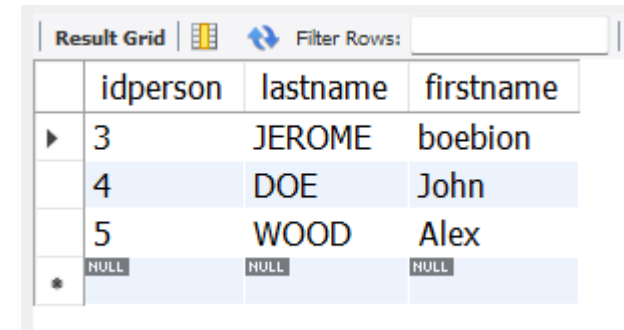
La table Person

J'ai fait le choix pour l'exemple, de créer une simple base de données MySQL nommée **Sparadrap** contenant une table **Person**.

Cette table possède juste 3 colonnes **lastname** et **firstname**.

Au lancement de l'API et afin de la tester, je préremplis de données.

Ensuite pour l'application Web, du fait de l'auto-incrémentation, je ferais en sorte de partir d'une table sans données.



	idperson	lastname	firstname
▶	3	JEROME	boebion
	4	DOE	John
	5	WOOD	Alex
*	NULL	NULL	NULL

Création d'un contrôleur REST pour gérer les données

2. Implémenter la communication entre l'application et la BDD

- Le principe est simple, notre code fait une requête à la base de données, et le résultat nous est retourné sous forme d'instances de l'objet Person.
- Spring Data JPA nous permet d'exécuter des requêtes SQL, sans avoir besoin de les écrire.
 - Créer une interface nommée `PersonRepository`
 - `@Repository` est une annotation Spring pour indiquer que la classe est un bean, et que son rôle est de communiquer avec une source de données (en l'occurrence la base de données).
 - Ainsi, vous pouvez utiliser les méthodes définies par l'interface `CrudRepository`
 - <https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html>

```
import fr.afpa.pompey.cda.api.model.Person;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

no usages
@Repository
public interface PersonRepository extends CrudRepository<Person, Integer> {
}
|
```

*CrudRepository est une interface fournie par Spring.
Elle fournit des méthodes pour manipuler votre entité.
Elle utilise la généricité pour que son code soit applicable à n'importe quelle entité, d'où la syntaxe `CrudRepository<Person, Integer>`*

Créer un service métier

La couche **service** est dédiée au **métier**. C'est-à-dire appliquer des traitements dictés par les règles fonctionnelles de l'application.

Chaque méthode a pour unique objectif d'appeler une méthode de l'`employeeRepository`.

`@Service` : tout comme l'annotation `@Repository`, c'est une spécialisation de `@Component`.

```
no usages
@Data
@Service
public class PersonService {

    @Autowired
    private PersonRepository personRepository;

    no usages
    public Optional<Person> getPerson(int id) {
        return personRepository.findById(id);
    }

    no usages
    public Iterable<Person> getPersons() {
        return personRepository.findAll();
    }

    no usages
    public void deletePerson(int id) {
        personRepository.deleteById(id);
    }

    no usages
    public Person savePerson(Person person) {
        Person savedPerson = personRepository.save(person);
        return savedPerson;
    }
}
```

Exposez les endpoints REST

Il faut désormais faire le mapping de notre API afin que les applications puissent communiquer avec elle.

On va suivre le modèle **REST** (par exemple pour le format des URL).

Le starter [spring-boot-starter-web](#) nous fournit justement tout le nécessaire pour créer un endpoint.

Il faut :

- une classe Java annotée [@RestController](#)
- que les méthodes de la classe soient annotées.
- Chaque méthode annotée devient alors un endpoint grâce aux annotations [@GetMapping](#), [@PostMapping](#), [@PatchMapping](#), [@PutMapping](#), [@DeleteMapping](#), [@RequestMapping](#).

Annotation	Type HTTP	Objectif
@GetMapping	GET	Pour la lecture de données.
@PostMapping	POST	Pour l'envoi de données. Cela sera utilisé par exemple pour créer un nouvel élément.
@PatchMapping	PATCH	Pour la mise à jour partielle de l'élément envoyé.
@PutMapping	PUT	Pour le remplacement complet de l'élément envoyé.
@DeleteMapping	DELETE	Pour la suppression de l'élément envoyé.
@RequestMapping		Intègre tous les types HTTP. Le type souhaité est indiqué comme attribut de l'annotation. Exemple : <code>@RequestMapping(method = RequestMethod.GET)</code>

✕ Classe Controller

```
@RestController
public class PersonController {

    @Autowired
    private PersonService personService;

    @PostMapping(⊕"/person")
    public Person createPerson(@RequestBody Person person) {
        return personService.savePerson(person);
    }

    @GetMapping(⊕"/persons")
    public Iterable<Person> getPersons() {
        return personService.getPersons();
    }

    @GetMapping(⊕"/person/{id}")
    public Person getPerson(@PathVariable("id") int id) {
        Optional<Person> person = personService.getPerson(id);
        if (person.isPresent()) {
            return person.get();
        } else {
            return null;
        }
    }
}
```

```
@PutMapping(⊕"/person/{id}")
public Person updateEmployee(@PathVariable("id") int id, @RequestBody Person person) {
    Optional<Person> e = personService.getPerson(id);
    if(e.isPresent()) {
        Person current = e.get();

        String firstName = person.getFirstname();
        if(firstName != null) {
            current.setFirstname(firstName);
        }

        String lastName = person.getLastname();
        if(lastName != null) {
            current.setLastname(lastName);
        }

        personService.savePerson(current);
        return current;
    } else {
        return null;
    }
}

@DeleteMapping(⊕"/person/{id}")
public void deletePerson(@PathVariable("id") int id) {
    personService.deletePerson(id);
}
}
```

Tester notre API

Spring Boot entre une nouvelle fois en jeu : il nous permet d'exécuter des requêtes sur notre controller.

La clé de tout cela est l'annotation `@WebMvcTest`.

- `@WebMvcTest(controllers = EmployeeController.class)` déclenche le mécanisme permettant de tester les controllers. On indique également le ou les controllers concernés.
- L'attribut `mockMvc` est un autre élément important. Il permet d'appeler la méthode "perform" qui déclenche la requête.
- L'attribut `employeeService` est annoté `@MockBean`. Il est obligatoire, car la méthode du controller exécutée par l'appel de `/employees` utilise cette classe.
- La méthode `perform` prend en paramètre l'instruction `get("/employees")`. On exécute donc une requête GET sur l'URL `/employees`.
- Ensuite, l'instruction `.andExpect(status().isOk())` indique que nous attendons une réponse HTTP 200.

```
import fr.afpa.pompey.cda.api.controller.PersonController;
import fr.afpa.pompey.cda.api.service.PersonService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

new *
@WebMvcTest(controllers = PersonController.class)
public class PersonControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private PersonService personService;

    new *
    @Test
    public void getPersonsTest() throws Exception {
        mockMvc.perform(get(uriTemplate: "/persons")).andExpect(status().isOk());
    }
}
```

Tester notre API

Le test est très similaire à notre test unitaire.

- Les annotations `@SpringBootTest` et `@AutoConfigureMockMvc` permettent de charger le contexte Spring et de réaliser des requêtes sur le controller.
- L'attribut annoté `@MockBean` a disparu, plus besoin d'un mock pour `PersonService`, car ce dernier a été injecté grâce à `@SpringBootTest`.
- En plus de vérifier si le statut vaut 200, on vérifie le contenu retourné grâce à `jsonPath("$.firstName", is("jero"))`.
 - `$` pointe sur la racine de la structure JSON.
 - `[0]` indique qu'on veut vérifier le premier élément de la liste.
 - `firstName` désigne l'attribut qu'on veut consulter.
 - `is("jero")` est ce que l'on attend comme résultat.

```
import static org.hamcrest.CoreMatchers.is;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@ neojero
@SpringBootTest
@AutoConfigureMockMvc
public class PersonControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @ neojero
    @Test
    public void getPersonsTest() throws Exception {
        mockMvc.perform(get(uriTemplate: "/persons"))
            .andExpect(status().isOk())
            .andExpect(jsonPath(expression: "$[0].firstname", is(value: "jero")));
    }
}
```

Tester notre API

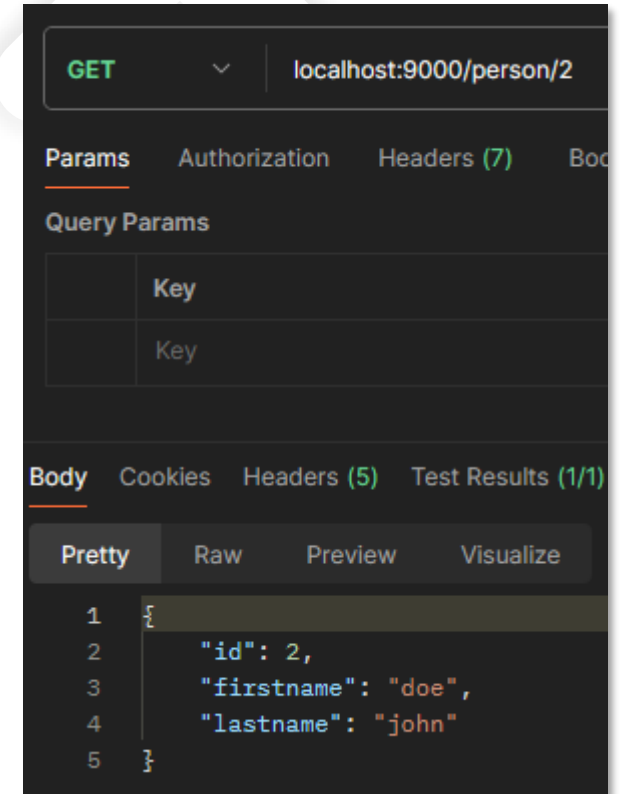
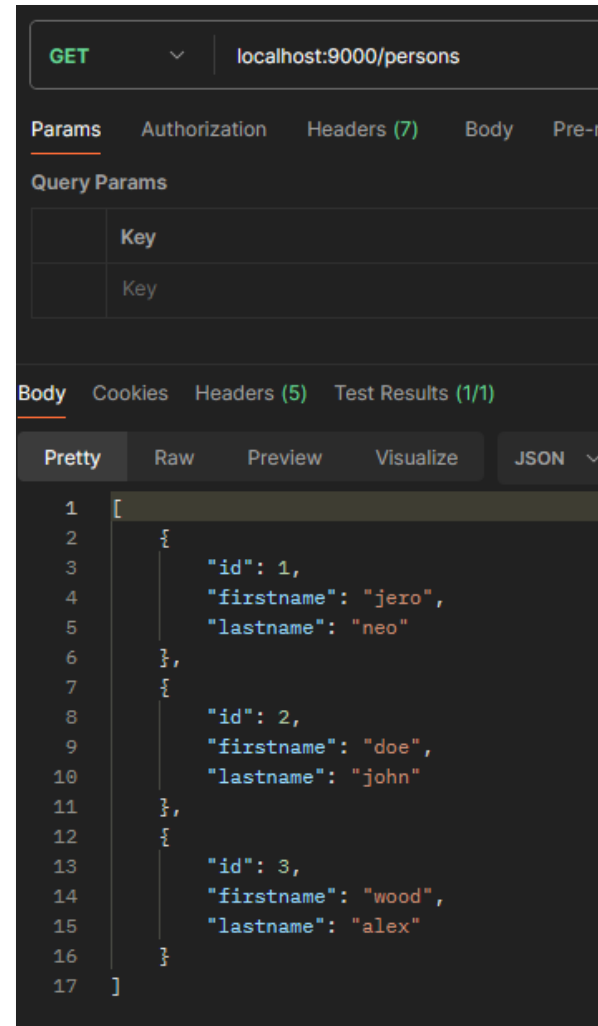
On peut également utiliser Postman pour tester notre API.

Postman est une interface graphique, utilisée par de nombreux développeurs. Elle facilite la construction de nos requêtes. C'est donc l'outil idéal pour tester des API sans devoir utiliser de code.

Après avoir build notre API depuis l'IDE, il suffit de tester nos différents mapping vers l'API.



<https://www.postman.com/>





Projet Web

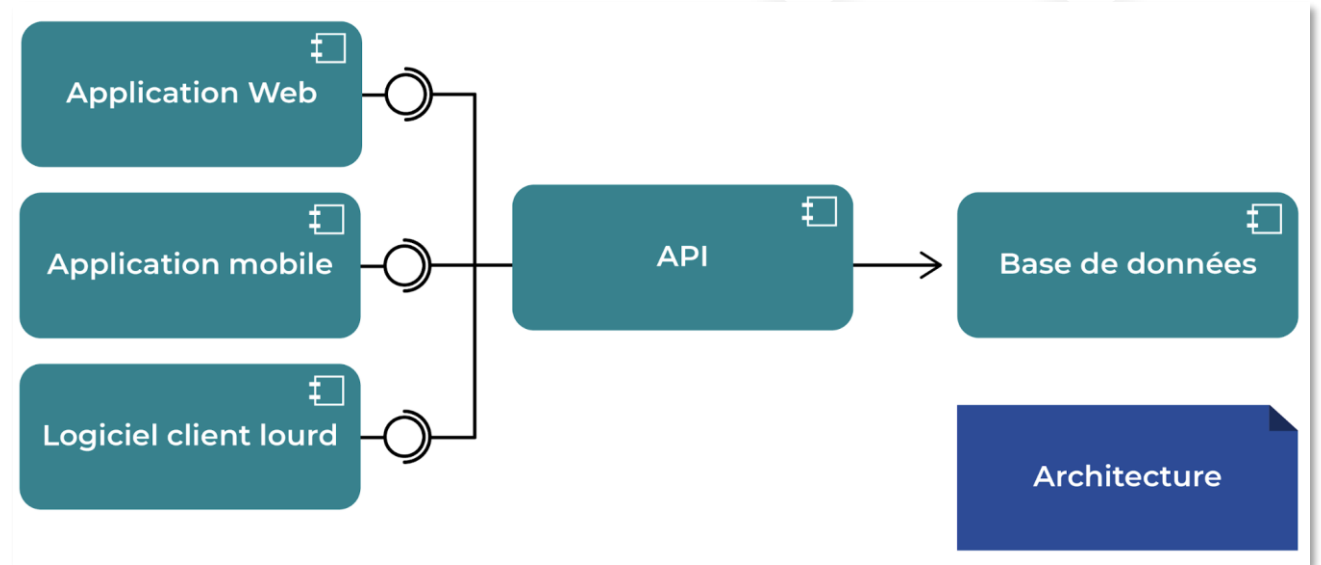
Créer un projet Web avec Spring Boot

Objectifs

Après avoir créé notre API, nous allons passer à l'étape suivante pour créer une application Web afin d'utiliser notre API.

Cette application Web permettra :

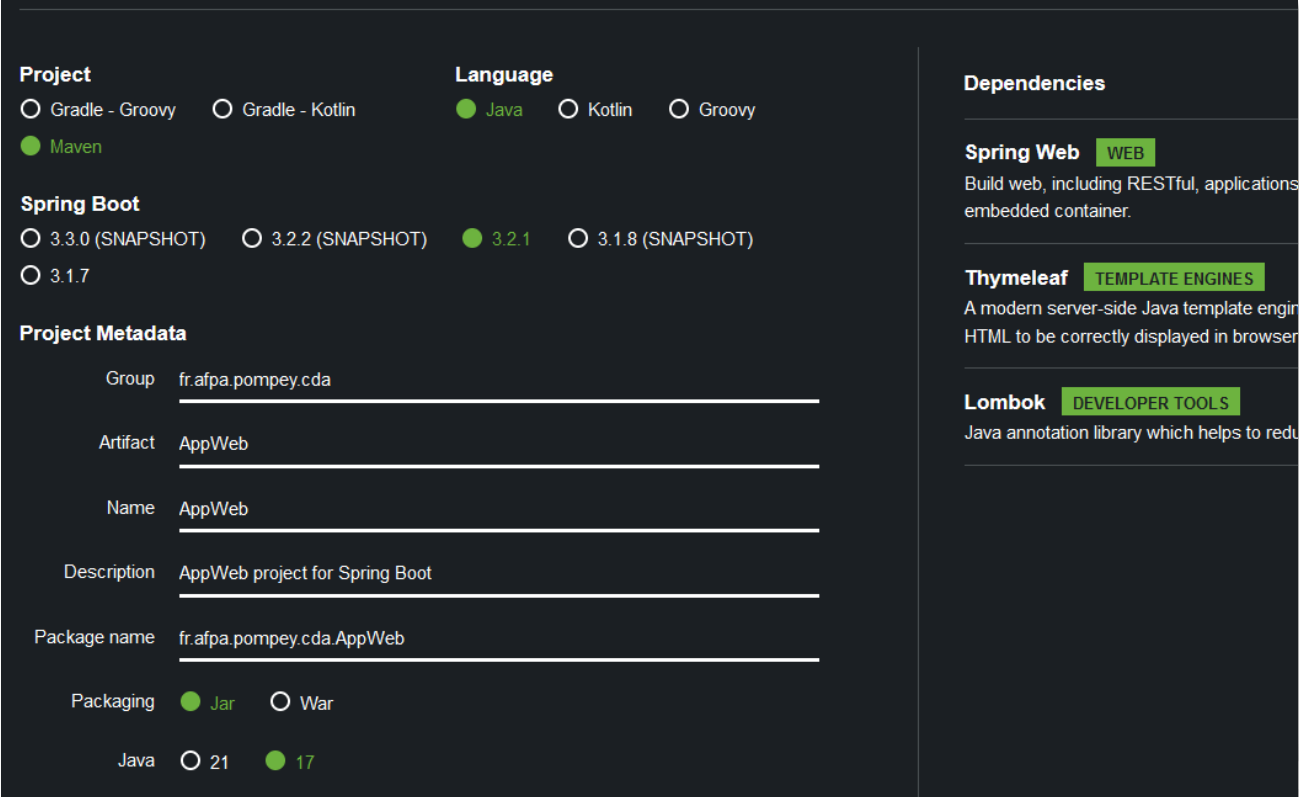
- Visualiser les personnes.
- Ajouter une nouvelle personne.
- Modifier une personne.
- Supprimer une personne.



Créer le projet

Pour les dépendances :

- **Spring Web** : il nous permettra de créer du HTML mais aussi de communiquer avec l'API
- **Lombok** : librairie pour optimiser certaines classes.
- **Thymeleaf** : c'est un moteur de template les plus couramment utilisés.



The screenshot shows the Spring Initializr web form for creating a new project. The form is divided into several sections:

- Project**: Includes radio buttons for **Gradle - Groovy**, **Gradle - Kotlin**, **Java** (selected), **Kotlin**, and **Groovy**. Below this is a radio button for **Maven** (selected).
- Spring Boot**: Includes radio buttons for **3.3.0 (SNAPSHOT)**, **3.2.2 (SNAPSHOT)**, **3.2.1** (selected), and **3.1.8 (SNAPSHOT)**. Below this is a radio button for **3.1.7**.
- Project Metadata**: Includes input fields for **Group** (fr.afpa.pompey.cda), **Artifact** (AppWeb), **Name** (AppWeb), **Description** (AppWeb project for Spring Boot), and **Package name** (fr.afpa.pompey.cda.AppWeb). Below these is a **Packaging** section with radio buttons for **Jar** (selected) and **War**. At the bottom is a **Java** section with radio buttons for **21** and **17** (selected).
- Dependencies**: A list of dependencies with checkboxes. **Spring Web** (WEB) is checked. **Thymeleaf** (TEMPLATE ENGINES) is checked. **Lombok** (DEVELOPER TOOLS) is checked.

Configurer et structurer le projet

Notre structure minimale étant prête, il nous faut désormais :

- Structurer avec des packages,
- Configurer notre application.
- <https://spring.io/guides/gs/accessing-data-mysql/>

Couche	Objectif
controller	Réceptionner la requête et fournir la réponse
service	Exécuter les traitements métiers
repository	Communiquer avec la source de données
model	Contenir les objets métiers

```
spring.application.name=AppWebbApplication

# port par défaut de tomcat
server.port=9001

# niveau de level logging
logging.level.root=error
# pour mon package
logging.level.fr.afpa.pompey.cda=info
# pour la BDD
logging.level.org.springframework.data=INFO
logging.level.org.springframework.jdbc.core.JdbcTemplate=DEBUG
# niveau de level logging pour TOMCAT
logging.level.org.springframework.boot.web.embedded.tomcat=INFO
```

la gestion des propriétés

Jusqu'à présent, nous avons saisi des valeurs pour des propriétés existantes. Propriétés utiles à des classes de Spring qu'on ne manipule pas.

Cependant, si je souhaite ajouter une nouvelle propriété, comment y accéder dans mon code ?

1. Créer une propriété custom dans le `application.properties`
2. Créer le bean associé
3. Utiliser les propriétés dans le code.

```
#Custom config
fr.afpa.pompey.cda.appweb.apiUrl=http://localhost:9000
```

```
package config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
@ConfigurationProperties(prefix = "fr.afpa.pompey.cda.appweb")
@Data
public class CustomProperties {

    private String apiUrl;

}
```

```
@Data
@EnableConfigurationProperties(CustomProperties.class)
@SpringBootApplication
public class AppWebApplication implements CommandLineRunner {

    @Autowired
    private CustomProperties props;

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

    @Override
    public void run(String... args) throws Exception {
        System.out.println(props.getApiUrl());
    }
}
```

Modéliser l'objet Person

Ici, rien de spécial, nous créons le modèle en rapport aux données que nous allons récupérer de l'API.

L'annotation `@Data` permet de générer automatiquement les getters et setters pour chaque attribut.

```
package model;  
  
import lombok.Data;  
  
no usages  
@Data  
public class Person {  
  
    private Integer id;  
    private String firstname;  
    private String lastname;  
}
```

La communication entre l'application web et l'API REST

1. Tout d'abord, nous allons créer un service métier avec un double objectif :
 1. Exécuter les traitements métiers,
 2. Faire les tests vers la couche repository.
2. Ensuite, nous allons créer le composant Repository métier.

Le starter Spring Web nous fournit tout le code nécessaire pour cela.

Nous allons nous servir de la classe [RestTemplate](#) pour la couche Repository.

Note : L'utilisation de l'objet RestTemplate va progressivement être remplacée par l'utilisation des classes du module Spring Webflux. RestTemplate reste une façon valide de communiquer avec une API en synchrone pour débiter

Couche	Objectif
controller	Réceptionner la requête et fournir la réponse
service	Exécuter les traitements métiers
repository	Communiquer avec la source de données
model	Contenir les objets métiers

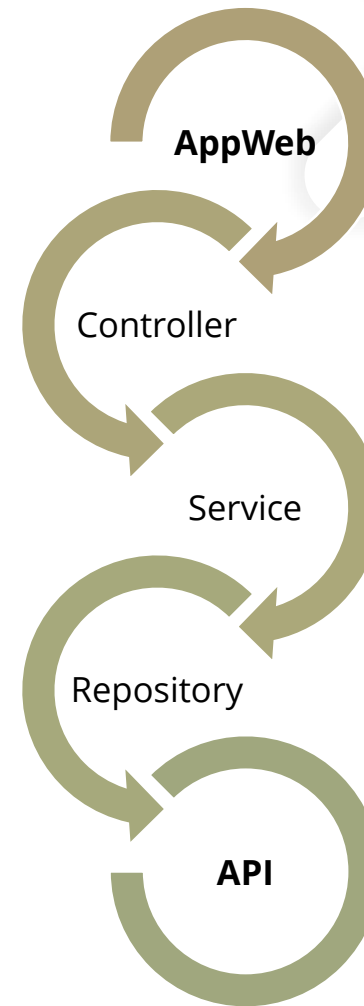
Interaction entre les couches

Même principe que pour l'API, au lancement de l'application web, le contrôleur est chargé.

En fonction du mapping mis en place, il fait appel au Service.

Le service se charge alors d'orienter la demande vers le repository en fonction de la demande faite à l'API.

Enfin le repository contacte l'API en lui donnant l'url adéquate pour récupérer la ou les données.





RestTemplate

La communication entre l'application web et l'API REST : couche [Repository](#)

Couche Repository : RestTemplate

[RestTemplate](#) permet d'exécuter une requête HTTP. En fournissant l'URL, le type de requête (GET, POST, etc.), et le type d'objet qui sera retourné. [RestTemplate](#) fait la requête à l'API mais en plus convertit le résultat JSON en objet Java

Grâce à notre objet [CustomProperties](#), on récupère l'URL de l'API.

- on instancie notre objet [RestTemplate](#).
- on appelle la méthode `exchange` en transmettant :
 - l'URL
 - la méthode HTTP
 - Null en lieu et place d'un objet [HttpEntity](#), ainsi on laisse un comportement par défaut.
 - le type retour, ici un objet [ParameterizedTypeReference](#) car `/persons` renvoie un objet `Iterable<Person>`. Mais si l'endpoint renvoie un objet simple, alors il suffira d'indiquer `<Object>.class`.
- on récupère notre objet `Iterable<Person>` grâce à la méthode `getBody()` de l'objet [Response](#).

```
/**
 * Get all Person
 * @return An iterable of all Persons
 */
1 usage
public Iterable<Person> getPersons() {

    String baseApiUrl = props.getApiUrl();
    String getPersonUrl = baseApiUrl + "/persons";

    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<Iterable<Person>> response = restTemplate.exchange(
        getPersonUrl,
        HttpMethod.GET,
        requestEntity: null,
        new ParameterizedTypeReference<>() {}
    );

    log.debug("Get Persons call " + response.getStatusCode());

    return response.getBody();
}
```


Couche Repository : RestTemplate

Si on souhaite créer une nouvelle personne, il nous faut envoyer des données à l'API.

Le code sera très similaire.

On retrouve notre objet RestTemplate et on fournit toujours l'URL, la méthode HTTP (cette fois POST) et le type retour (en l'occurrence Person.class).

La grande différence correspond au nouvel objet `HttpEntity` qui, comme vous le constatez, a été instancié avec en paramètre du constructeur l'objet Person (nommé e).

Je transmets ensuite cet objet `HttpEntity` comme 3e paramètres de la méthode `exchange` (ce paramètre que nous avons mis à null précédemment).

```
/**
 * Add a new Person
 * @param e A new Person (without an id)
 * @return The Person full filled (with an id)
 */
1 usage
public Person createPerson(Person e) {
    String baseApiUrl = props.getApiUrl();
    String createPersonsUrl = baseApiUrl + "/person";

    RestTemplate restTemplate = new RestTemplate();
    HttpEntity<Person> request = new HttpEntity<>(e);
    ResponseEntity<Person> response = restTemplate.exchange(
        createPersonsUrl,
        HttpMethod.POST,
        request,
        Person.class);

    log.debug("Create Person call " + response.getStatusCode());

    return response.getBody();
}
```

Couche Service

La couche service a pour but un double objectif :

1. Exécuter les traitements métiers.
2. Faire le relais vers la couche repository.

Nous allons créer une classe nommée `PersonService` dans le package associé.

Bien évidemment, cette classe devra être identifiée comme étant un bean.

L'annotation utilisée sera `@Service`, spécialisation de l'annotation `@Component`.

C'est dans cette classe que nous pouvons mettre en place la gestion de règles de gestion dictée par les besoins fonctionnel.

Par exemple, formaliser les données avant de les envoyer dans la base de données.

```
2 usages
@Data
@Service
public class PersonService {

    @Autowired
    private PersonRepository personRepository;

    2 usages
    public Person getPerson(int id) {
        return personRepository.getPerson(id);
    }

    1 usage
    public Iterable<Person> getPersons() {
        return personRepository.getPersons();
    }

    1 usage
    public void deletePerson(final int id) {
        personRepository.deletePerson(id);
    }
}
```

code source

Classe PersonService.java et
PersonRepository.java

La communication entre l'application web et l'API REST : la classe de service

```
package service;

import lombok.Data;
import model.Person;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import repository.PersonRepository;

no usages
@Data
@Service
public class PersonService {

    @Autowired
    private PersonRepository personRepository;

    no usages
    public Person getPerson(int id) {
        return personRepository.getPerson(id);
    }

    no usages
    public Iterable<Person> getPersons() {
        return personRepository.getPersons();
    }
}
```

```
no usages
public void deletePerson(final int id) {
    personRepository.deletePerson(id);
}

no usages
public Person savePerson(Person person) {
    Person saved;

    // Règle de gestion : Le nom de famille doit être mis en majuscule.
    person.setLastname(person.getLastname().toUpperCase());

    if(person.getId() == null) {
        // Si l'id est nul, alors c'est un nouvel employé.
        saved = personRepository.createPerson(person);
    } else {
        saved = personRepository.updatePerson(person);
    }

    return saved;
}
```


La communication entre l'application web et l'API REST : la classe Repository

```
package repository;

import config.CustomProperties;
import lombok.extern.slf4j.Slf4j;
import model.Person;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

2 usages
@Slf4j
@Component
public class PersonRepository {

    @Autowired
    private CustomProperties props;
```

```
/**
 * Get all Person
 * @return An iterable of all Persons
 */
1 usage
public Iterable<Person> getPersons() {

    String baseApiUrl = props.getApiUrl();
    String getPersonUrl = baseApiUrl + "/persons";

    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<Iterable<Person>> response = restTemplate.exchange(
        getPersonUrl,
        HttpMethod.GET,
        requestEntity: null,
        new ParameterizedTypeReference<>() {}
    );

    log.debug("Get Persons call " + response.getStatusCode());

    return response.getBody();
}
```

La communication entre l'application web et l'API REST : la classe Repository

```
/**
 * Get an employee by the id
 * @param id The id of the Person
 * @return The Person which matches the id
 */
1 usage
public Person getPerson(int id) {
    String baseApiUrl = props.getApiUrl();
    String getPersonUrl = baseApiUrl + "/person/" + id;

    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<Person> response = restTemplate.exchange(
        getPersonUrl,
        HttpMethod.GET,
        requestEntity: null,
        Person.class
    );

    log.debug("Get Person call " + response.getStatusCode());

    return response.getBody();
}
```

```
/**
 * Add a new Person
 * @param e A new Person (without an id)
 * @return The Person full filled (with an id)
 */
1 usage
public Person createPerson(Person e) {
    String baseApiUrl = props.getApiUrl();
    String createPersonsUrl = baseApiUrl + "/person";

    RestTemplate restTemplate = new RestTemplate();
    HttpEntity<Person> request = new HttpEntity<>(e);
    ResponseEntity<Person> response = restTemplate.exchange(
        createPersonsUrl,
        HttpMethod.POST,
        request,
        Person.class);

    log.debug("Create Person call " + response.getStatusCode());

    return response.getBody();
}
```

La communication entre l'application web et l'API REST : la classe Repository

```
/**
 * Update an Persons - using the PUT HTTP Method.
 * @param e Existing Persons to update
 */
1 usage
public Person updatePerson(Person e) {
    String baseApiUrl = props.getApiUrl();
    String updatePersonUrl = baseApiUrl + "/person/" + e.getId();

    RestTemplate restTemplate = new RestTemplate();
    HttpEntity<Person> request = new HttpEntity<>(e);
    ResponseEntity<Person> response = restTemplate.exchange(
        updatePersonUrl,
        HttpMethod.PUT,
        request,
        Person.class);

    log.debug("Update Person call " + response.getStatusCode());

    return response.getBody();
}
```

```
/**
 * Delete an Person using exchange method of RestTemplate
 * instead of delete method in order to log the response status code.
 * @param id The Person to delete
 */
1 usage
public void deletePerson(int id) {
    String baseApiUrl = props.getApiUrl();
    String deletePersonUrl = baseApiUrl + "/person/" + id;

    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<Void> response = restTemplate.exchange(
        deletePersonUrl,
        HttpMethod.DELETE,
        requestEntity: null,
        Void.class);

    log.debug("Delete Person call " + response.getStatusCode());
}
```

Renvoyez les pages HTML

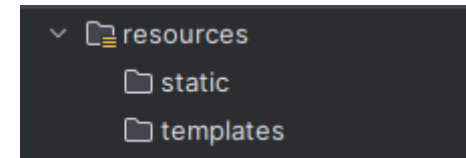
Pour suivre l'approche MVC, nous avons besoin de nous pencher sur la séparation entre le code "contrôleur" (traiter les requêtes entrantes) et le code "vue" (construction du résultat visuel à fournir à l'utilisateur).

Lors du choix des starters, nous avons bien évidemment sélectionné Spring Web, mais également le moteur de template Thymeleaf.

Il nous faut apprendre à :

- Utiliser Spring Web pour traiter une requête qui nous parvient.
- Utiliser Spring Web pour renvoyer une réponse HTML.
- Formater notre réponse HTML avec Thymeleaf.

- + Selon l'url saisie par l'utilisateur de l'application web, le contrôleur va jouer le rôle de redirection vers la bonne page.
- + Les méthodes du contrôleur devront :
 - Récupérer les données en entrée (s'il y en a).
 - Faire appel aux traitements métiers (en l'occurrence, communiquer avec la couche service).
 - Retourner une vue HTML.
- + Nos templates html seront placés dans le répertoire appropriés.



Le contrôleur

Notre contrôleur `PersonController` va contenir l'annotation `@Controller` pour qu'elle soit détectée comme un bean.

Le concept est le suivant :

1. L'annotation `@GetMapping` spécifie le type de requête HTTP et l'URL correspondante.
2. Le texte "home" retourné correspond au nom du fichier HTML.

À l'appel de l'URL racine via le type GET de mon application web, la méthode `home()` sera **automatiquement exécutée**, et Spring **renverra automatiquement une réponse** HTTP contenant dans son corps (donc le body HTTP) le contenu du fichier `home.html`.

```
@GetMapping("/")
public String home(Model model) {
    Iterable<Employee> listEmployee = service.getEmployees();
    model.addAttribute("employees", listEmployee);

    return "home";
}
```

Fournir des données à la vue

Spring met à disposition différentes classes pour cela.

L'objet Model [org.springframework.ui.Model](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.ui.Model.html) a été ajouté en paramètre de la méthode `index()`.

Grâce à cela, Spring se charge de nous fournir une instance de cet objet.

Puis, dans le corps de la méthode, j'utilise une méthode `addAttribute` qui permet d'ajouter à mon Model un objet.

Le premier paramètre spécifie le **nom** et le deuxième **l'objet**.

Ensuite au sein de la vue, nous utiliserons [Thymeleaf](https://www.thymeleaf.org/) pour afficher les données.

```
@Data
@Controller
public class PersonController {

    @Autowired
    private PersonService service;

    // neojero *
    @GetMapping("/")
    public String index(Model model) {
        Iterable<Person> listPersons = service.getPersons();
        model.addAttribute( attributeName: "persons", listPersons);
        return "index";
    }
}
```

Création des vues

Nos pages HTML





<https://www.thymeleaf.org/>

Thymeleaf

Moteur de template



Introduction

Thymeleaf est un template engine (moteur de rendu de document) écrit en Java.

Principalement conçu pour produire des vues Web, il fournit un support pour la génération de documents HTML, XHTML, JavaScript et CSS (voire de n'importe quel format texte).

<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>

La dépendance ajoutée à la création du projet :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

- + Il existe plusieurs **template engines** en Java :
 - JSP (directement inclus dans le moteur de Servlet des serveurs Web Java),
 - FreeMarker, Velocity...
- + **Il n'est donc pas facile de savoir quelle solution choisir.**
- + **Thymeleaf** a cependant plusieurs avantages par rapport aux autres template engines
 - créé des templates qui respectent le format du document. Si vous écrivez un template pour une page HTML, le template sera une page HTML valide.
 - fournit des valeurs par défaut dans un template. On peut ainsi concevoir des templates pour un site Web avec des données d'exemple et le site reste navigable hors ligne comme un simple site statique. **Utile pour réaliser un maquettage de votre site.**
 - s'intègre parfaitement dans une application Spring puisque vous pourrez utiliser le langage d'expression de Spring (**SpEL**) pour dynamiser vos templates.
 - Enfin facile à apprendre.

Les attributs Thymeleaf

Un **template Thymeleaf** pour une page HTML5 est une page HTML utilisant des attributs Thymeleaf qui seront interprétés par le serveur pour produire la page finale.

Afin de respecter le format HTML5, les attributs thymeleaf sont déclarés comme des attributs **data-th-***.

Ces attributs ne seront pas présents dans la page finale mais permettent de garantir que le template est lui-même une page HTML5 valide.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Une page</title>
  </head>
  <body>
    <div data-th-text="${ user.login }">David</div>
  </body>
</html>
```

Dans l'exemple ci-dessus, on utilise l'attribut data-th-text pour indiquer le contenu texte de la balise <div>. Le texte David sera remplacé à l'exécution par le résultat de l'expression `${ user.login }`.

Les attributs Thymeleaf

Le contenu des attributs Thymeleaf peut être du simple texte ou une expression qui sera interprétée à l'exécution.

Les expressions sont délimitées par :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Une page</title>
  </head>
  <body>
    <div data-th-object="${ utilisateur }">
      <p>Prénom : <span data-th-text="*{ prenom }">Jean</span></p>
      <p>Nom : <span data-th-text="*{ nom }">Dubois</span></p>
      <p>Age : <span data-th-text="*{ age }">35</span> ans</p>
    </div>
  </body>
</html>
```

+ \${...}

- Pour les expressions à évaluer. cette expression est écrite avec SpEL et peut référencer n'importe quel objet présent dans le modèle.
- les objets `param`, `session` et `application` sont disponibles pour accéder aux paramètres de la requête, aux objets présents dans la session utilisateur et aux objets présents dans la portée Java EE de l'application
- Il est également possible de référencer n'importe quel bean du contexte d'application en précisant `@` devant son nom.
- Il existe également des objets prédéfinis qui sont accessibles en préfixant leur nom avec `#`
- On trouve également des objets utilitaires pour nous aider à manipuler ou à formater des données (date, URI, nombres, listes...) `#strings` par exemple.

+ *{...}

- Pour les expressions de sélection à évaluer. Ce type d'expression est très utile lorsqu'on veut construire des formulaires ou afficher les propriétés d'un objet du modèle

Les attributs Thymeleaf

Le contenu des attributs Thymeleaf peut être du simple texte ou une expression qui sera interprétée à l'exécution.

```
<link rel="stylesheet" href="../../static/style.css" data-th-href="@{/style.css}">
<script src="../../static/app.js" data-th-src="@{/app.js}"></script>
```

+ #{...}

- Pour les expressions de message. Une expression délimitée par #{...} représente un message à extraire d'un fichier de ressources.
- Dans une application Spring Boot, vous pouvez externaliser vos messages dans le fichier src/main/resources/messages.properties.

+ @{...}

- Pour les expressions d'URL qui doit être résolue par rapport au contexte de génération de la page. Tous les liens internes à votre application référençant vos pages, vos feuilles de style, vos fichiers de script, ... doivent utiliser des attributs Thymeleaf pour positionner le chemin. L'attribut HTML correspondant peut être conservé pour permettre d'ouvrir la page localement.

+ ~{...} Pour les expressions de fragment

Prévention de l'injection

Thymeleaf est configuré par type de fichier pour prévenir l'injection de données.

Pour la génération de page HTML, Thymeleaf fera un échappement HTML pour les caractères réservés comme `<` ou `>`.

Ainsi, les données dynamiques dans la page ne modifient pas la structure du template et l'injection HTML n'est pas possible.

Ainsi le template :

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>Une page</title>
6   </head>
7   <body>
8     <div data-th-text="${ ' <strong>Bonjour</strong>' }"></div>
9   </body>
10 </html>
```

produira la page HTML finale :

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="UTF-8">
5     <title>Une page</title>
6   </head>
7   <body>
8     <div>&lt;strong&gt;Bonjour&lt;/strong&lt;/div>
9   </body>
10 </html>
```

Les conditions et les boucles

Dans un template, il est très souvent nécessaire de produire une partie d'un document à certaines conditions ou de répéter certaines portions (par exemple pour générer un tableau).

Le langage [SpEL](#) ne permet pas d'exprimer des structures aussi complexes.

Par contre, il existe les attributs Thymeleaf [if](#) et [each](#) qui sont fait pour cela.

- + L'attribut `if` permet de contrôler la production de la balise et de tout ce qu'elle contient suivant l'expression d'une condition.

```
<div data-th-if="${ commande.remise > 0 }">
    Vous pouvez disposez d'une remise de
    <span data-th-text="${ commande.remise }"></span>%
</div>
```

- + L'attribut `each` permet de parcourir un tableau ou une liste ou n'importe quelle structure itérable et de répéter la balise et tout son contenu.

```
<table>
  <thead>
    <tr>
      <th>Nom</th>
      <th>Age</th>
    </tr>
  </thead>
  <tbody>
    <tr data-th-each="p : ${ personnes }">
      <td data-th-text="${ p.nom }">Jean Dubois</td>
      <td data-th-text="${ p.age }">35</td>
    </tr>
  </tbody>
</table>
```

Thymeleaf

Opérateur de sûreté

- + Si on veut afficher la valeur de l'attribut nom d'un objet du modèle. On peut écrire :
 - o `le nom ici`
- + *Mais que se passe-t-il si l'objet n'est pas présent dans le modèle ? La génération de la page va échouer avec une erreur car l'objet vaut null.*
- + SpEL définit un opérateur de sûreté ([safe navigation operator](#)) : ?
- + Si nous modifions l'expression :
 - o `le nom ici`
 - o La génération de la page n'échouera plus si l'objet n'est pas disponible.
 - o Le résultat de l'expression `objet?.nom` sera simplement une chaîne vide.

Le formatage des données

- + Les objets utilitaires disponibles dans les expressions peuvent nous permettre de formater les données à afficher.
 - o `${ #dates.format(maintenant, 'dd MMMM YYYY') }`
- + Une application Spring peut enregistrer des formateurs de données à son lancement. Pour cela, il faut ajouter un bean de configuration dans le contexte de l'application.
- + Ce bean doit implémenter l'interface `WebMvcConfigurer` et implémenter la méthode `addFormatters`.

```
@Configuration
public class PizzaSpringMvcConfigurer implements WebMvcConfigurer {
    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatter(new NumberStyleFormatter("#,##0.##"));
    }
}
```

Récupérer des données de la vue

Situation n° 1 : la donnée est transmise par URL.

La méthode `deletePerson` possède un paramètre nommé `id` de type `int`.

Cependant le point clé est l'annotation `@PathVariable` qui signifie que ce paramètre est présent dans l'URL de requête

Je peux ensuite me servir de la variable `id` dans le code `service.Person(id);`

Côté Thymeleaf, voici le code correspondant :

```
<a th:href="@{/deletePerson/{id}(id=${person.id})}"><button class="btn btn-danger">Supprimer</button></a>
```

```
@GetMapping("/deletePerson/{id}")
public ModelAndView deletePerson(@PathVariable("id") final int id) {
    service.deletePerson(id);
    return new ModelAndView("redirect:/");
}
```

Récupérer des données de la vue

Situation n° 2 : la donnée est transmise par un formulaire.

La méthode `savePerson` est annotée `@PostMapping` car généralement les formulaires exécutent des requêtes `POST`.

L'autre point clé est le paramètre de la méthode `@ModelAttribute` l'annotation magique.

Cette annotation permet à Spring de récupérer les données saisies dans les champs du formulaire et de construire un objet `Person`.

Côté Thymeleaf, le code sera le suivant

```
neojero
@PostMapping("/savePerson")
public ModelAndView savePerson(@ModelAttribute Person person) {
    Person current = null;
    if(person.getId() != null) {
        current = service.getPerson(person.getId());
    }
    service.savePerson(current);
    return new ModelAndView("redirect:/");
}
```

```
<form method="post"
    th:action="@{/savePerson}" th:object="${person}">
    <div class="form-group">
        <label for="firstNameInput">Prénom</label>
        <input type="text" th:field="*{firstName}" class="form-control" id="firstNameInput"
            aria-describedby="firstNameHelp" placeholder="Saisir le prénom">
        <small id="firstNameHelp" class="form-text text-muted">Merci d'écire le prénom de la personne.</small>
    </div>
    <div class="form-group">
        <label for="lastNameInput">Prénom</label>
        <input type="text" th:field="*{lastName}" class="form-control" id="lastNameInput"
            aria-describedby="lastNameHelp" placeholder="Saisir le prénom">
        <small id="lastNameHelp" class="form-text text-muted">Merci d'écire le nom de la personne.</small>
    </div>

    <button type="submit" class="btn btn-primary">Créer</button>
</form>
```

Code des différentes vues

Template thymeleaf

Index.html

Note : pour la réalisation de ces pages, utilisation de Bootstrap.

Cette page affiche la liste des personnes et propose d'en créer une nouvelle ou de modifier une personne.

ATTENTION : En début de chaque page, il faut penser à ajouter la déclaration du raccourci `th` pour Thymeleaf

```
<!DOCTYPE html>
<html lang="fr" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Person Web Application</title>
  <link rel="stylesheet" type="text/css" th:href="@{/css/style.css}">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet"
    integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCm61MXxSR1GAsXEVDwWykc2MPK8M2HN" crossorigin="anonymous">
</head>
<body>
```

```
<div class="container special">
  <h2 class="h2">Liste des personnes</h2>
  <div class="table-responsive">
    <table class="table table-striped table-sm">
      <thead>
        <tr>
          <th>Prénom</th>
          <th>Nom</th>
        </tr>
      </thead>
      <tbody>
        <tr th:if="${#lists.isEmpty(listPersons)}">
          <td colspan="3">Aucune personnes en base de données</td>
        </tr>
        <tr th:each="person: ${listPersons}">
          <td><span th:text="${person.firstname}"> Prénom </span></td>
          <td><span th:text="${person.lastname}"> Nom </span></td>
          <td>
            <a th:href="@{/updatePerson/{id}(id=${person.id})}"><button class="btn btn-info">Modifier</button></a>
            <a th:href="@{/deletePerson/{id}(id=${person.id})}"><button class="btn btn-danger">Supprimer</button></a>
          </td>
        </tr>
      </tbody>
    </table>
  </div>
  <div>
    <h4><a th:href="@{/createPerson}">Ajouter une nouvelle personne</a></h4>
  </div>
```

Formulaire de création

Ici, vous trouverez le code du formulaire pour la création d'une personne.

```
<div class="container special">

  <h2 class="h2">Ajout d'une nouvelle personne</h2>
  <div>
    <form method="post"
      th:action="@{/savePerson}" th:object="${person}">
      <div class="form-group">
        <label for="firstNameInput">Prénom</label>
        <input type="text" th:field="*{firstName}" class="form-control" id="firstNameInput"
          aria-describedby="firstNameHelp" placeholder="Saisir le prénom">
        <small id="firstNameHelp" class="form-text text-muted">Merci d'écrire le prénom de la personne.</small>
      </div>
      <div class="form-group">
        <label for="lastNameInput">Prénom</label>
        <input type="text" th:field="*{lastName}" class="form-control" id="lastNameInput"
          aria-describedby="lastNameHelp" placeholder="Saisir le prénom">
        <small id="lastNameHelp" class="form-text text-muted">Merci d'écrire le nom de la personne.</small>
      </div>

      <button type="submit" class="btn btn-primary">Créer</button>
    </form>
  </div>
</div>
```


Formulaire de mise à jour

Ici, vous trouverez le code du formulaire pour la mise à jour d'une personne.

```
<div class="container special">
  <h2 class="h2">Modification d'une personne</h2>
  <div>
    <form method="post"
      th:action="@{/savePerson}" th:object="${person}">
      <div class="form-group">
        <input type="hidden" th:field="*{id}" class="form-control">
      </div>
      <div class="form-group">
        <label for="firstNameInput">Prénom</label>
        <input type="text" th:field="*{firstname}" class="form-control" id="firstNameInput"
          aria-describedby="firstNameHelp" placeholder="Saisir le prénom">
        <small id="firstNameHelp" class="form-text text-muted">Merci d'écrire le prénom.</small>
      </div>
      <div class="form-group">
        <label for="lastNameInput">Nom</label>
        <input type="text" th:field="*{lastname}" class="form-control" id="lastNameInput"
          aria-describedby="lastNameHelp" placeholder="Saisir le nom">
        <small id="lastNameHelp" class="form-text text-muted">Merci d'écrire le nom.</small>
      </div>
      <button type="submit" class="btn btn-primary">Modifier</button>
    </form>
  </div>
</div>
```

Page d'erreurs

Cette page permet d'afficher les erreurs.

Pour pouvoir afficher les erreurs de Spring Boot, il faut positionner dans le fichier `application.properties`, ces deux lignes de configuration :

```
# port par d?faut de tomcat
server.port=9001
#affichage d'une page d'erreurs
server.error.include-exception=true
server.error.include-stacktrace=always
```

```
<h1>Page erreurs</h1>

<section class="container">
  <table class="table">
    <tr>
      <td>Error Message</td>
      <td th:text="${message}"></td>
    </tr>
    <tr>
      <td>Status Code</td>
      <td th:text="${status}"></td>
    </tr>
    <tr>
      <td>Exception</td>
      <td th:text="${exception}"></td>
    </tr>
    <tr>
      <td>Stacktrace</td>
      <td><pre th:text="${trace}"></pre></td>
    </tr>
    <tr>
      <td>Binding Errors</td>
      <td th:text="${errors}"></td>
    </tr>
  </table>
</section>
```



Tester et déployer le projet

Dernière étape

Test d'intégration

- L'annotation `@SpringBootTest` permet au contexte Spring d'être chargé lors de l'exécution des tests.
- Pour exécuter une requête au contrôleur, on utilise un objet `MockMvc` annoté `@Autowired` pour l'injection de la dépendance.
- Et la classe doit obligatoirement être annotée `@AutoConfigureMockMvc`, afin qu'un objet `MockMvc` soit disponible dans le contexte Spring (ainsi il pourra être injecté dans l'attribut).
- La méthode `perform` a en paramètre l'URL à appeler. Puis il s'ensuit une suite d'instructions pour vérifier l'attendu :
 - `status().isOk()` : la réponse a un code de statut 200
 - `view().name("home")` : le nom de vue retourné correspond au paramètre "home"
 - `content().string(containsString("jero"))` : le corps de la réponse contient à un moment ou à un autre le texte jero.
- Notons également que `andDo(print())` permet au retour de l'appel d'être affiché dans la console (on y verra donc tout le HTML généré).

```
@SpringBootTest
@AutoConfigureMockMvc
public class PersonControllerTest {

    @Autowired
    private MockMvc mockMvc;

    new *
    @Test
    public void getPersonsTest() throws Exception {

        mockMvc.perform(get(urlTemplate: "/"))
            // affiche le résultat dans la console
            // avec status 200
            // nom de la vue = index
            // une des personnes de la liste est égal à john
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(view().name(expectedViewName: "home"))
            .andExpect(MockMvcResultMatchers.content().string(containsString(substring: "john")));
    }
}
```

Déploiement

Pour lancer le projet, vous pouvez évidemment utiliser l'IDE pour lancer les deux projets et vérifier l'interaction avec l'API et votre application WEB.

Vous pouvez également générer le .jar du projet avec Maven et lancer chaque jar individuellement en ligne de commande.

1. Il faut vous positionner dans le dossier projet et lancer le jar avec la commande java
2. Tester depuis un navigateur sur le localhost:portdeTomcat
3. Si tout se déroule normalement, vous pouvez dès lors tester les deux projets.

Nom	Modifié le	Type	Taille
.git	19/01/2024 10:00	Dossier de fichiers	
.idea	23/01/2024 14:07	Dossier de fichiers	
.mvn	17/01/2024 12:45	Dossier de fichiers	
src	17/01/2024 12:45	Dossier de fichiers	
target	23/01/2024 11:43	Dossier de fichiers	
.gitignore			
HELP.md			
mvnw			
mvnw.cmd			
pom.xml			

```

MINGW64:/d/PROJETS/JAVA/SPRINGBOOT/api
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/JAVA/S
$ java -jar target/api-0.0.1-SNAPSHOT.jar
  
```

Nom	Modifié le	Type	Taille
.git	23/01/2024 13:13	Dossier de fichiers	
.idea	23/01/2024 14:07	Dossier de fichiers	
.mvn	18/01/2024 08:05	Dossier de fichiers	
src	18/01/2024 08:05	Dossier de fichiers	
target	23/01/2024 11:46	Dossier de fichiers	
.gitignore			
HELP.md			
mvnw			
mvnw.cmd			
pom.xml			

```

MINGW64:/d/PROJETS/JAVA/SPRINGBOOT/appweb
jboeb@LAPTOP-RVGOQTRQ MINGW64 /d/PROJETS/JAVA/S
$ java -jar target/appweb-0.0.1-SNAPSHOT.jar
  
```




Conclusion

Bravo, vous êtes à la fin !!

Conclusion

- + Ce petit aperçu Spring Boot vous permet de voir toutes les possibilités que peut offrir un framework au quotidien.
- + Evidemment, cela demande d'approfondir l'apprentissage du framework afin d'en connaître toutes les possibilités.
- + Notamment en termes de sécurité avec [Spring Security](#)
- + Note :
 - [IDE IntelliJ](#)
 - [JDK version 17](#)
 - [Spring Boot version 3.2.1](#)
 - [MySQL](#)
 - [Thymeleaf](#)



Merci d'avoir parcouru ce tutoriel.

FIN