

Spring et SpringBoot



P.Mathieu

IUT de Lille

<http://www.iut-a.univ-lille.fr>

prenom.nom@univ-lille.fr

Le Framework Spring

Spring-Boot

Spring Data JDBC

Spring Data JPA

Le Framework Spring

Des frameworks, en veux-tu ? en voilà !

Java : Spring, Quarkus, Struts, JSF, GWT, Wicket, ...

PHP : Symphony, Laravel, CakePHP, Zend, Phalcon, ...

Python : Django, Flask, Bottle, CherryPy, Pyramid, ...

Javascript : Express, Next.js, Nuxt, Meteor, Koa, ...

...

Intérêt

Fournir un guide de conception, Automatiser les tâches répétitives
réduire le “boilerplate code”

Principe

- ▶ Framework libre permettant de construire des applications Java
- ▶ Considéré comme un conteneur léger
(les liens inter-classes sont faits à l'exécution, pas à la compilation)
- ▶ Structure modulaire (avec de nombreux modules > 20) s'appuyant sur 3 concepts clés :
 - 1 l'inversion de contrôle
 - 2 La programmation orientée Aspects (AOP)
 - 3 Une couche d'abstraction

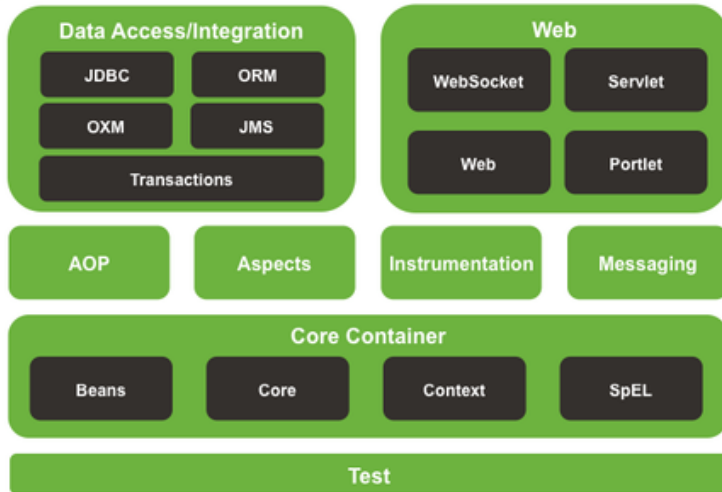
C'est un framework général
On peut faire du Java "traditionnel" avec Spring !

Le Framework Spring

Composants principaux de Spring



Spring Framework Runtime



- ▶ **L'inversion de contrôle** (IoC) est une technique de programmation dans laquelle le couplage d'objets est lié au moment de l'exécution au lieu d'être effectué à la compilation
- ▶ C'est ce qu'on appelle du "**couplage faible**" : éviter les dépendances directes entre les classes de l'application
- ▶ En Spring, l'inversion de contrôle (IoC) est réalisée par **injection de dépendances** (DI) via l'annotation `@Autowired`

Exemple : Une classe A doit utiliser une classe B

```
.  
  
class B1 {  
    public String toString()  
    {return "Hello world";}  
}  
  
class A1 {  
    private B1 b;  
  
    public A1() {  
        this.b= new B1();  
        System.out.println(b);  
    }  
}
```

A1 a1 = new A1();

Dependance forte
A1 "connait" B1

```
interface I{};  
  
class B2 implements I {  
    public String toString()  
    {return "Hello world";}  
}  
  
class A2 {  
    private I i;  
  
    public A2() {  
        this.i = new B2();  
        System.out.println(i);  
    }  
}
```

A2 a2 = new A2();

Il reste une référence à B dans A

```
interface I{};  
  
class B3 implements I {  
    public String toString()  
    {return "Hello world";}  
}  
  
class A3 {  
    private I i;  
  
    public A3(I i) {  
        this.i=i;  
        System.out.println(i);  
    }  
}
```

I b3 = new B3();
A3 a= new A3(b3);

Couplage faible
On "injecte" la dépendance dans A3

- ▶ Spring est construit à l'aide de `Composants` auto-injectés
- ▶ Mais on peut aussi créer ses propres composants “injectables”
- ▶ Ces objets spéciaux sont annotés avec le stéréotype `@Component`.
- ▶ Spring scanne le code au démarrage, instancie immédiatement ces composants (Beans) et les range dans la collection `ApplicationContext`
- ▶ `@Autowired` devant un attribut permet de retrouver automatiquement le composant cité (sans import)

- ▶ Soit devant un attribut
Spring récupère automatiquement une instance du bon type
- ▶ Soit devant un setter
Spring appelle automatiquement le setter avec l'instance à la création de la classe
- ▶ Soit devant un constructeur
Quand la classe est créée avec un constructeur vide, Spring injecte automatiquement les instances nécessaires

L'injection sur le constructeur est la plus recommandée !

Différentes manières de comprendre @Autowired

► Version condensée

```
public class UserService {  
    @Autowired  
    private UserRepository userRepository;  
    .....  
}
```

► Version avec constructeur explicite

```
public class UserService {  
    private UserRepository userRepository;  
  
    @Autowired  
    public UserService(UserRepository userRepository) {  
        this.userRepository=userRepository;  
    }  
    .....  
}
```

Un exemple TRES simple

OBJ.java

```
public interface OBJ {}
```

MonOBJ.java

```
@Component
public class MonOBJ implements OBJ {
    private String name="taratata";
    public String toString()
        {return name;}
}
```

Dans n'importe quelle classe

```
@Autowired
OBJ o; // et pas forcément MonOBJ

public meth()
{
    System.out.println(o);
}
```

- ▶ Spring cherche une instance d'une implémentation de `OBJ` dans le classpath à l'exécution (même si c'est par héritage)
- ▶ Par défaut c'est un modèle `singleton` : 1 seule instance possible.

- ▶ Spring, scanne aussi l'intérieur des Composants à la recherche de Beans
- ▶ La déclaration d'un Bean se fait obligatoirement dans un composant
- ▶ Quand une méthode est annotée avec `@Bean`, Spring l'exécute et range le résultat comme un bean dont le nom est le nom de cette méthode
- ▶ Comme pour les composants le mode par défaut est `singleton`
- ▶ Ils sont aussi retrouvés automatiquement à l'exécution (via l'annotation `@Autowired`)

Exemple

Personne.java

```
public class Personne {  
    private String nom;  
    private String prenom;  
  
    Personne(String n, String p)  
    {  
        this.nom = n;  
        this.prenom = p;  
    }  
  
    public String toString() {  
        return nom + " " + prenom;  
    }  
}
```

Config.java

```
@Component  
//idealement @Configuration  
public class Config  
{  
    @Bean  
    Personne getPersonne()  
    { return new Personne("paul", "duchemin"); }  
}
```

Et dans n'importe quelle classe

```
@Autowired  
Personne p;  
public String meth() {  
    return p.toString();  
}
```

- ▶ Assez similaire
- ▶ C'est une affaire de style, de goût
- ▶ Ecrire une classe `Personne` puis annoter une méthode avec `bean` pour avoir une instance **est équivalent** à

Mettre une annotation `@Component` sur la classe `Personne`

- ▶ les deux sont instanciés au démarrage de l'appli
- ▶ Les deux sont par défaut en singleton
- ▶ les deux sont auto-injectables
- ▶ Le Bean permet néanmoins de gérer plus facilement des instances multiples et des scopes différenciés.

```
@SpringBootApplication
public class Tp11Application {
    public static void main(String[] args) {
        SpringApplication.run(Tp11Application.class, args);
    }
}
```

@SpringBootApplication inclut

- ▶ @Configuration : indique que cette classe est une source de définition de beans.
- ▶ @EnableAutoConfiguration : charge les composants nécessaires à ce type d'application (le DispatcherServlet si c'est du web par ex).
- ▶ @ComponentScan : charge les composants du développeur, notamment les contrôleurs et les Beans.

Un Exemple de composant

Le composant `ApplicationRunner` est fourni par Spring. Sa méthode `run` est automatiquement lancée à chaque démarrage du projet.

`@Component`

```
public class Demarrage implements ApplicationRunner {  
    private static final Logger logger =  
        LoggerFactory.getLogger(Demarrage.class);  
  
    public void run(ApplicationArguments args) throws Exception {  
        logger.info("Hello World !");  
    }  
}
```

Le composant `CommandLineRunner` est un autre composant quasi identique.

Ce composant est automatiquement instancié par Spring
et automatiquement lancé au démarrage

Exemple de Bean

ApplicationContext est lui-même un bean !

@Component

```
public class Demarrage implements ApplicationRunner {  
    private static final Logger logger =  
        LoggerFactory.getLogger(Demarrage.class);  
  
    @Autowired  
    private ApplicationContext applicationContext;  
  
    public void run(ApplicationArguments args) throws Exception {  
        for (String s : applicationContext.getBeanDefinitionNames())  
            logger.info("bean : "+s);  
    }  
}
```

Différents stéréotypes

Il y a 4 affinements à `@Component` :

- ▶ `@Service`
utilisé pour décrire des objets de la couche “métier” de l’appli
- ▶ `@Repository`
utilisé principalement pour la couche persistance (DAO), dans SpringData
- ▶ `@Controller`
utilisé principalement pour les contrôleurs web dans Spring MVC
- ▶ `@Configuration`
utilisé pour déclarer des `@Bean`

Tous sont candidats à l’autodétection via `@Autowired`
Tous fonctionnent de manière parfaitement identique

Différents scopes ...

Components **comme** Beans peuvent avoir différents scopes

- ▶ singleton (**par défaut**, toujours la même instance)
- ▶ prototype (une instance différente à chaque invocation)

Et plus tard, nous verrons un contexte web les scopes : `request`, `session`, `application`, `websocket`

Lever les ambiguïtés

Soit avec `@primary` soit avec `@Qualifier`
`@Component`

```
public class TestBeans implements ApplicationRunner {  
    public void run(ApplicationArguments args) throws Exception {  
        System.out.println(test);  
    }  
}
```

`@Bean`

`@Scope("prototype")`

`Integer getMonCompteur1() {return Integer.valueOf(1);}`

`@Bean`

`@Scope("prototype")`

`@Primary`

`Integer getMonCompteur2() {return Integer.valueOf(2);}`

`@Autowired`

`// @Qualifier("getMonCompteur1")`

`Integer test;`

Conclusion

- ▶ Spring est un framework open-source de développement java
 - ▶ Il existe de très nombreux modules pour Spring
 - ▶ Spring s'appuie sur la notion d'inversion de contrôle (IoC)
 - ▶ Les différents modules Spring fournissent de nombreux composants
 - ▶ Les `@Composant` et les `@Bean` sont rangés dans le bean `ApplicationContext`
 - ▶ Ces instances sont récupérables via `@Autowired`
-
- ▶ Toute la difficulté c'est de connaître ces composants ;-)

Le Framework Spring

Spring-Boot

Spring Data JDBC

Spring Data JPA

- ▶ Micro-Framework dérivé de Spring
- ▶ S'appuie sur des configurations par défaut pour faciliter la configuration de Spring
- ▶ Fournit un ensemble de composants pré-configurés ainsi qu'un client `spring cli`
- ▶ Facilite la construction et l'exécution d'une application "tout en un"
- ▶ **Plus aucune configuration XML nécessaire**
- ▶ Configuration hiérarchique et souple (dont : ligne de commande, propriété système, environnement, fichier de profile, configuration application ...)

On peut faire du Java "traditionnel" avec SpringBoot
(Parmi les composants offerts , il y a Spring MVC)

Attention aux versions !

2025	Springboot 3.4.0	Spring 6.2.x	Java 17-23	Tomcat 10.1
mai 2023	Springboot 3.1.x	Spring 6.1.x	Java 17-21	Tomcat 10
nov 2022	Springboot 3.0.x	Spring 6.0.x	Java 17-19	Tomcat 10
mars 2018	Springboot 2.0.x	Spring 5.3.x	Java 8-19	Tomcat 8
dec 2017	Springboot 1.5.x	Spring 4.x	Java 6	Tomcat 6

- ▶ La version de Springboot est indiquée dans le POM (version courante 3.3.5)
- ▶ Depuis Tomcat 10 passage de `javax` à `jakarta`
- ▶ les versions 1.5.x ne sont plus maintenues
- ▶ Modules utilisés : `mvn dependency:list | grep core`

Spring Initializr fournit automatiquement le `.pom` adapté à SpringBoot et aux modules souhaités (“Spring Boot Starter Projects”)

- ▶ Aller sur <https://start.spring.io/>
- ▶ Ajouter les dépendances nécessaires
WEB, Devtools, Actuator, JPA, Postgresql, ...
- ▶ Generate
- ▶ Décompresser le zip obtenu
- ▶ `mvn package`
 - ▶ `mvn spring-boot:run`
 - ▶ `java -jar target/monappli-0.0.1-SNAPSHOT.jar`



Project

☒ Gradle - Groovy ☐ Gradle - Kotlin
☐ Maven

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 3.0.2 (SNAPSHOT) ☒ 3.0.1 ☐ 2.7.8 (SNAPSHOT) ☐ 2.7.7

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 19 ☒ 17 ☐ 11 ☐ 8

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.

Spring Boot DevTools

DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Lombok

DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

H2 Database

SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

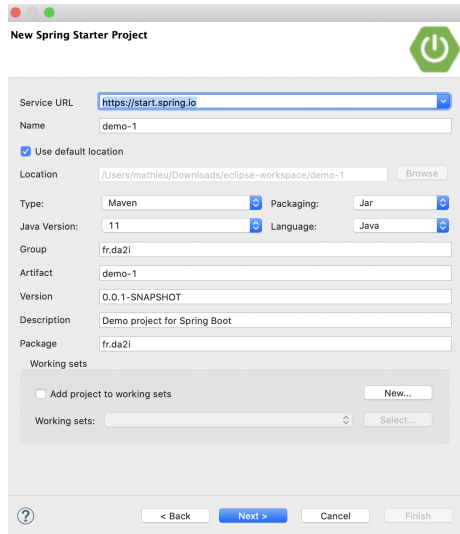


GENERATE ⌘ + ↵

EXPLORE CTRL + SPACE

SHARE...

Eclipse : File / new / other / Spring Starter project



The screenshot shows the 'New Spring Starter Project' dialog box in Eclipse. The dialog has a title bar with standard window controls and a green power icon. The main area contains several fields and options for configuring the project:

- Service URL:** A dropdown menu showing 'https://start.spring.io'.
- Name:** A text field containing 'demo-1'.
- Use default location:** A checked checkbox.
- Location:** A text field showing '/Users/mathieu/Downloads/eclipse-workspace/demo-1' and a 'Browse' button.
- Type:** A dropdown menu showing 'Maven'.
- Packaging:** A dropdown menu showing 'Jar'.
- Java Version:** A dropdown menu showing '11'.
- Language:** A dropdown menu showing 'Java'.
- Group:** A text field containing 'fr.da2l'.
- Artifact:** A text field containing 'demo-1'.
- Version:** A text field containing '0.0.1-SNAPSHOT'.
- Description:** A text field containing 'Demo project for Spring Boot'.
- Package:** A text field containing 'fr.da2l'.
- Working sets:** A section with a checkbox 'Add project to working sets' (unchecked), a 'New...' button, a 'Working sets:' dropdown menu, and a 'Select...' button.

At the bottom of the dialog, there is a help icon (?) and four buttons: '< Back', 'Next >', 'Cancel', and 'Finish'.

Spring fournit une interface en ligne de commande Spring CLI
(à installer à part)

- Permet d'initialiser les projets en une ligne !

```
spring --version
```

```
spring help init
```

```
spring init tp --build maven
```

```
spring init --build maven -d=web,h2 -g=fr.but3 tp
```

- Permet d'exécuter des scripts Groovy

```
spring shell
```

```
tp
|-- pom.xml
|-- src
    |-- main
        |-- java
            |-- fr
                |-- but3
                    |-- tp
                        |-- TpApplication.java
        |-- resources
            |-- application.properties
    |-- test
        |-- java
            |-- fr
                |-- but3
                    |-- tp
                        |-- TpApplicationTests.java
```

Modules principaux (Starter projects)

- ▶ **Spring Core.**
Auto-configuration, loggers, injection de dépendances(LoC), contexte basique
`@Component, @Bean, @Autowired, ApplicationContext`
principalement : `bean CommandLineRunner`
- ▶ **Spring Data JDBC**
principalement : objet `JdbcTemplate`
- ▶ **Spring Data JPA**
`@Entity, @Repository,`
principalement objet : `CrudRepository`
- ▶ **Spring Web MVC**
Support d'exécution WEB (Tomcat, Jetty, Undertow ...)
`@Controller, @RequestMapping, Model`
- ▶ **Spring Security**
`WebSecurityConfigurerAdapter, DelegatingPasswordEncoder`
- ▶ **Spring session**
Sessions indépendantes d'un conteneur spécifique

Le Framework Spring

Spring-Boot

Spring Data JDBC

Spring Data JPA

Principe

- ▶ **Spring Data JDBC** simplifie les accès aux BDD via JDBC
- ▶ Les caractéristiques de la BDD sont définies dans `application.properties`
- ▶ Fournit les classes `JdbcTemplate` et `NamedParameterJdbcTemplate` qui facilitent l'expression des requêtes
- ▶ Fournit la classe `RowMapper` qui permet de coder le mapping entre une ligne de résultat et l'objet Java correspondant
- ▶ Permet de passer directement un bean pour remplir les paramètres d'une requête grâce à la classe `BeanPropertySqlParameterSource`
- ▶ Exécute selon paramètres les fichiers `schema.sql`, `data.sql`, `import.sql`

L'objet clé : JdbcTemplate

- Bean `dataSource` automatiquement créé à partir des propriétés

```
spring.datasource.url=jdbc:postgresql://psqlserv/but3
spring.datasource.username=duchemin
spring.datasource.password=paul
```

```
# Pour import.sql
spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always
```

- Bean `JdbcTemplate` automatiquement créé avec cette datasource

```
@Autowired
JdbcTemplate jdbcTemplate;
```

Spring s'occupe de tout, y compris ouvrir et fermer les connexions !

`import.sql`, `schema.sql` et `data.sql` sont exécutés dès `mvn package` !

Principalement ... ([voir doc](#))

- ▶ `update` ou `batchUpdate` pour la mise à jour
- ▶ `query` avec ou sans `RowMapper`
- ▶ `queryForObject` ...
- ▶ `queryForList` ...
- ▶ `queryForMap` ...

(par défaut, SpringBoot2 utilise HikariCP comme pool)

Exemple

```
@Repository
public class DAO {
    @Autowired
    JdbcTemplate jdbcTemplate;

    class ProduitRowMapper implements RowMapper<Produit> {
        @Override
        public Produit mapRow(ResultSet rs, int rowNum) throws SQLException{
            Produit produit = new Produit();
            produit.id=rs.getInt("id");
            produit.prenom=rs.getString("prenom");
            produit.nom=rs.getString("nom");
            return produit;
        }
    }

    public List<Produit> findAll() {
        return jdbcTemplate.query("select * from produit", new ProduitRowMapper()); }

    public int insert(Produit produit) {
        return jdbcTemplate.update("insert into produit values(?, ?, ?)",
            new Object[] { produit.id, produit.prenom, produit.nom });
    }

    public int getCount() () {
        return jdbcTemplate.queryForObject("select count(*) from personne", Integer.class)
    }
}
```

- ▶ `mvn clean package`
Tout ce qui touche à la BDD est immédiatement exécuté
Une base mal paramétrée plante déjà ici !
- ▶ `mvn spring-boot:run`
Tout est ré-exécuté à nouveau !

Et lire les traces !!

```
....  
H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:lapin'  
....  
Tomcat started on port(s): 8080 (http) with context path ''
```

Le Framework Spring

Spring-Boot

Spring Data JDBC

Spring Data JPA

- ▶ **Spring Data JPA** permet d'injecter automatiquement une classe DAO (composant de type "repository") pour chaque Entity Bean
- ▶ Cette classe contient automatiquement les méthodes CRUD, des méthodes de tri ou de pagination : `count`, `delete`, `deleteById`, `save`, `saveAll`, `findById` et `findAll` (**voir doc**)
- ▶ La classe repository est de préférence indiquée avec l'annotation `@Repository`
- ▶ Elle doit étendre l'une des interfaces `JpaRepository`, `PagingAndSortingRepository`, `CrudRepository`,

Exemple

POJO User.java à créer

```
@Entity
public class User {
    @Id
    private Long id;
    private String nom;
    ....
}
```

Repository à définir : UserRepository.java

```
public interface UserRepository extends CrudRepository<User, Long> {}
// Et on a automatiquement le CRUD sur USER
// Mais on peut bien sûr en rajouter d'autres selon les besoins
```

Dans le controleur

```
@Controller
public class MonContrôleur {
    @Autowired
    private UserRepository userRepository;
    .... userRepository.findAll()
}
```

Toutes les propriétés doivent être définies dans `application.properties` ou `application.yml`

```
spring.datasource.url=jdbc:postgresql://psqlserv/but3
spring.datasource.username=duchemin
spring.datasource.password=paul
spring.jpa.show-sql=true                                (default false)
spring.jpa.generate-ddl=true                             (default false)
spring.jpa.hibernate.ddl-auto=create                     (default none)
```

Automatiquement

- ▶ Il charge le bon driver
- ▶ Il optimise pour le SGBD utilisé
- ▶ Il exécute les fichiers `schema.sql+data.sql` ou `import.sql` placés n'importe où (a priori dans `resources`)
- ▶ La propriété `spring.jpa.hibernate.ddl-auto` peut prendre les valeurs `create`, `create-drop`, `validate`, `update`
- ▶ Attention : Hibernate génère les colonnes dans l'ordre alphabétique !

findById

- ▶ Depuis la version 2.0 de Spring JPA, la méthode `findOne` a été renommée `findById`
- ▶ Le type de retour est maintenant `Optional<T>` `findById(ID id)` ;
- ▶ Permet d'implémenter différentes gestions en cas d'absence de l'élément :

- ▶ Renvoyer une erreur

```
return repository.findById(id)
    .orElseThrow(() -> new EntityNotFoundException(id));
```

- ▶ Créer l'objet

```
Foo f = repo.findById(id).orElse(new Foo());
```

- ▶ Renvoyer null

```
Foo f = repo.findById(id).orElse(null);
```

Il est possible d'accéder directement aux propriétés JPA en préfixant ces propriété par `spring.jpa.properties`

Par exemple pour les traces de création et destruction :

```
s.j.p.jakarta.persistence.schema-generation.scripts.action=create  
s.j.p.jakarta.persistence.schema-generation.scripts.create-target=creer.sql  
s.j.p.jakarta.persistence.schema-generation.scripts.drop-target=détruire.sql
```

Mais attention aux incompatibilités ! c'est parfois périlleux.

Par exemple, si les lignes précédentes sont indiquées il faut aussi rajouter :

```
s.j.p.jakarta.persistence.schema-generation.database.action=create
```

Il est quand même préférable d'éviter de court-circuiter SpringBoot ;-)

Spring Data permet d'écrire des méthodes à partir des noms d'attributs et quelques **mots-clés** (And, Or, Containing, StartingWith, etc). Il se charge de traduire automatiquement ce nom de méthode en requête puis de l'exécuter !

```
public interface PersonneRep extends CrudRepository {  
    // recherche une personne par son attribut "nom"  
    Personne findByNom(String nom);  
  
    // ici, par son "nom" ou "prenom"  
    Personne findByNomOrPrenom(String nom, String prenom);  
  
    List<Personne> findByNomAndPrenomAllIgnoreCase(String nom, String prenom);  
  
    List<Personne> findByNomOrderByPrenomAsc(String nom);  
}
```

Ajouter ses propres requêtes : l'annotation `@Query`

```
@Repository
public interface PersonneRepo extends CRUDRepository {
    ....
    @Query(value = "select prenom, count(*) " +
        "from users" +
        "GROUP BY prenom ",
        nativeQuery = true)
    List<PrenomCpt> getNombrePrenoms();
}
```

Il est aussi possible d'utiliser JPQL, de créer son propre objet de récupération, ou même d'utiliser automatiquement une Map

Avec Spring JPA il y a en général très peu de code à écrire !



► POM

```
com.h2database
```

► Différentes URL (voir [liste](#))

```
jdbc:h2:~/test      # embedded      (mono util)
jdbc:h2:mem:test    # in-memory      (multi connect)
jdbc:h2:tcp://localhost/~/test  (client-serveur)
```

► properties

```
spring.h2.console.enabled=true
# Autorise l'url http://localhost:8080/h2-console  (fait par défaut)
```

► fichiers

```
main/resources/import.sql
```

- ▶ SpringBoot simplifie la configuration d'un projet Spring
- ▶ SpringBoot offre de nombreux “starters” pré-configurés qui facilitent le développement
- ▶ SpringInitializer ou Spring CLI créent automatiquement le projet
- ▶ Spring Data JDBC fournit principalement `JdbcTemplate`
- ▶ Spring Data JPA fournit principalement `CrudRepository`
- ▶ Avec Spring JPA quasi plus rien à écrire pour l'accès à la BDD !