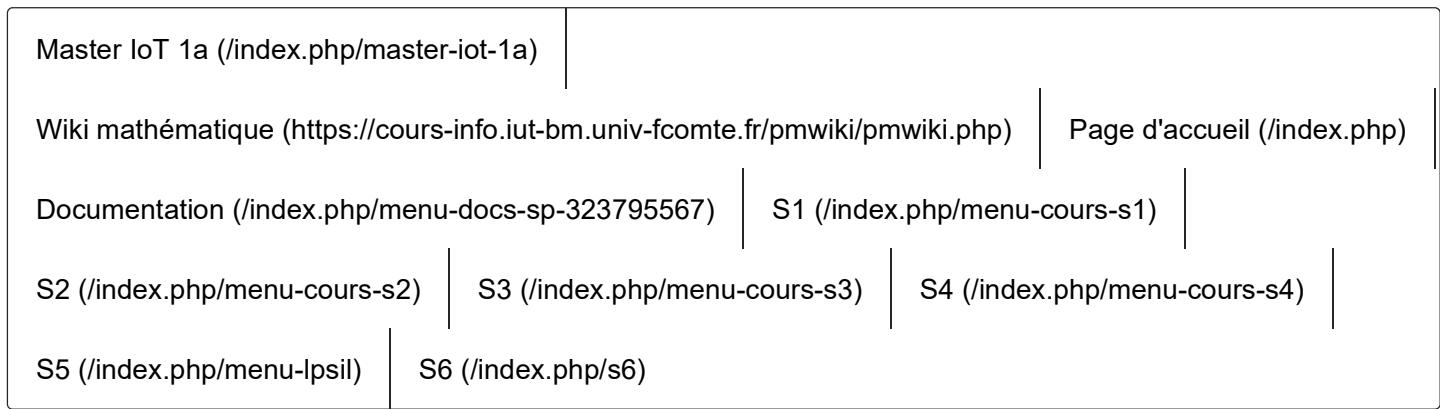




Menu principal



norrmnext.com (<http://norrmnext.com>)

Connexion

| | |
|--|--------------|
| | Identifiant |
| | Mot de passe |

Se souvenir de moi

[Connexion](#)

Identifiant oublié ? (</index.php/component/users/?view=remind&Itemid=125>)

Mot de passe oublié ? (</index.php/component/users/?view=reset&Itemid=125>)

R6.05 - Prog. Frameworks : Spring boot

TD n°1 : découverte de Spring boot

Détails



Écrit par stéphane Domas

Catégorie : R6.05 - Prog. Framework : Spring boot (</index.php/s6/objets-connectes>)

Publication : 31 décembre 2020

Affichages : 1702

1°/ Spring, Spring boot : kesako ?

Spring est un framework Java dont la première version est sortie en 2004. Cet environnement permet d'écrire des applications Java de tout type mais en suivant les principes J2E (Java Enterprise Edition). Son objectif premier est de faciliter la vie des développeurs car le J2E des origines était horriblement complexe à utiliser. Cependant, la "philosophie" derrière Spring reste relativement différente de la façon classique de construire une application Java, par exemple, comme on peut la découvrir au cours du BUT. C'est pourquoi Spring reste compliqué à apprendre car il nécessite de connaître beaucoup de concepts et de nouvelles façons de structurer le code, et ce plus encore quand l'application doit interagir avec une base de données, ou bien quand il s'agit d'écrire un micro-service. Un point notable est que Spring nécessite d'écrire un nombre conséquents de fichier XML pour configurer tout un tas d'aspects d'une application, et que ces fichiers sont quasi illisibles pour quelqu'un ne connaissant pas Spring.

Spring Boot est sorti en 2014. Il est une surcouche de Spring qui simplifie de façon drastique l'écriture de micro-services.

Spring boot n'est donc pas fait pour créer une application "sans serveur", mais pour justement créer le côté serveur. Par exemple, Spring Boot est particulièrement indiqué pour créer une API REST, mais il est tout aussi utile pour créer une API non REST, ou un serveur de page Web. L'énorme avantage est que Spring Boot ne nécessite pas de connaître Spring, et que le résultat est relativement compréhensible, même par un novice en Spring Boot. De plus, le volume de code à écrire est vraiment très faible, même si on le compare à son équivalent en nodejs. Cela vient essentiellement des annotations (par exemple `@RestController`) utilisées par Spring Boot, qui permettent soit de générer automatiquement un volume important de code lors de la compilation, soit de faire des tâches bien précises lors de l'exécution. Il y a également des principes de structuration du code et de nommage des classes/méthodes, qui réduisent encore le volume de code à écrire. L'inconvénient majeur est que l'on ne comprend pas forcément l'influence de telle ou telle annotation et pourquoi on doit créer tel ou tel fichier. Pour résumé, quand on suit une procédure éprouvée pour créer son application, généralement tout va bien, mais dès que quelque chose plante, cela devient compliqué de comprendre pourquoi sans une forte expertise de Spring Boot.

2°/ Premiers pas en Spring Boot

Sur le site de Spring, on trouve bon nombre de tutoriaux (<https://spring.io/guides> (<https://spring.io/guides>)) permettant de découvrir Spring boot. La plupart sont facilement compréhensibles mais ils n'expliquent pas forcément la "logique" utilisée ou pourquoi on utilise telle ou telle annotation, structure de code etc. Ces guides sont donc une source d'exemples intéressante mais pas vraiment d'explications. D'autres sites proposent des ressources très intéressantes sur Spring Boot, souvent en relation avec d'autres frameworks, ou bien dans un contexte précis comme par exemple Spring Boot+MongoDb. On peut citer entre autres :

- <https://www.baeldung.com/rest-with-spring-series> (<https://www.baeldung.com/rest-with-spring-series>)
- <https://www.bezkoder.com/category/spring/> (<https://www.bezkoder.com/category/spring/>)

A noter que sur les deux sites mentionnés ci-dessus, on trouve également pas mal d'articles avec des explications sur certaines classes, annotations, ... de Spring Boot. Ils sont donc souvent plus intéressants que les guides du site Spring, à condition de farfouiller !

L'archive du projet contenant les démonstrations qui suivent peut être téléchargé [ici (</upload/supports/S6/SpringBoot/TD/td1/springboot-td1-src.tgz>)]

2.1°/ API Hello world

L'objectif de ce premier exemple est de créer une API très simple avec 2 routes, dont une avec paramètre :

- / : renvoie le message "Welcome Home"
- /hello/{name} : renvoie le message "Hello name" si name existe, et un message d'insulte sinon.

Pour créer un projet Spring Boot, le plus simple est de faire comme dans les guide : obtenir un squelette de projet via le site Spring Initializr : <https://start.spring.io/> (<https://start.spring.io/>). On peut générer et télécharger une archive zip contenant tout ce qu'il faut pour commencer le projet en précisant le langage (Java, Kotlin, Groovy), le gestionnaire de projet voulu (gradle, maven), et d'autres paramètres de version (Spring, Java, ...). Le seul problème est de choisir les dépendances nécessaires au projet.

Quand on veut créer une API REST, il faut au moins sélectionner "Spring Web" comme dépendance. Si cette API doit être en relation avec une base de données, d'autres dépendances sont nécessaires, comme montré dans la sections suivante.

La figure 1 ci-dessous montre l'état de la fenêtre pour créer le projet qui sert de démonstration 1.

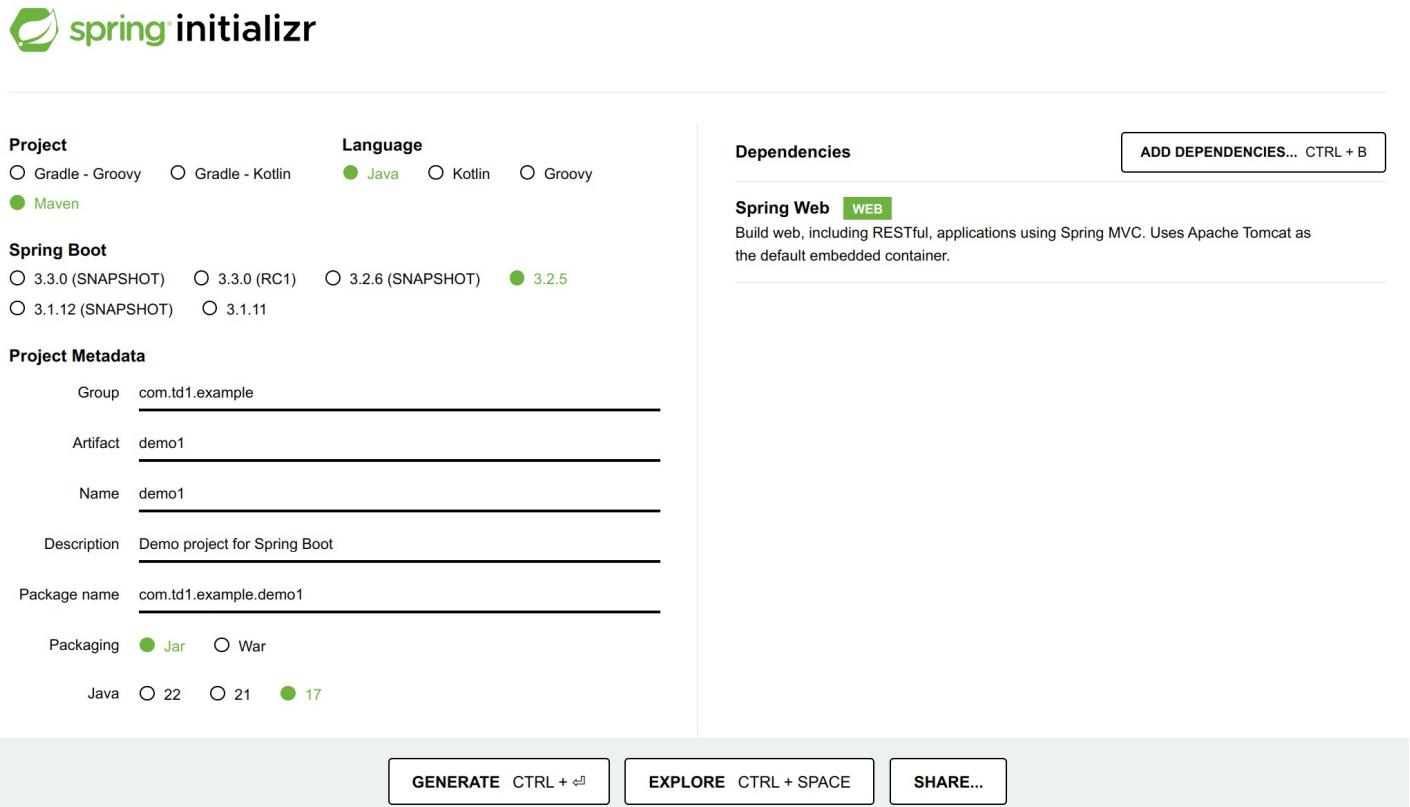


Figure 1 : sélection des paramètres (Java + maven + Spring Web) de la démonstration 1 sur Spring Initializr

On remarque dans la description de la dépendance "Spring Web" qu'elle repose sur Spring MVC et Tomcat. Le premier représente simplement le module de développement d'application Web de Spring. Il contient les classes, annotations, et mécanismes d'exécution basiques pour toute application web Spring. Cependant, il faut pouvoir "servir" le résultat via un serveur HTTP et c'est le rôle de Tomcat qui est en gros un serveur Web sachant interpréter/exécuter du Java.

En cliquant sur "Generate", on télécharge une archive zip du même nom que le projet, donc dans l'exemple demo1.zip. Après décompactage, un répertoire demo1 est créé et on peut directement ouvrir ce répertoire, sous par exemple IDEA.

ATTENTION : ne jamais utiliser "Nouveau projet à partir de sources existantes".

Après avoir "Accepter de faire confiance à ce projet", IDEA télécharge toutes les dépendances nécessaires (indiquées dans pom.xml). Après quelques secondes/minutes, on peut commencer le développement. Le seul fichier source existant est celui représentant le main() de l'application, comme on peut le voir dans la figure 2.

The screenshot shows the IntelliJ IDEA interface with the 'demo1' project open. The 'Project' tool window on the left shows the directory structure: demo1 (containing .idea, .mvn, src, test, .gitignore, demo1.iml, HELP.md, mvnw, mvnw.cmd, pom.xml, External Libraries, and Scratches and Consoles). The 'Structure' tool window on the right shows the Maven dependencies. The code editor displays Demo1Application.java:

```

package com.td1.example.demo1;
import ...
@SpringBootApplication
public class Demo1Application {
    public static void main(String[] args) { SpringApplication.run(Demo1Application.class, args); }
}

```

Figure 2 : fenêtre IDEA après ouverture du squelette de projet.

Il suffit maintenant d'ajouter des fichiers dans le répertoire des sources (src/main/java/com.td1.example.demo1), en suivant la structuration dictée par Spring MVC, ce qui ressemble fortement à ce que l'on ferait pour écrire une API REST avec flask. On est donc censé écrire du code pour chacune des parties Modèle, Vue et Contrôle, et pour bien faire, de mettre ce code dans des répertoires différents.

Dans cette démonstration, on a juste besoin de définir la partie contrôle, c'est-à-dire créer la route et le traitement qui doit être fait quand un navigateur la demande. Pour cela :

- On ajoute un package `control`,
- Dans ce package, on ajoute un fichier `HelloController.java`

Ce fichier contient :

```

1 package com.td1.example.demo1.control;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import org.springframework.web.bind.annotation.PathVariable;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RestController
8 public class HelloController {
9
10     @GetMapping("/")
11     public String home() {
12         return "<h2>Welcome home</h2>";
13     }
14
15     @GetMapping(value= {"/hello","/hello/{name}"})
16     public String hello(@PathVariable(required = false) String name) {
17         if (name != null) {
18             return "Hello " + name;
19         }
20         return "Go to hell, impolite one !";
21     }
22 }
```

Explications :

- L'annotation `@RestController` permet de spécifier que cette classe représente un contrôleur pour une API REST, ce qui implique notamment que les méthodes qu'elle contient peuvent se contenter de retourner une valeur afin que celle-ci soit automatiquement empaquetée dans la réponse au client, soit sous forme de simple chaîne de caractère comme dans cette démonstration, soit en JSON (cf. démo suivante).
- NB : pour les applications web non API, qui veulent retourner du contenu HTML, il est préférable d'utiliser l'annotation `@Controller`, comme abordé dans un futur article.
- L'annotation `@GetMapping` permet d'associer une route pour une requête GET à la méthode contrôleur qui vient juste après. Le contenu entre parenthèse peut être simplement la route, ou bien plusieurs champs afin de paramétriser la requête. Dans le code ci-dessus, on voit que le champ `value` permet de spécifier les différentes routes possibles pour appeler le contrôleur qui suit. Il est également possible de définir des entêtes, des paramètres, etc.

Démonstration :

- Le projet généré contient déjà une configuration d'exécution. Il suffit donc de cliquer sur la flèche verte pour lancer la compilation puis l'exécution.
- On remarque qu'un répertoire `target` s'est créé à la racine du projet. C'est lui qui contient le résultat de la compilation.
- On constate dans la console qu'il y a bien un serveur tomcat lancé sur le port 8080 (cf. Figure 3)

```

:: Spring Boot :: (v3.2.5)

2024-04-20T11:56:19.010+02:00 INFO 13556 --- [demo1] [main] com.tdi.example.demo1.Demo1Application : Starting Demo1Application using Java 17.0.10 with PID 13556 (/home/sdomas/cours/SpringBoot/TD/td1/demo1/target)
2024-04-20T11:56:19.012+02:00 INFO 13556 --- [demo1] [main] com.tdi.example.demo1.Demo1Application : No active profile set, falling back to 1 default profile: "default"
2024-04-20T11:56:19.297+02:00 INFO 13556 --- [demo1] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port 8080 (http)
2024-04-20T11:56:19.302+02:00 INFO 13556 --- [demo1] [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2024-04-20T11:56:19.303+02:00 INFO 13556 --- [demo1] [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.20]
2024-04-20T11:56:19.320+02:00 INFO 13556 --- [demo1] [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2024-04-20T11:56:19.326+02:00 INFO 13556 --- [demo1] [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 288 ms
2024-04-20T11:56:19.429+02:00 INFO 13556 --- [demo1] [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with context path ''
2024-04-20T11:56:19.433+02:00 INFO 13556 --- [demo1] [main] com.tdi.example.demo1.Demo1Application : Started Demo1Application in 0.559 seconds (process running for 0.734)

```

Figure 3 : affichage console au lancement de la démonstration 1

- Dans un navigateur, taper `localhost:8080`. On obtient l'affichage du titre "Welcome home". On constate que la chaîne renvoyée par l'API contient du HTML qui a été interprété par le navigateur. Si on va dans l'inspecteur, onglet "Réseau" et que l'on clique sur la requête, on voit que dans les entêtes de réponse on a : `Content-Type : text/html`. C'est pour cela que le navigateur interprète le contenu de la réponse comme du HTML.
- si on tape l'URL `localhost:8080/hello/toto`, on obtient l'affichage "Hello toto"
- si on tape l'URL `localhost:8080/hello`, on obtient l'affichage "Go to hell .."
- si on tape l'URL `localhost:8080/hello/`, on obtient un message d'erreur. En effet, toute route non conforme affiche ce message par défaut. On verra comment le modifier dans les futurs articles.
- si on tape n'importe quelle autre route, par ex. `localhost:8080/salut`, on obtient également ce message d'erreur par défaut.

2.2°/ API avec base de données type SQL en mémoire/fichier

L'objectif de ce deuxième exemple est d'illustrer comment utiliser la base de données H2 ainsi que JPA/Hibernate afin de gérer de façon transparente les accès à la BdD, c'est-à-dire sans écrire de requête SQL mais en manipulant simplement des objets. Les opérations de manipulation de la BdD se font au travers d'une API. Cet exemple sert également à montrer comment initialiser la BdD à partir d'un fichier SQL, et/ou d'instructions Java.

ATTENTION : cet exemple n'utilise qu'une seule table et n'aborde donc pas comment prendre en compte des références entre des tables (avec clé étrangère), ou comment gérer les interactions one-to-many, many-to-one, ... Ce sera l'objet du cours suivant.

Voici les grandes lignes afin de créer ce type d'application :

- Chaque table va être décrite par une classe de type POJO (Plain Old Java Object) avec des annotations. Ces classes doivent généralement contenir :
 - un attribut représentant la clé primaire,
 - des attributs représentant les colonnes, potentiellement annotées, si par exemple le nom de l'attribut doit être un peu différent du nom de colonne, si la valeur est non nulle, ...
 - si besoin des annotations pour décrire des relations (one-to-many, many-to-one, ...) avec d'autres tables, donc d'autres POJO.
 - un ou plusieurs constructeurs,
 - des getters/setters pour chacun des attributs, sauf généralement la clé primaire
- Grâce à ces classes, nommées "entités", Spring Boot va être capable de créer la BdD et les tables correspondantes, si besoin, mais surtout de générer tout le code permettant de faire le lien entre des instances de ces entités et la BdD. Pour cela, Spring utilise un ensemble de fonctionnalités décrites par JPA sous la forme d'interfaces, et par défaut Hibernate qui implémente ces fonctionnalités. Hibernate est donc l'ORM qui va s'occuper de faire les requêtes SQL, sans que l'on ait normalement besoin d'écrire du SQL.
- Pour gérer très facilement l'accès aux tables et donc obtenir des instances d'entités, Spring Boot définit la notion de repository.
- Un repository est simplement une classe interface déclarant les méthodes d'accès à la BdD pour en obtenir des entités. Le côté "magique" est qu'il n'est pas nécessaire d'implémenter les méthodes car Spring Boot va le faire automatiquement. En effet, il est capable de déterminer quelle requête exécuter en fonction du nom des méthodes.

2.2.1°/ configuration générale de l'application

Pour créer le squelette de base du projet dans Spring Initializr, il faut ajouter comme dépendances : "Spring Web", "Spring Data JPA", "H2 database". Une fois le projet ouvert sous IDEA, la première chose à faire est de modifier la fichier de configuration principale de l'application, à savoir `src/main/resources/application.properties`. Il faut ajouter à ce fichier tous les paramètres de configuration liés à l'accès à la BdD. Par exemple, pour cette démonstration, cela donne :

```

1 # 1 - app name
2 spring.application.name=demo2
3
4 # 2 - create the tables inferring from entities. NB : use update to just update the table stru
5 spring.jpa.hibernate.ddl-auto=create
6 # 3 - delay the execution of data.sql until database+tables are created
7 spring.jpa.defer-datasource-initialization=true
8
9 # 4.1 - setup db type I: H2 in memory with db name = herocorp
10 spring.datasource.url=jdbc:h2:mem:herocorp
11
12 # 4.2 - setup db type II: H2 in a "file" ~/herocorpdb. NB : must set init.mode to always to ev
13 #spring.sql.init.mode=always
14 #spring.datasource.url=jdbc:h2:file:~/herocorpdb
15
16 # 5 - enable the db console to have an access with url like localhost:8080/h2-console
17 spring.h2.console.enabled=true
18
19 # 6 - uncomment to show SQL requests on console
20 #spring.jpa.show-sql=true

```

Remarques :

- La 2ème ligne de configuration permet d'indiquer à Hibernate ce qu'il doit faire au sujet de la BdD lors du lancement de l'application. Dans le cas présent, il recrée les tables à chaque lancement, ce qui serait stupide en mode production, avec une vraie base de donnée, par exemple postgres. Il est donc possible de spécifier comme valeur `none` pour qu'il ne fasse rien. Dans ce cas, il est cependant possible d'utiliser un script SQL pour créer les tables, en le mettant dans `/src/main/resources/script.sql`
- Si Hibernate doit créer lui-même les tables, il va utiliser les entités pour inférer la structure des tables à créer.
- Pour initialiser des tables, il faut créer un fichier `src/main/resources/data.sql` contenant les instructions SQL pour remplir les tables. Malheureusement, il faut que ces tables existent au préalable et c'est pourquoi il faut attendre que hibernate ait fini d'analyser les entités pour créer la BdD et les tables. C'est le rôle de la 3ème ligne de configuration.
- Il faut également indiquer quel type de BdD on utilise. Dans le cas présent, c'est une BdD "embarqué", qui peut soit être placée en mémoire, soit dans un fichier, comme on le voit avec les lignes de configuration 4.1 et 4.2. L'avantage du deuxième cas est que la BdD ne disparaît pas après l'arrêt de l'application.
- Les deux dernières lignes sont plus en vue du débogage.

2.2.2°/ Structuration en packages

Pour structurer correctement le code, il est souhaitable de créer différents packages :

- `model` : contient les classes représentant les entités,
- `repository` : contient les classes représentant les repository
- `service` : contient les classes qui vont utiliser les repository pour récupérer des données
- `control` : contient les classes qui vont créer les routes et accéder aux classes de service pour récupérer les données.

A noter que dans les cas très simples, on peut également créer des contrôleurs qui utilisent directement les repository. Cependant, le fait de créer des services permet de conserver les contrôleurs tels quels même si on change de type de repository, par exemple quand on change de type de BdD.

2.2.3°/ Les entités

Dans cette démonstration, on suppose que l'on manipule une table représentant un héro. C'est pourquoi on crée un fichier model/Hero.java, comme suivant :

```
1 package com.td1.example.demo2.model;
2 import jakarta.persistence.*;
3
4 @Entity
5 @Table(name = "heroes")
6 public class Hero {
7
8     @Id
9     @GeneratedValue(strategy= GenerationType.IDENTITY)
10    private Long id;
11
12    private String publicName;
13    private String realName;
14    private String power;
15    private int powerLevel;
16
17    public Hero() {}
18    ... // other constructors, getters/setters
19 }
```

Remarques :

- l'annotation `@Entity` permet de spécifier que le POJO représente un entité
- `@Table` est optionnelle mais elle permet de changer le nom de la table, qui sinon est déduit du nom de la classe.
- `@Id` permet de définir l'attribut qui suit comme étant la clef primaire
- `@GenerateValue` indique la façon d'initialiser cette clef. Dans le cas présent, la stratégie utilisée (IDENTITY) est celle du SGBD utilisé, donc H2. D'autres stratégies sont possible, notamment en utilisant hibernate (AUTO), ou bien de façon générative.
- Le nom des attributs donne indirectement le nom de la colonne dans la BdD. Par exemple, `publicName` correspond à la colonne `public_name`. Il est possible de change ce nom avec l'annotation `@Column`

2.2.4°/ Les repository

La création d'un repository se fait en créant une interface héritant d'une des interface déjà existante. Dans le cas présent, vu que l'on utilise JPA, on va hériter de `JpaRepository`. Il est également possible d'hériter de classes plus "simples" telles que `CrudRepository`, mais cela limite les possibilités de récupérer des données depuis la BdD.

Le côté magique de Spring Boot vient en grande partie du fait que la plupart du temps, on a juste besoin de créer une interface mais pas ensuite de l'implémenter. En effet, Spring Boot est capable de créer tout seul l'implémentation des méthodes de nos repository juste en se basant sur leur nom. Par exemple, pour cette démonstration, on créer le fichier repository/HeroesRepository.java :

```
1 package com.td1.example.demo2.repository;
2
3 import com.td1.example.demo2.model.Hero;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 import java.util.List;
7
8 public interface HeroesRepository extends JpaRepository<Hero, Long> {
9
10     // NB : no need to implement methods below : spring will do it automatically inferring it
11     Hero findByPublicName(String publicName);
12     List<Hero> findAllByPower(String power);
13 }
```

Remarques :

- Le premier type générique utilisé dans `JpaRepository< ..., ...>` doit être la classe entité pour laquelle on veut créer des méthodes d'accès à la BdD. Dans le cas présent, c'est la classe `Hero`. Le deuxième paramètre correspond au type de l'attribut représentant la clef primaire, dans ce cas `Long`.
- Le fait d'hériter de `JpaRepository` permet d'utiliser des méthodes "passe-partout" telles que `findAll()`, `findById()`, `save()`, `delete()`, `count()`, ...
- Pour les cas où il faut récupérer des données de façon plus ciblée, il est possible de déclarer des méthodes avec des noms bien précis qui vont être utilisés pour définir la requête qui sera faite. Des exemples sont données ici : <https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html> (<https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html>)
- La valeur de retour des méthodes est généralement le type de l'entité ou bien une liste de ce type. Cependant, cela peut être n'importe quel type valide par rapport à ce que renvoie la requête : entier, double, string, ...

2.2.5° Les services

Dans cette démonstration, on crée un service unique qui utilise l'unique repository. De plus, les méthodes de ce service se contentent d'appeler celles du repository. On pourrait donc considérer que la couche service est inutile. C'est plutôt vrai pour cette démonstration mais pas du tout dans le cas général. En effet, le fait de créer un service permet de découpler les contrôleurs de la partie accès à la BdD via les repository. Si on change le type de BdD, il suffit de changer le repository et le service, alors que le contrôleur reste le même. De plus, il est parfaitement possible de définir des méthodes de service qui vont utiliser plusieurs repository afin de fournir des données au contrôleur. C'est pourquoi il est fortement conseillé d'utiliser une couche de service, même dans les projets simples.

Dans le cas présent, on créer un fichier `service/HeroesService.java` :

```
1 package com.td1.example.demo2.service;
2
3 import com.td1.example.demo2.model.Hero;
4 import com.td1.example.demo2.repository.HeroesRepository;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import java.util.List;
9
10 @Service
11 public class HeroesService {
12     private HeroesRepository heroesRepository;
13
14     @Autowired
15     public HeroesService(HeroesRepository heroesRepository) {
16         this.heroesRepository = heroesRepository;
17     }
18
19     public List<Hero> findAll() {
20         System.out.println("Getting all heroes");
21         List<Hero> list = heroesRepository.findAll();
22         return list;
23     }
24     ...
25 }
```

Remarques :

- Il est conseillé d'annoter un service avec `@Service` plutôt que `@Component` pour que Spring le classe comme étant de la couche service. Mais cela n'a pas vraiment d'importance tant que l'on utilise pas de mécanisme Spring basé sur le type de composants.
- `@Autowired` devant le constructeur indique à Spring qu'il faut automatiquement créer une instance de `HeroesRepository` pour la donner en paramètre du constructeur. On a donc pas besoin de la créer soi-même ailleurs. (NB : comme on a qu'un seul constructeur, il est possible de ne pas utiliser `@Autowired`)

2.2.6°/ Les contrôleurs

La partie contrôle a exactement la même forme que dans la démonstration 1, excepté que l'on doit faire appel aux services. Dans le cas présent, cela donne un fichier `control/HeroesController.java` :

```
1 package com.td1.example.demo2.control;
2
3 import com.td1.example.demo2.model.Hero;
4 import com.td1.example.demo2.service.HeroesService;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.PathVariable;
7 import org.springframework.web.bind.annotation.RequestParam;
8 import org.springframework.web.bind.annotation.RestController;
9
10 import java.util.List;
11
12 @RestController
13 public class HeroesController {
14
15     private final HeroesService heroesService;
16
17     HeroesController(HeroesService heroesService) {
18         this.heroesService = heroesService;
19     }
20
21     @GetMapping("/heroes")
22     public List<Hero> getAllHeroes() {
23         System.out.println("Getting all heroes");
24         List<Hero> list = heroesService.findAll();
25         return list;
26     }
27     ...
28     @GetMapping("/heroes/getbypower")
29     public List<Hero> getHeroesByPower(@RequestParam String power, @RequestParam(required = f
30         List<Hero> list = null;
31         if ((pattern == null) || (pattern == false)) {
32             System.out.println("Getting heroes with power " + power);
33             list = heroesService.findAllByPower(power);
34         }
35         else {
36             list = heroesService.findAllByPowerContaining(power);
37         }
38         return list;
39     }
40     ...
41 }
```

Remarques :

- Comme on a qu'un seul constructeur prenant en paramètre un service, il n'est pas utile d'utiliser `@Autowired` pour créer automatiquement le service (cf. rq en 2.2.5)
- A part les paramètres de route, on remarque qu'il est possible d'extraire les paramètres de la requête HTTP, via l'annotation `@RequestParam`
- Si l'on veut que des paramètres de requête soient optionnels, il faut :
 - utiliser `(required = false)` comme modificateur de l'annotation,
 - mettre un type objet pour le paramètre de la méthode représentant le paramètre de requête, afin que s'il n'existe pas dans l'URL, la valeur du paramètre de la méthode soit `null`.

2.2.7°/ Les runners

Lors du lancement de l'application et après toutes les initialisation, il est possible de lancer des "jobs", grâce à l'interface `CommandLineRunner`. Il suffit d'implémenter cette interface afin d'obtenir un thread qui sera exécuter automatiquement après toutes les initialisations. Ce thread peut être utilisé pour à peu près n'importe quelle opération, dont le remplissage de la BdD via des instructions. Par exemple, on peut créer un fichier `MyRunner.java`, dans le même répertoire que le fichier `main()`, avec :

```
1 package com.td1.example.demo2;
2
3 import com.td1.example.demo2.model.Hero;
4 import com.td1.example.demo2.repository.HeroesRepository;
5 import org.slf4j.Logger;
6 import org.slf4j.LoggerFactory;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.boot.CommandLineRunner;
9 import org.springframework.stereotype.Component;
10
11 import java.util.List;
12
13 @Component
14 public class MyRunner implements CommandLineRunner {
15     private static final Logger logger = LoggerFactory.getLogger(MyRunner.class);
16
17     private final HeroesRepository heroesRepository;
18
19     @Autowired
20     public MyRunner(HeroesRepository heroesRepository) {
21         this.heroesRepository = heroesRepository;
22     }
23
24     @Override
25     public void run(String... args) throws Exception {
26
27         // add a new hero, using a entity instance
28         Hero phil = new Hero("stickman", "phil deferre", "superthin", 99);
29         heroesRepository.save(phil);
30
31         logger.info("getting all heroes");
32         List<Hero> list = heroesRepository.findAll();
33         for(Hero h : list) {
34             System.out.println(h);
35         }
36     }
37 }
```

Remarques :

- Un `CommandLineRunner` doit définir une méthode `run()` qui représente son point d'entrée d'exécution.
- Pour remplir la BdD de façon programmatique, il suffit de créer des instances des entités puis d'utiliser le repository associé et sa méthode `save()`. Attention, dans cet exemple, il n'y a aucun test d'exception mais il faudrait normalement utiliser try/catch au cas où l'insertion en BdD échoue.

Démonstration :

- quand on lance l'application, on remarque dans le console que le runner s'exécute bien après toutes les initialisations et affiche tous les héros, y compris celui ajouté via des instructions.
- si on tape l'URL `localhost:8080/heroes`, on obtient la liste de tous les héros, preuve que la BdD a bien été initialisée.
- si on tape l'URL `localhost:8080/h2-console`, on obtient une page pour se connecter à la BdD H2. si on met les bons identifiants, on a accès à une interface du type phpmyadmin (NB: par défaut l'utilisateur est "sa" et le mot de passe vide)
- si on tape l'URL `localhost:8080/heroes/getbypublicname/baguettor`, on obtient bien le JSON du héros.
- si on tape l'URL `localhost:8080/heroes/getbypublicname/aaa`, on obtient rien => problème car le service ne trouve pas le héros et renvoie null, ce qui fait que le contrôleur renvoie lui-aussi null, donc on réponse HTTP vide.
- si on tape l'URL `localhost:8080/heroes/getbypower?power=pain`, on obtient le JSON du héros qui a pain comme pouvoir.

◀ Précédent (/index.php/s6/objets-connectes/2579-tds)

Suivant ➤ (/index.php/s6/objets-connectes/2583-td-n-6-fonctionnalites-avancees-acces-a-un-api-avec-axios-6)