Création d'une API REST JSON avec Spring-boot

Table of Contents

REST

REST = **RE**presentational **S**tateless **T**ransfer
REST est un type d'architecture qui impose des règles lors de la création de services web.
REST est aujourd'hui principalement utilisé avec **HTTP** et du **JSON**. Attention: REST ne se limite pas à HTTP et au JSON; d'autres protocoles peuvent implémenter une architecture REST et d'autres formats de données sont aussi possibles (XML, YAML, …).
REST, comme son nom l'indique est **stateless**, ce qui signifie que chaque requête est indépendante. Si on prend l'exemple du HTTP, mon GET peut être compris à tout moment par mon serveur, ainsi que mon POST (pour la création). La réponse que le serveur renverra ne sera peut-être pas la même, mais le serveur n'a pas besoin de connaître l'état des précédentes requêtes pour pouvoir répondre aux requêtes suivantes.

HTTP

Protocole permettant d'accéder aux ressources web.
Des clients et des serveurs communiquent afin d'échanger des ressources. Les ressources demandées sont identifiées par une URL (http://csid.iut-paris8.fr/students/12345).
HTTP se base sur un ensemble de verbes (https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods)

- **GET**
- **POST**
- **PUT**
- **DELETE**
- **PATCH**
- HEAD
- OPTIONS
- CONNECT
- TRACE

Dans le cadre de notre cours, nous allons construire une HTTP REST API basée sur JSON.

GET

GET est utilisé pour accéder à une ressource. Par exemple, si l'on veut avoir la liste de tous les employés, on utilisera la méthode GET.

Exemple :

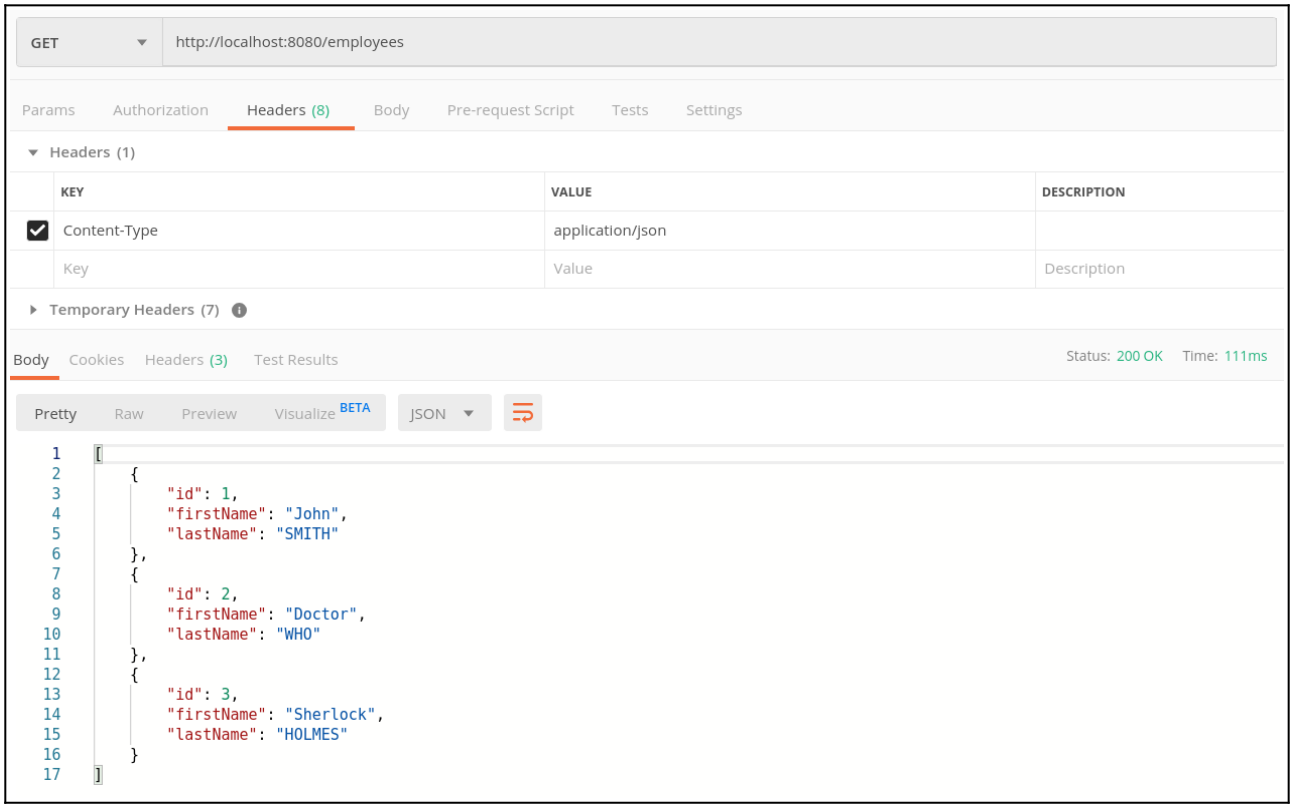


Figure 1: Requête pour récupérer tous les employés

Pour avoir les informations détaillées d'un employé, on utilisera aussi un GET, mais on précisera l'id de celui-ci.

Exemple

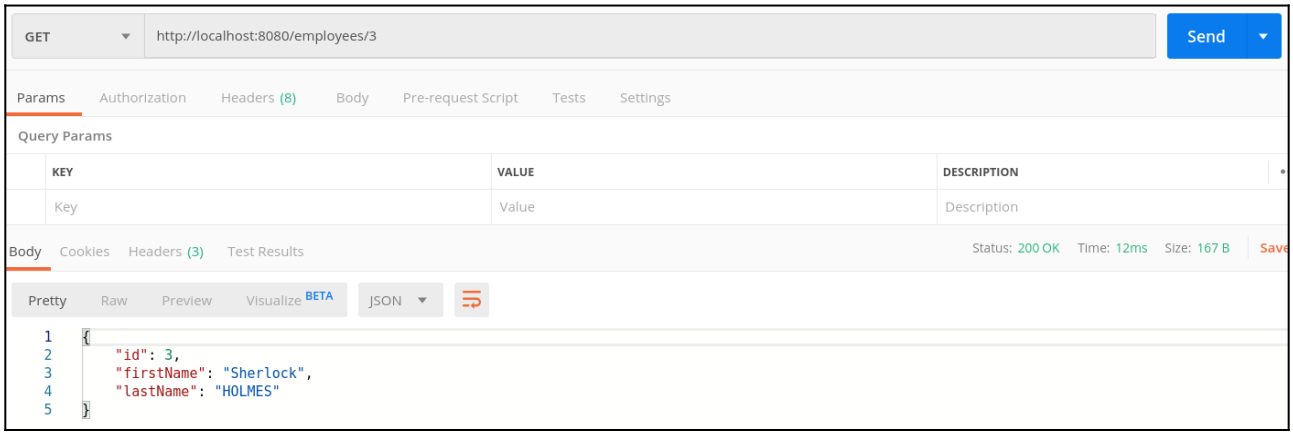


Figure 2: Requête GET pour récupérer un employé

La méthode GET ne doit avoir aucun effet sur les ressources d'un serveur (pas de modification) : elle est uniquement utilisée pour faire de la lecture.

POST

Cette méthode indique au serveur qu'on souhaite lui transmettre une entité. Cette entité est passé dans le body ("corps") de la requête. Ce body va être traitée par le serveur pour, par exemple, faire

une insertion en base de données ou lancer un traitement (par exemple, demander une archive de vos données personnelles)

Exemple :

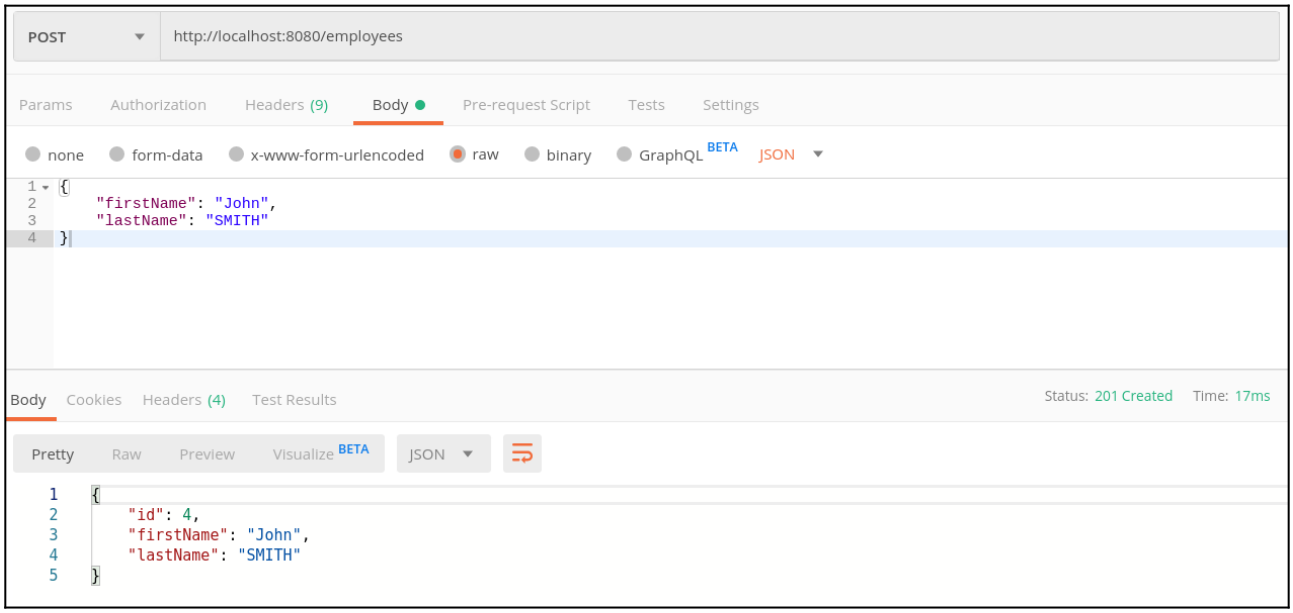


Figure 3: Requête POST pour créer un employé

PUT

Comme pour la méthode *POST*, PUT indique au serveur d'accepter l'entité qui la requête lui présente dans son body, mais à la différence de POST, **PUT indique au serveur quelle est l'adresse de la ressource dans l'URI dans la requête**. Si la ressource existe, PUT modifie celle-ci. Si la ressource n'existe pas, PUT la crée. PUT envoie l'entité **entière** dans le body de sa requête.

PUT est une méthode **idempotente**[1] : peut importe le nombre de fois qu'on exécutera la même requête, l'état du serveur après que la requête soit finie sera le même. Par exemple, dans le cas du PUT, que la ressource existe avant ou non, à la fin de la requête, soit elle existera, soit elle aura été modifiée (ou non), mais elle sera dans l'état que la requête PUT transmettra.

Exemple:

1 Voir https://tools.ietf.org/html/rfc7231#section-4.2.1 pour les safe methods et https://tools.ietf.org/html/rfc7231#section-4.2.2 pour les idempotent methods et https://tools.ietf.org/html/rfc7231#section-8.1.3 pour un récapitulatif

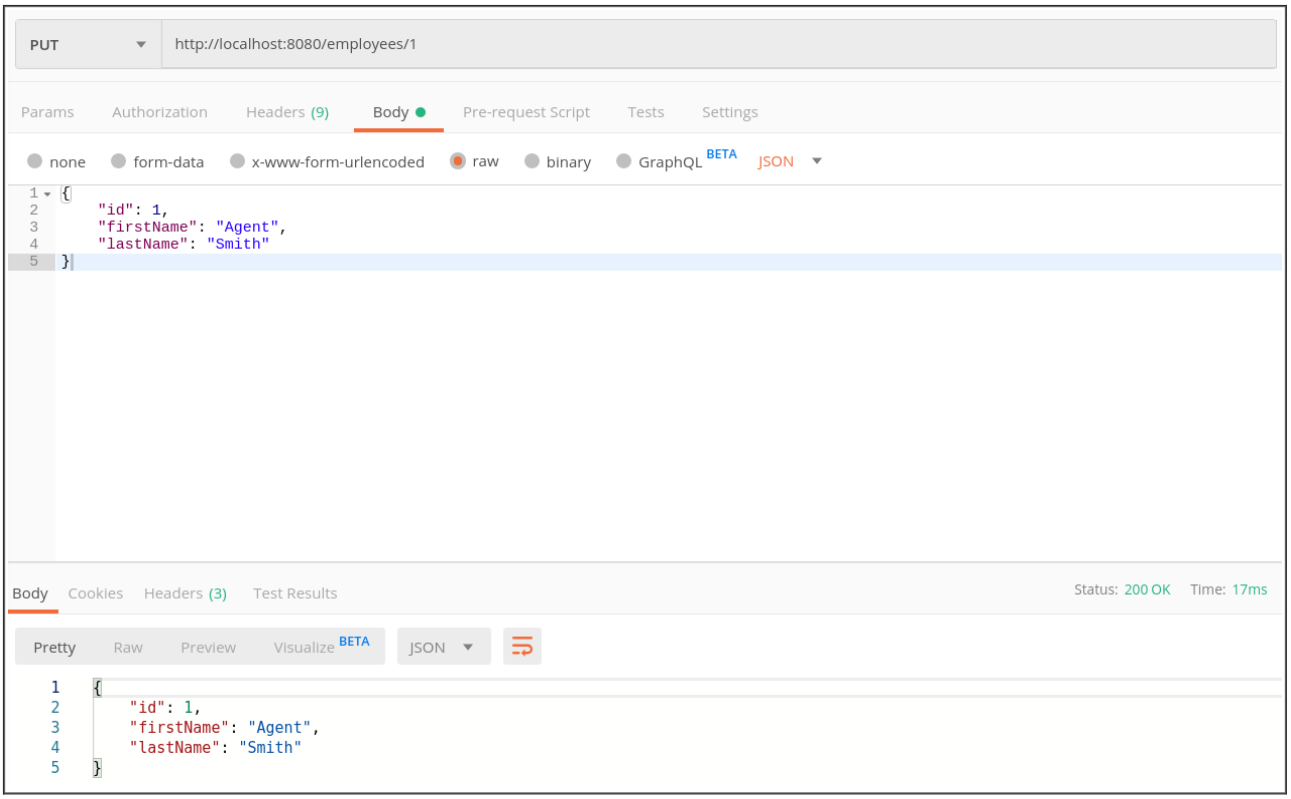


Figure 4: Requête PUT pour modifier un employé

DELETE

Comme son nom l'indique, ***DELETE*** informe le serveur que l'on souhaite supprimer la ressource à l'URI spécifiée.

DELETE est aussi une requête **idempotente**. Que la ressource ait existée ou non avant la requête, à la fin, elle ne sera plus (ou pas) présente sur le serveur.

Exemple :

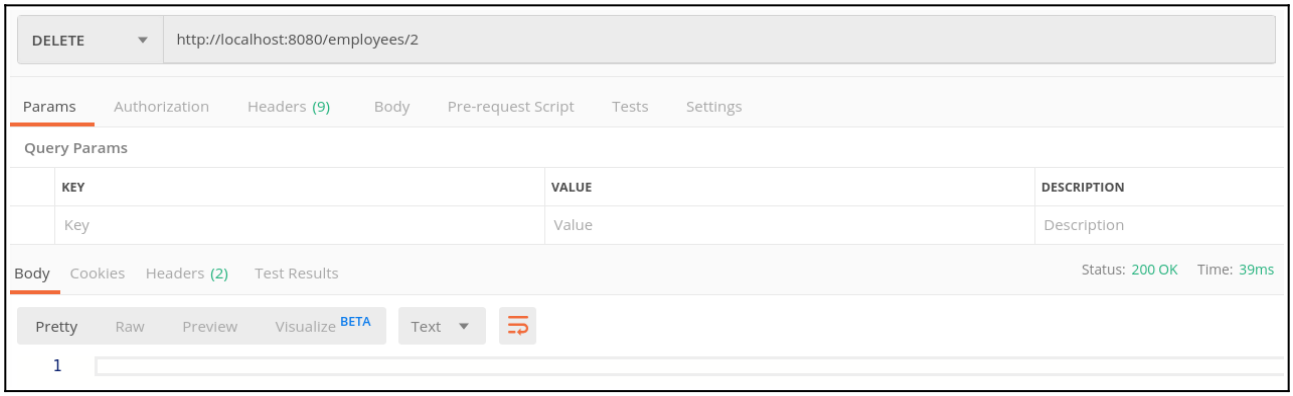
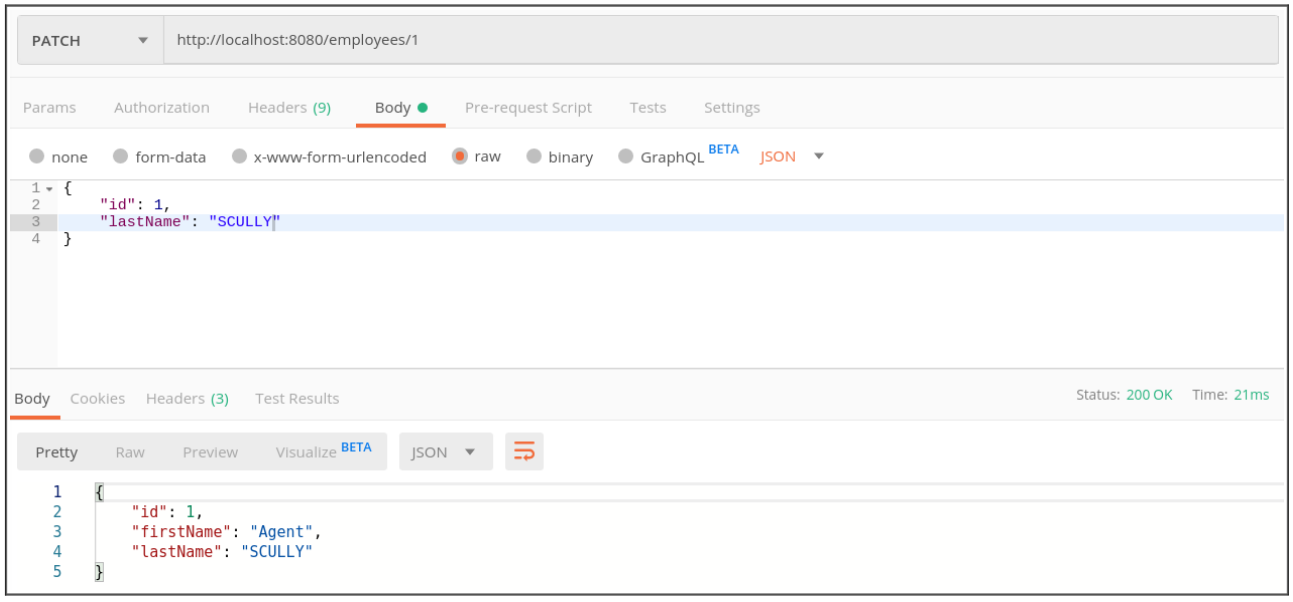


Figure 5: Requête DELETE pour supprimer un employé

PATCH

A venir dans la prochaine version. N'hésitez pas à lire https://williamdurand.fr/2014/02/14/please-do-not-patch-like-an-idiot/ et https://tools.ietf.org/html/rfc7396 pour vous spoilez le contenu du paragraphe.



Spring

Le guide super simple pour avoir une application Spring-boot qui répond à des requêtes HTTP: https://spring.io/guides/gs/rest-service/

L'annotation **@SpringBootApplication** indique à Spring que cette classe est le point d'entrée de notre application. Cette annotation est une association de plusieurs autres annotations:

- **@ComponentScan** : indique à Spring de rechercher et de créer des *beans* à partir des classes annotées. Spring va rechercher les classes annotées *@Configuration*, *@Component*, *@Controller*, … Si on ne précise pas de paramètre à l'annotation, la recherche est effectué dans le package courant et tous les sous packages, mais pas dans les packages précédents.
- **@EnableAutoConfiguration** : autorise Spring-boot à configurer automatiquement certaines fonctionnalités en fonction des classes présentes dans le classpath de l'application. Par exemple, lorsque spring-webmvc est présent, Spring-boot va automatiquement ajouté un bean de type ObjectMapper à son contexte, afin de pouvoir serializer[2] et deserializer[3] les classes vers et depuis du JSON.
- **@Configuration** : indique à Spring que cette classe est apte à créer des beans, que Spring pourra ajouter à son contexte

Injection de dépendances (Dependency Injection)

Avant d'introduire plus de concept sur Spring, il est essentiel de parler de l'Injection de Dépendances (DI), sur lequel Spring se base.

IOC

Le DI est un design pattern qui prend ses racines dans l'**IoC** (Inversion of Control), un autre design pattern.

L'IoC est une technique qui va inverser le contrôle de votre application. L'exemple le plus courant est de prendre celui d'un terminal qui va vous demander de répondre à plusieurs questions. Les exemples suivant sont tirés de https://martinfowler.com/bliki/InversionOfControl.html

Sans IoC :

Dans l'exemple précédent, votre programme a la main sur toute son exécution : il décide quel texte afficher en premier et exécute des méthodes de façon séquentielles.

Imaginons maintenant, que l'on veuille atteindre le même objectif, mais que nous ayons, au lieu d'un terminal, une interface graphique avec des *input text*.

Avec IoC :

2 https://en.wikipedia.org/wiki/Serialization
3 https://en.wikipedia.org/wiki/Unmarshalling

```
require 'tk'
root = TkRoot.new()
name_label = TkLabel.new() {text "What is Your Name?"}
name_label.pack
name = TkEntry.new(root).pack
name.bind("FocusOut") {process_name(name)}
quest_label = TkLabel.new() {text "What is Your Quest?"}
quest_label.pack
quest = TkEntry.new(root).pack
quest.bind("FocusOut") {process_quest(quest)}
Tk.mainloop()
```

Ici, on se sert de l'event "FocusOut" (qui se déclenche lorsque que l'élément graphique perd le focus) pour déclencher l'appel aux méthode "process_name" et "process_quest" : notre programme n'est plus responsable d'une partie de son déroulement, le contrôle est "inversé", c'est l'interface graphique qui a le contrôle.

Plusieurs méthodes existent pour mettre en place l'IoC. Dans cette exemple, on passe par une "*closure*" déclencher après un *event*, mais on peut aussi utiliser le pattern Observer, des *callbacks*, ou la *Dependency Injection*.

Dependency Injection

L'injection de dépendance est un pattern qui laisse un objet extérieur assigner des valeurs aux classes de votre application.

Sans DI:

```
public class DiscPlayer {

    private DiscReader discReader;

    public DiscPlayer() {
        this.discReader = new DvdReader();
    }
}
```

La dépendance est créée par notre classe: cela crée un couplage fort. Si demain, on souhaite lire, non plus un DVD mais un Blu-Ray ou un mini-disc, nous serions obligé de changer l'implémentation de notre classe DiscPlayer.

Avec DI:

```
public class DiscPlayer {

    private DiscReader discReader;

    public DiscPlayer(DiscReader discReader) {
        this.discReader = discReader;
    }
}
```

Ici, DiscPlayer n'est plus responsible de l'instanciation de sa dépendance, c'est de l'IoC. Au lieu de cela, un élément externe, généralement notre Framework, va est responsable de lui fournir sa dépendance, c'est de l'injection de dépendance (DI). Si à l'avenir, on souhaite changer le format de lecture, le framework pourra injecter une autre classe implémentant l'interface DiscReader sans modifier la classe DiscPlayer!

Cette exemple utilise l'injection par constructeur. Il est aussi possible de faire un injection par setter.

```
public class DiscPlayer {

    private DiscReader discReader;

    public void setDiscReader(DiscReader discReader) {
        this.discReader = discReader;
    }
}
```

Voilà, il suffit simplement d'ajouter un setter pour notre champ.

Pour plus d'informations sur la DI, n'hésitez pas à lire https://martinfowler.com/articles/injection.html.

Spring Beans

From Spring documentation

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

Dit autrement, les beans Spring sont l'ensemble des objets qui forment votre application Spring.

Lorsque Spring crée les beans, il lui assigne un portée, un ***scope***. Par défaut, ce scope est **Singleton**. Le scope singleton signifie qu'une seul instance de ce bean sera créé pour toute votre application. Lorsque que Spring injectera un bean portant le même nom, ce sera toujours le même objet qui sera retourné.

D'autres scope existent, voici le tableau de la documentation officiel de Spring (https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes):

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
session	Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
application	Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.
websocket	Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

Créer un bean avec Spring

Créer un bean avec Spring peut être assez simple. La méthode la plus rapide est d'ajouter l'annotation *@Component* sur la classe que l'on souhaite voir devenir un bean Spring. Celle-ci sera automatiquement scannée (si le @ComponentScan est bien configuré) et sera ajoutée au contexte Spring.

Sur le même principe que *@Component*, *@Service*, *@Controller*, *@Repository* permettent de créer des beans : ce sont des stéréotypes (pour connaître la liste des stéréotypes: https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/package-summary.htm). Un stéréotype spécifie un rôle au sein de votre architecture, au final, vos classes annotés par ces annotations (sauf *@Indexed*) seront tous des *@Component* spécialisés.

Il est aussi possible de créer un bean Spring en le déclarant dans une classe annotée de *@Configuration* (qui est un @Component derrière) :

```
@Configuration
public class CsidConfiguration {

   @Bean
   public DiskReader dvdReader() {
      return new DvdReader();
   }
}
```
Exemple 1: Création d'un bean avec @Bean

Spring, lorsqu'il scannera notre classe *CsidConfiguration*, créera le bean *CsidConfiguration* puis il s'occupera d'instancier les beans de cette classe, grâce aux méthodes annotés par *@Bean*.

DI avec Spring (finally!)

Spring est capable d'injecter les beans qu'il gère dans d'autres de ses beans.

Comme vu dans le paragraphe sur la Dependency Injection, il est possible d'injection des beans de différentes manières : par constructeur, par setter et aussi directement sur le champ correspondant à notre dépendance. L'injection de dépendance est réalisé grace à la présence de l'annotation **@Autowired**.

Injection par constructeur :

```
public class DiscPlayer {

    private DiscReader discReader;

    @Autowired
    public DiscPlayer(DiscReader discReader) {
        this.discReader = discReader;
    }
}
```

Injection par setter :

```
public class DiscPlayer {

    private DiscReader discReader;

    @Autowired
    public void setDisReader(DiscReader discReader) {
        this.discReader = discReader;
    }
}
```

Injection par field :

```
public class DiscPlayer {

    @Autowired
    private DiscReader discReader;

}
```

Il est possible d'utiliser les 3 méthodes en même temps dans un même bean, mais ce n'est pas conseillé, pour des raison de lisibilité et d'harmonie du code.

Dans la plupart des cas, l'injection par constructeur est à privilégier. L'injection par constructeur permet rapidement de détecter les problèmes d'architecture du code, comme une classe avec trop de dépendances, alors qu'avec une injection par field, on aura tendance à rajouter des dépendances sans se soucier de leur nombre. De plus, cette méthode d'injection permet de définir ses dépendances comme *final* et d'éviter qu'elles ne soient modifier au runtime.

L'annotation @Autowired n'est pas obligatoire sur votre constructeur si votre classe n'a qu'un seul constructeur.

Injection directement dans une methode annotée @Bean :

Comme vu dans Exemple 1: Création d'un bean avec @Bean, il est possible de définir un bean à partir d'un méthode annotée par l'annotation @Bean. Si on rajoute comme paramètre de cette méthode un autre bean injectera la dépendance demandée directement dans la méthode.

Spring Controllers

Les controlleurs Spring sont les composants qui vont recevoir les requêtes HTTP émises par un client.

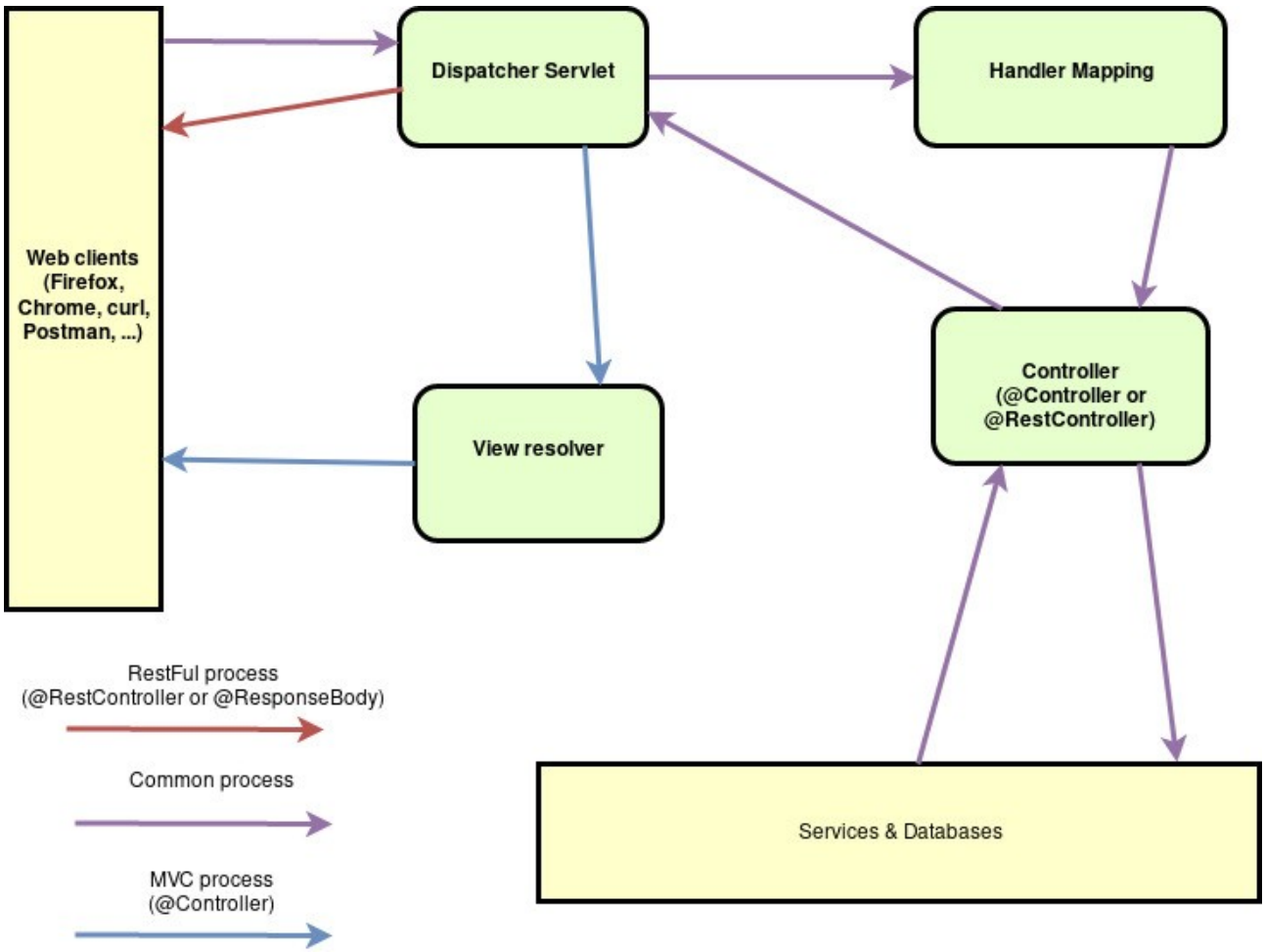


Figure 6: Spring controller flow

Les controllers sont annotés avec *@Controller* ou *@RestController*.

L'annotation @Controller est une spécialisation de @Component et @RestController est une méta-annotation qui rassemble @Controller et @ResponseBody.

@Controller est utilisé avec @RequestMapping. L'alliance de ces deux annotation permet à Spring d'établir une carte des URI auxquelles l'application va répondre : Spring va scanner le projet, et lorsqu'il va trouver une classe annotée avec @Controller (ou @RestController), il va rechercher les annotations @RequestMapping dans celle-ci afin d'enregister les méthodes vers lesquelles il doit redirger les requêtes.

@RequestMapping s'utilise sur les méthodes, mais il est aussi possible d'annoter une classe avec elle : si une classe porte cette annotation, toutes les méthodes de celle-ci se verront préfixer de la valeur présente dans l'annotation de la classe.

Par exemple :

```java
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    @GetMapping("/{id}")
    public EmployeeDto getEmployee(@PathVariable int id) {
        return new EmployeeDto(id);
    }

}
```

Ici, la méthode *getEmployee(...)* aura comme URL ***/employees/{id}***

Dans l'exemple précédent, on peut voir un *@GetMapping* comme annotation pour la méthode getEmployee(…). Cette annotation est un raccourci de l'annotation *@RequestMapping(method = {RequestMethod.GET})*. Elle indique simplement que la méthode ne traitera que les méthodes *GET*. Il est possible pour une méthode de traiter toutes les méthodes HTTP: il suffit de l'annoter avec @RequestMapping sans paramètre!

D'autres annotation sont disponibles afin de pouvoir mapper nos requêtes:

- **@PostMapping**
- **@PutMapping**
- **@DeleteMapping**
- **@PatchMapping**

Ce sont toutes des raccourcis vers les différents verbes HTTP et fonctionnent de la même manière que *@GetMapping*.

Recevoir des paramètres depuis les requêtes:

Depuis le path:

Repartons sur l'image précédente.

On peut voir un *@PathVariable* comme annotation du paramètre "*id*" de notre méthode. Cette annotation indique que le paramètre de notre méthode, "*id*", doit être lié à un des paramètre du path défini dans l'annotation *@GetMapping* (ou *@RequestMapping* ou autre). Ici, le template défini par le *@GetMapping* est "/{id}"; Spring va donc appeler la méthode "*getEmployee*" avec la valeur réelle présente dans l'URL (par exemple 12345).

@PathVariable peut prendre un paramètre "*name*" (ou "*value*"). Si le paramètre *name/value* n'est pas renseigné, Spring va essayer de faire correspondre le nom du paramètre du template avec le nom de la variable de la méthode. Par défault, le path variable est "required" : s'il n'est pas présent dans la requête, Spring retournera un erreur HTTP 400 Bad request. (TODO DANS PARAMS)

Si le type de l'objet annoté par *@PathVariable* est une *Map<String, String>*, alors cette map contiendra toutes les paires clé/valeur des variables du path.

Depuis les paramètres de requêtes:

De la même manière que pour *@PathVariable*, il est possible d'annoter un paramètre avec *@RequestParam*. Les *RequestParam* sont les paramètres de l'URI présents après le symbole '***?***' (https://www.google.com/search?**q=iut+de+montreuil**)

Tout comme *@PathVariable*, *@RequestParam* peut prendre un paramètre "*name*" (ou "*value*"). Dans le cas contraire, Spring tentera de faire correspondre les noms des paramètres de l'URI avec les noms des paramètres de la méthode.

Si le type de l'objet annoté par @RequestMapping est *Map<String, String>* ou *MultiValueMap<String, String>*.

Depuis les headers :

Dans les requêtes HTTP, on trouve aussi des headers. Spring permet d'accéder aux données de ses headers rapidements grace à l'annotation @RequestHeader. Cette annotation fonctionne de la même manière que @RequestParam ou @PathVariable, que ce soit pour les attributs ou la la map de toutes les valeurs.

Spring et les bases de données

Connexion à une base de données

Avec Spring-boot, il est simple de se connecter à une base de données d'à peu près n'importe quel type (MySQL, PostgreSQL, …).

Spring-boot fournit un start permettant de configurer rapidement la connexion à une base de données : **spring-boot-starter-data-jpa**. Ce starter permet de configurer une connexion à une base de données en simplement quelques lignes en base de données ou bien en configurant un bean *DataSource*.

Configuration de la base de données via le fichier application.properties :

Créer ou modifier le fichier ***src/main/resources/application.properties*** pour y rajouter les propriétés suivantes:

```
spring.datasource.url=jdbc:mysql://localhost/database_name
spring.datasource.username=user
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

N'hésitez pas à chercher les valeurs correspondantes à vos base de données.

Configuration de la base de données via un bean Datasource :

```java
@Bean
public DataSource dataSource() {
    return DataSourceBuilder.create()
            .url("jdbc:h2:mem:csid")
            .username("your_username")
            .password("your_password")
            .driverClassName("your.driver.className")
            .build();
}
```

ORM, JPA et Hibernate

// Todo: définir un ORM

JPA (Java Persistence API) est une spécification définissant des interfaces permettant d'effectuer les interactions les plus courantes avec les bases de données. Ces interfaces décrivent comment enregistrer dans une base de données un objet Java (**POJO**[4]).

Afin d'enregistrer un objet en base, JPA se base sur des POJO annotés par ***@Entity*** et délègue à l'**EntityManager** les différentes opérations, que sont l'enregistrement, la modification et la suppression de ligne en base de données.

Hibernate est l'une des implémentations les plus populaires de JPA. C'est donc aussi un ORM. Hibernate permet d'utiliser les fonctionnalités décrites par JPA et plus encore.

JPA défini dans sa norme un langage proche du SQL, le JPQL[5], permettant d'écrire des requêtes qui seront interprétées par votre implémentation de JPA (ici Hibernate). Les requêtes JPQL vont se baser sur vos « *entity* » et donc votre modèle objet. Il est possible d'écrire des SELECT, UPDATE et DELETE and JPQL mais pas de INSERT.

4 Plain Old Java Objet
5 Java Persistence Query Language

Entity

Comme dit précédemment, les entités qui vont être persistées en base sont annotées par @Entity. Une classe annotée @Entity est le reflet d'une des tables de votre base de données. Cette classe va contenir toutes les informations permettant de mapper votre objet à votre base de données. Ces informations sont exprimées sous forme d'annotations, voici un tableau recensant les plus courantes :

Annotation	Utilisation	Commentaire
@Entity	**Sur la classe**. Indique à JPA que cette classe est une entité. Chaque entité représente une table de votre base de données	
@Table	**Sur la classe**. Permet de spécifier le nom de la table avec laquelle il faut mapper votre entité	Si l'annotation n'est pas présente, le nom de la table sera égal à celui de votre classe
@Column	Sur le champ. Spécifie le nom du champ de la base de données avec lequel il faut mapper votre champ	
@Id	Sur le champ. Indique que ce champ est la clé primaire de votre entité	Votre clé primaire peut aussi être composé de 2 champs
@GeneratedValue	Sur le champ.	
@JoinColum	Sur le champ.	
@ManyToOne	Sur le champ. Relation n to 1	
@OneToMany	Sur le champ. Relation 1 to n	
@ManyToMany	Sur le champ. Relation n to n	Une table pourra être automatiquement créer par JPA
@OneToOne	Sur le champ. Relation 1 to 1	
@OrderBy	Sur le champ. Permet de trier les éléments d'une relation @OneToManyer	ASC ou DESC

Une fois les entités définies, il faut les utiliser!

Spring repository

Spring-data-jpa offre des interfaces qui permettent de rapidement requêter les tables de notre base de données à partir de nos entités : ce sont des ***repository***.

Un repository est donc une interface qui étend l'interface Repository<T, ID> de spring-data-jpa. Cette interface requiert 2 paramètres , T et ID. T désigne le type (classe) de l'entité pour lequel vous déclare le repository et ID le type de la clé primaire de cette même entité. Les repository sont annotés par @Repository, dont nous avons parlé plus haut dans le cours. Ce sont aussi des beans Spring.

```java
@Repository
public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {

}
```

Figure 7: Un repository d'EmployeeEntity

Dans l'exemple précédent, on utilise un JpaRepository, mais d'autres type de repository existent et chacun à ses propres spécificités : CrudRepository, PagingAndSortingRepository et JpaRepository (pour ne citer qu'eux).

En étendant ces interfaces, il va être possible de faire la plupart des opérations courantes sans écrire la moindre requête SQL. Avec un CrudRepository, on pourra sélection un objet avec son Id, sélectionner tous les éléments d'une table, les supprimer et les mettre à jour. Le PagingAndSortingRepository va, quant à lui, apporter des capacité de tri. JpaRepository est très similaire au CrudRepository, mais on peut noter quelques différences qui peuvent être importantes. Premièremnet, les méthodes *findAll* retournent une liste et non plus un iterable, ce qui peut être plus pratique pour boucler dessus. Deuxièmement, et plus important, JpaRepository ajoute la méthode ***getOne***. Cette méthode est très proche de la méthode findById de CrudRepository, à la différence qu'elle retourne une instance de l'entité et non pas un *Optional*. Mais la plus grosse différence avec le *findById* est que, si *getOne* ne retourne aucune entité, c'est-à-dire que la requête SQL avec l'ID spécifié ne retourne aucune ligne, alors *getOne* va lancer une exception ! Dès lors, cette méthode n'est à utiliser que lorsque l'on est sûr que l'id passé en paramètre existe (je nuance tout de suite mon propos : lorsque vous recevez l'ID en tant que paramètre d'une URL, */employee/1*, on peut supposer que le client est certain que l'ID 1 existe, et l'on peut alors utiliser getOne. Si au final, l'ID n'existe pas, on catchera l'exception et renverra une réponse 404 NOT FOUND à notre client).

Si les méthodes fournies par les interfaces ne vous suffisent pas, il est toujours possible de requêter votre base de données de plusieurs autres façons.

A partir des nom de méthodes :

Query creation :

En respectant un format particulier, il est possible d'indiquer à spring-data-jpa quelle requête vous souhaitez exécuter en ne déclarant qu'une méthode dans votre repository. Un exemple étant plus parlant que de longue phrases ; supposons que vous souhaitez lister tous les employées dont le nom est « *Quentin* », il vous suffira alors de déclarer la méthode suivante :

```java
@Repository
public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {

    List<EmployeeEntity> findAllByFirstName(String firstName);
```

On peut bien sûr ajouter plusieurs conditions ou bien trier les la liste qui sera retourner, ou même les deux en même temps :

```java
@Repository
public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {

    List<EmployeeEntity> findAllByFirstName(String firstName);

    List<EmployeeEntity> findAllByFirstNameAndLastNameOrderByLastNameDesc(String firstName, String lastName);


}
```

*Figure 8: select * from employee where first_name = xxxxxx and last_name = xxxxx order by last_name desc*

Vous trouverez la liste de mots clés ici : https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

Named Query :

À partir de l'annotation @NamedQuery, vous pouvez déclarer des requêtes sur vos entités. L'annotation prend en paramètre la requête en JPQL et le nom de la méthode à laquelle elle sera associée.

```java
@Entity
@NamedQuery(query = "select e from  EmployeeEntity e order by e.firstName desc", name = "EmployeeEntity.findAllOrderByFirstNameDesc")
@Table(name = "employee")
public class EmployeeEntity {
```

Sur l'entité, on défini notre requête en JPQL en lui spécifiant un nom : le nom est composé du nom de l'entité suivi du nom de la méthode que l'on appelera depuis le repository (voir image suivante).

```java
@Repository
public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {

    List<EmployeeEntity> findAllOrderByFirstNameDesc();

}
```

Je recommande d'utiliser cette annotation avec parcimonie. En effet, en l'utilisant trop, vous vous retrouverez avec beaucoup de requêtes déclarées sur votre entité et la maintenance peut devenir laborieuse.

Vous pouvez aussi utiliser l'annoation ***@NamedQueries***, qui prend un liste de *@NamedQuery*, pour rassemblez vos requêtes.

N'hésitez pas à aussi regardez du côté de ***@NamedNativeQuery***, pour déclarer des requêtes natives (sans passer par JPQL).

Query :

Une autre annotation peut être utilisée pour déclarer de nouvelles requêtes : ***@Query***. Contrairement à *@NamedQuery*, cette annotation se place directement sur la méthode de l'interface. En mettant l'attribut «native» à «*true*», il est possible d'écrire directement du SQL.

```
@Repository
public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {

    @Query("select e from  EmployeeEntity e order by e.firstName asc")
    List<EmployeeEntity> findAllOrderByFirstNameDesc();

}
```

Je recommande l'utilisation de cette annotation à l'instar de *@NamedQuery*. De cette manière, toutes vos méthodes et requêtes se trouvent au même endroit, et non plus séparé entre votre entité et votre repository.

Sources

https://martinfowler.com/articles/injection.html

https://martinfowler.com/bliki/InversionOfControl.html

https://en.wikipedia.org/wiki/Inversion_of_control

https://en.wikipedia.org/wiki/Representational_state_transfer

https://en.wikipedia.org/wiki/Stateless_protocol

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

https://tools.ietf.org/html/rfc2616

https://en.wikipedia.org/wiki/Serialization

https://en.wikipedia.org/wiki/Unmarshalling

https://www.baeldung.com/spring-controllers (le schéma)

https://docs.spring.io/spring/docs/current/javadoc-api/

https://stackoverflow.com/a/18580299/1766602

https://www.tutorialspoint.com/jpa/jpa_architecture.htm