iP Net Institute of Technology — Next Generation IT Professionals

SE 352 - Développement d'Applications de Classe Entreprise avec la Programmation Orientée Objet

Ferdinand BATANA, DBA & Web Developer(OCAJP, MCSE SQL SERVER)

QU'EST CE QUE SPRING ?

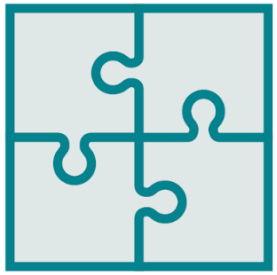
1. Le Spring Framework est une espèce de méta framework. Son but est d'apporter une aide aux développeurs d'applications et d'API et d'en accélérer le développement. Très utilisé dans la communauté Java, il s'agit d'une alternative architecturale au modèle que proposait la plateforme J2EE.
2. Le Spring Framework, quant à lui, propose des applications incluant déjà les services nécessaires. Grâce à cette stratégie, l'application se transforme en une sorte de mini-conteneur, appelé aussi conteneur léger, qui ne nécessite donc plus le recours à un serveur

QU'EST CE QUE SPRING ?

Spring Security

Spring Data

Spring Core



Spring Cloud

Spring Boot

QU'EST CE QUE SPRING BOOT ?

1. SpringBoot n'est pas destiné à remplacer Spring, mais à rendre le travail avec lui plus rapide et plus facile
2. Le développement avec SpringBoot présente plusieurs avantages :
   - Une gestion des dépendances plus simple.
   - Une auto-configuration par défaut
   - Un serveur web intégré
   - Des métriques et contrôles de santé de l'application.
   - Une configuration avancée externalisée.

SPRING INITIALIZR

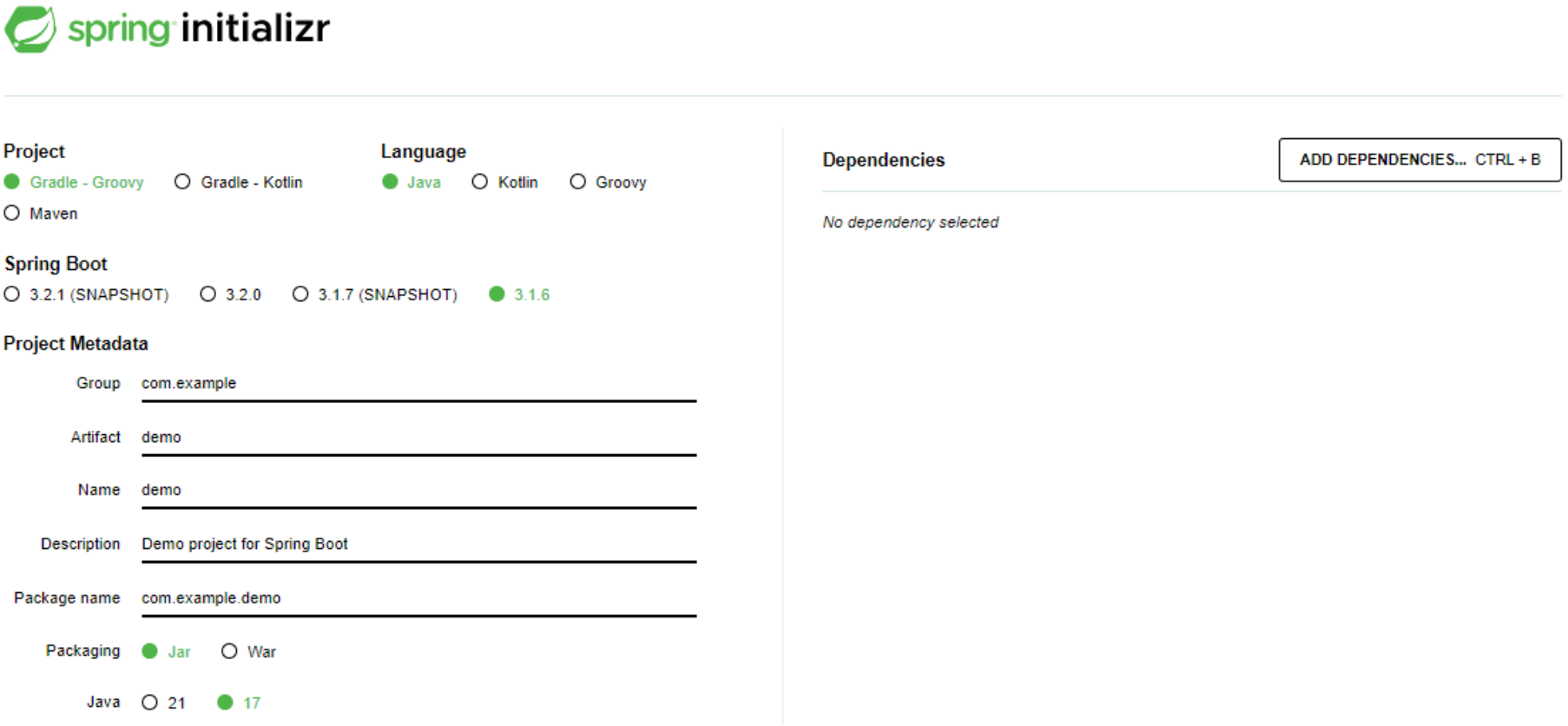
Création d'une application d'entreprise

Les applications d'entreprise modernes fournissent des service restes et exploitent au moins une base de données

SPRING INITIALIZR

Pour créer facilement une application java qui tient compte de tous ces librairies/composants, nous allons utiliser spring initializr:
https://start.spring.io/

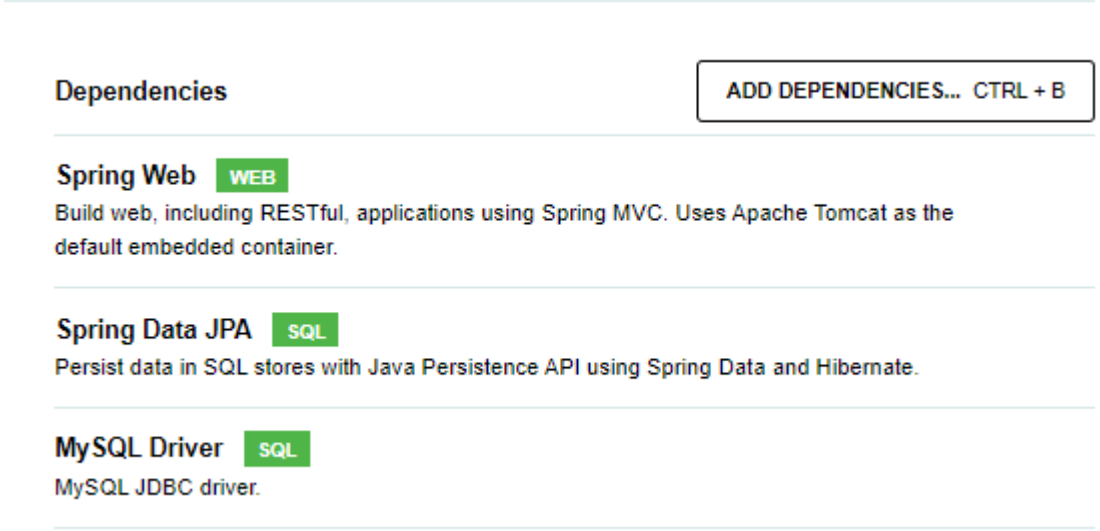
SPRING INITIALIZR



SPRING INITIALIZR

Dépendances:

- Spring web
- Spring Data JPA
- MySQL Driver



Cliquer sur GENERATE

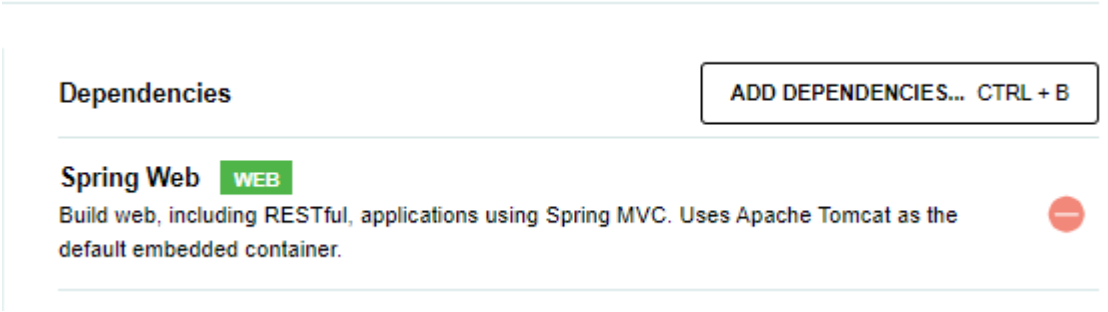
Le projet peut être démarré en ligne de commande.

Rendez-vous dans le repertoire du projet pour taper la commande: mvnw spring-boot:run dans l'invite de commande.

SPRING INITIALIZR

Dépendances:

- Spring web



Cliquer sur GENERATE

Le projet peut être démarré en ligne de commande.

Rendez-vous dans le repertoire du projet pour taper la commande: mvnw spring-boot:run dans l'invite de commande.

SPRING INITIALIZR

Votre application doit normalement démarrer sur le port 8080

```
2023-12-10T22:14:24.958Z  INFO 14564 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
Tomcat started on port 8080 (http) with context path ''
2023-12-10T22:14:24.967Z  INFO 14564 --- [           main] com.cours.hello.HelloApplication
```

Fichier html

hello › src › main › resources › static

AUTO-CONFIGURATION SPRING

Changement des données du fichier application.properties

Server.port=

Server.servlet.context-path

AUTO-CONFIGURATION SPRING

@SpringBootApplication est un raccourcis vers

@Configuration

@ComponentScan

@EnableAutoConfiguration

AUTO-CONFIGURATION SPRING

Changer de serveur d'application

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
                <!– Exclure la dépendance vers tomcat -->
                <exclusion>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
        </exclusions>
</dependency>
<!-- Utilisons Jetty -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

AUTO-CONFIGURATION SPRING

Problème: La modification des données static nécessite le redemarrage du serveur.

Configurer l'accès aux fichiers static(html, css, js, images etc…)

Spring.resources.static-locations = file:path

CONTROLLEUR FRONTAL: Controller

- Modification de la valeur par défaut (/) des paths dans le fichier applicaition.properties: spring.mvc.servlet.path =

CONTROLLEUR FRONTAL: RestController et Controller

- Différence entre RestController Controller:
- L'annotation @Controller est là depuis le premier jour et elle est utilisée pour marquer une classe en tant que contrôleur Web pour traiter les requêtes HTTP et renvoyer un nom de vue, qui est ensuite résolu par un résolveur de vue pour générer la vue HTML finale.
- L'annotation @RestController a été ajoutée dans les versions ultérieures de Spring, comme Spring 3.4, pour augmenter la prise en charge du développement de l'API REST. Dans le cas de l'API REST, au lieu de renvoyer du HTML, vous préférerez probablement renvoyer du JSON ou du XML car votre client n'est plus un humain mais une machine.

CONTROLLEUR FRONTAL: RestController et Controller

- Mise en place du controller

1. Créer un package web (ex: cours.formation.se352.web)
2. Créer une classe dont le nom suit le forma suivant: classeRestController
3. Mettre une annotation @RestController

Chaque requête http est destinée à un controleur. Mais comment reconnaitre le controleur adapter: C'est l'url de la requête qui sera considéré et cette url va mener à une méthode destinée à répondre en effectuant le traitement souhaité.

CONTROLLEUR FRONTAL: RestController et Controller

- Si vous souhaitez renvoyer un JSON ou un XML à partir d'un contrôleur (@Controller) Spring MVC, vous devez ajouter l'annotation @ReseponseBody à chacune des méthodes du contrôleur et cela semble excessif lors de l'implémentation des API REST à l'aide de Spring et Spring Boot.

```
no usages
@Controller
public class TestRestController {
    no usages
    @RequestMapping(value="/bonjour", method = RequestMethod.GET)
    @ResponseBody
    public String bonjour(){
        return "Bonjour ";
    }
}
```

CONTROLLEUR FRONTAL: RestController et Controller

- @RestController : plus besoin d'ajouter RequestBody

```
no usages
@RestController
public class TestRestController {
    no usages
    @RequestMapping(value="/bonjour", method = RequestMethod.GET)
    public String bonjour(){
        return "Bonjour ";
    }
}
```

Conclusion: @RestController = @Controller + @ResponseBody

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: cette annotation permet de faire un mapping entre une requette http et une méthode d'un controlleur
  - Il possède divers attributs permettant de faire le filtrage par URL, méthode HTTP, paramètres de requête, en-têtes et types de médias.
- Il existe également des variantes de raccourci spécifiques à la méthode HTTP @RequestMapping:
  - @GetMapping = @RequestMapping(method = RequestMethod.GET)
  - @PostMapping = @RequestMapping(method = RequestMethod.POST)
  - @PutMapping (Mise à jour d'un objet entire) = = @RequestMapping(method = RequestMethod.PUT)
  - @DeleteMapping = @RequestMapping(method = RequestMethod.DELETE)
  - @PatchMapping (Mise à jour d'une partie d'un objet) = @RequestMapping(method = RequestMethod.PATCH)

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: URL et méthode

```
no usages
@RestController
public class TestRestController {
    no usages
    @RequestMapping(value="/bonjour", method = RequestMethod.GET)
    public String bonjour(){
        return "Bonjour ";
    }
}
```

```
C:\Users\Alou>curl -i http://localhost:8080/bonjour
```

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: Consumes et Produces

Le mapping des types de médias produit par une méthode de contrôleur mérite une attention particulière.

Nous pouvons mapper une requête en fonction de son attribut Accept de l'en-tête :

```
@RequestMapping(value="/bonjour", method = RequestMethod.GET, headers = "Accept=application/json")
public String bonjour(){
    return "Bonjour ";
}
```

Définir comme suit est moins regoureux : la correspondance se fait par « Contenir » et non « Egal », donc une requête telle que la suivante serait toujours mappée correctement:

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: Consumes et Produces

```
C:\Users\Alou>curl -H curl -H "Accept:application/json,text/html"   http://localhost:8080/bonjour
```

- À partir de Spring 3.1, l'annotation @RequestMapping possède désormais les attributs Produces et Consumes, spécifiquement à cet effet :

```
@RequestMapping(value="/bonjour", method=RequestMethod.GET, produces = "application/json")
public String bonjour( ){
    return "Bonjour " ;
}
```

- La méthode va retourner un objet de type json

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: Consumes et Produces

```
@RequestMapping(value="/bonjour", method = RequestMethod.GET, consumes ="application/json")
public String bonjour(){
    return "Bonjour ";
}
```

La méthode accepte uniquement des objet json

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: Les variables

Une variable @PathVariable

```
@RequestMapping(value={"/bonjour/{nom}","/bonjour"}, method=RequestMethod.GET)
public String test(@PathVariable("nom") String  nomUtilisateur){

    return "Bonjour "+nomUtilisateur ;
}
```

Test

```
C:\Users\Alou>curl  http://localhost:8080/bonjour/OOZONS
Bonjour OOZONS
```

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: Les variables

Une variable @PathVariable: Si la variable et le paramètre de la méthode ont la même écriture, plus la peine de mettre la variable dans les paranthèses de @PathVariabl

```
@RequestMapping(value="/bonjour/{nom}", method = RequestMethod.GET)
public String bonjour(@PathVariable String nom){
    return "Bonjour "+nom;
}
```

Test

```
C:\Users\Alou>curl  http://localhost:8080/bonjour/OOZONS
Bonjour OOZONS
```

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: Les variables facultatives

Une variable peut être définie facultative

```
@PostMapping(value={"/bonjour/{nom}","/bonjour"}, produces = "application/json")
public String test(@PathVariable(required = false) String  nom){
    String resultat = "";
    if(nom != null)
        resultat = nom;
    else
        resultat = "Absent";
    return resultat ;
}
```

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: Les variables

Plusieurs variables

```
@RequestMapping(value="/bonjour/{nom}/{age}", method = RequestMethod.GET)
public String bonjour(@PathVariable String nom, int age){
    return "Bonjour "+nom  + " Vous êtes agé de "+ age;
}
```

Test

```
C:\Users\Alou>curl  http://localhost:8080/bonjour/ferdinand/20
Bonjour ferdinand Vous êtes agé de 20
```

CONTROLLEUR FRONTAL: Les annotations

- @RequestMapping: Les paramètres

```
@RequestMapping(value="/bonjour", method = RequestMethod.GET)
public String bonjour(@RequestParam("nom")  String nom){
    return "Bonjour "+nom ;
}
```

Test

CONTROLLEUR FRONTAL: Les annotations

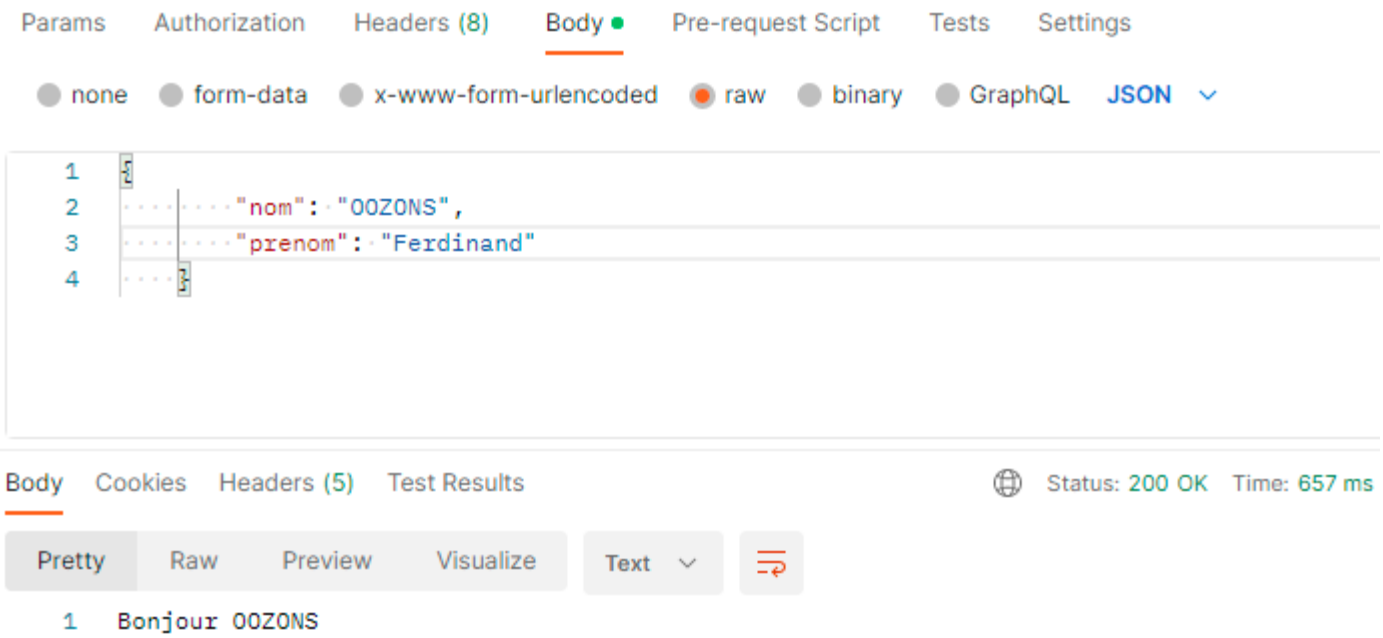
- @RequestBody

En termes simples, l'annotation @RequestBody fait le mapping d'un objet HttpRequest vers un objet java, permettant la désérialisation automatique

```
@RequestMapping(value="/bonjour", method = RequestMethod.GET)
public String bonjour(@RequestBody Auteur auteur){
    return "Bonjour "+auteur.getNom() ;
}
```

CONTROLLEUR FRONTAL: Les annotations

- @RequestBody



CONTROLLEUR FRONTAL: Les annotations

- @ResponseBody

L'annotation @ResponseBody indique à un contrôleur que l'objet renvoyé est automatiquement sérialisé en JSON et renvoyé dans l'objet HttpResponse.

```
@PostMapping(value="/bonjour")
@ResponseBody
public Auteur bonjour(@RequestBody Auteur  auteur){
    return auteur ;
}
```

NB: L'utilisation de @ResponseBody n'est plus obligatoire si vous utilisez @RestController

CONTROLLEUR FRONTAL: Les annotations

- @ResponseEntity<?>

ResponseEntity représente l'intégralité de la réponse HTTP : code d'état, en-têtes et corps. De ce fait, nous pouvons l'utiliser pour configurer entièrement la réponse HTTP

- ResponseBody: Changement du statut de la requête http

```
@ResponseStatus(HttpStatus.CREATED )
@ResponseBody
@RequestMapping(value = "/bonjour", method = RequestMethod.GET)
public String bonjour(){
    return "Bonjour ";
}
```

CONTROLLEUR FRONTAL: Les annotations

- @ResponseEntity<?>
- ResponseEntity: Changement du statut de la requête http

```
@RequestMapping(value = "/bonjour", method = RequestMethod.GET)
public ResponseEntity<String> bonjour(){
    return ResponseEntity.status(HttpStatus.OK).body("Bonjour");
}
```

Ou

```
@RequestMapping(value = "/bonjour", method = RequestMethod.GET)
public ResponseEntity<String> bonjour(){
    return new ResponseEntity<>(
            body: "Bonjour", HttpStatus.OK
    );
```

CONTROLLEUR FRONTAL: Les annotations

- Définition du type de contenu

Lorsque nous utilisons l'annotation @ResponseBody, nous sommes toujours en mesure de définir explicitement le type de contenu renvoyé par notre méthode.

```
@PostMapping(value="/bonjour", produces = MediaType.APPLICATION_JSON_VALUE)
public Auteur bonjour(@RequestBody Auteur  auteur){
    return auteur ;
}
```

Ou

```
@PostMapping(value="/bonjour", produces = "application/json")
public Auteur bonjour(@RequestBody Auteur  auteur){
    return auteur ;
}
```

@Service

- Les services

Nous marquons les classes avec @Service pour indiquer qu'ils contiennent une logique métier rien d'autres.

- Imaginons que nous aimerions dire bonjour M. ou bonjour Mme en fonction du sexe d'un auteur. Comment allez-vous procéder ?
- Solution: Les services
- Un service est une classe java qui contient une logique métier.

@Service

- Les services
- Créer un sous package com.example.demo.service du package com.example.demo.
- Créer la classe MonService dans ce package
- Mettre l'annotation @Service sur le nom de la classe
- Ajouter l'attribut sexe dans la classe auteur.
- Définir une méthode permettant de dire bonjour M. ou Bonjour Mme en fonction du sexe dans la classe MonService .

@Service

- Les services

```
@Service
public class MonService {
    1 usage
    public String salutation(Auteur auteur){
        String resultat = "Bonjour ";
        if(auteur.getSexe()=='F'){
            resultat += "M.";
        }else if(auteur.getSexe()=='F'){
            resultat +="Mme";
        }else{
            resultat +="X";
        }
        return resultat;
    }
}
```

@Service

- Les services – injection de dépendances

L'injection de dépendance est un mécanisme simple à mettre en œuvre dans le cadre de la programmation objet et qui permet de diminuer le couplage entre deux ou plusieurs objets.

- Dépendance: Lorsqu'une classe (A) a besoin d'une autre classe (B) pour fonctionner, on dit que A a une dépendance vers B et que B est une dépendance pour A
- Nous allons utiliser l'annotation @Autowired pour injecter une dépendance dans une classe.
- Nous pouvons aussi faire l'injection par le constructeur(Meilleure méthode)

@Service

- Les services – injection de dépendances

Injectons notre service MonService dans le controller TestRestController

@Autowired

```
@Autowired
private MonService monService;
    no usages
    @RequestMapping(value="/bonjour", method=RequestMethod.GET, produces = "application/json")
    public String bonjour(@RequestBody Auteur auteur ){
        return this.monService.salutation(auteur) ;
    }
```

@Service

- Les services – injection de dépendances

Injectons notre service MonService dans le controller TestRestController

Par le constructeur

```
private MonService monService;
    no usages
    public TestRestController(MonService monService) {
        this.monService = monService;
    }
    no usages
    @RequestMapping(value="/bonjour", method=RequestMethod.GET, produces = "application/json")
    public ResponseEntity<String> bonjour(@RequestBody Auteur auteur ){
        return new ResponseEntity<>(
            this.monService.salutation(auteur), HttpStatus.OK
        );
    }
```

@Service

- Les services – injection de dépendances

@Qualifier, @Primary etc…

Mapping relationnel avec JPA

- ORM - JPA

ORM, ou Object-Relational Mapping, est une technique de programmation qui permet de faire correspondre les objets d'un langage de programmation orienté objet (comme Java, C#, Python, etc.) avec les structures de données d'une base de données relationnelle.

JPA (Java Persistence API) est une API Java standard qui fournit une interface de programmation pour la gestion de la persistance des données. JPA simplifie le développement d'applications Java en permettant aux développeurs d'interagir avec des bases de données relationnelles à l'aide d'objets Java plutôt qu'en écrivant directement des requêtes SQL.

Mapping relationnel avec JPA

- Les relations avec JPA
- Une des fonctionnalités majeures des ORM est de gérer les relations entre objets comme des relations entre tables dans un modèle de base de données relationnelle. JPA définit des modèles de relation qui peuvent être déclarés par annotation.*
- Exemple d'annotation: @OneToOne, @ManyToOne, @OneToMany, @ManyToMany.

Mapping relationnel avec JPA

- Prérequis : Pour pouvoir utiliser l'API JPA dans un projet vous devez ajouter la dépendance suivante:

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

- Pour connecter votre application à MYSQL, vous devez ajouter la dépendance suivante

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

Mapping relationnel avec JPA

- Entités : Une entité, déclarée par l'annotation **@Entity** définit une classe Java comme étant persistante et donc associée à une table dans la base de données. Cette classe doit être implantée selon les normes des beans: propriétés déclarées private, les accesseurs et les mutateurs, nommés selon les conventions habituelles.
- Par défaut, une entité est associée à la table portant le même nom que la classe. Il est possible d'indiquer le nom de la table par une annotation **@Table**

```
@Entity
@Table(name = "Auteur") // Optionnel au cas ou le nom de la table et de la classe sont identique
public class Auteur {
    2 usages
    private String nom;
```

Mapping relationnel avec JPA

- Identifiant d'une entité : Toute entité doit avoir une propriété déclarée comme étant l'identifiant de la ligne dans la table correspondante.
- L'identifiant est indiqué avec l'annotation **@Id**. Pour produire automatiquement les valeurs d'identifiant, on ajoute une annotation **@GeneratedValue** avec un paramètre **Strategy**. Voici deux possibilités pour ce paramètre:
  - Strategy = GenerationType.AUTO: Hibernate produit lui-même la valeur des identifiants grâce à une table hibernate_sequence
  - Strategy = GenerationType.IDENTITY. Hibernate s'appuie alors sur le mécanisme propre au SGBD pour la production de l'identifiant.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;
```

Mapping relationnel avec JPA

- Les colonnes: Par défaut, toutes les propriétés non-statiques d'une classe-entité sont considérées comme devant être stockées dans la base. Pour indiquer des options (et aussi pour des raisons de clarté à la lecture du code) on utilise le plus souvent l'annotation **@Column** (Cette annotation n'est pas obligatoire)

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;
2 usages
private String nom;
2 usages
private String prenom;
2 usages
private String matricule;
2 usages
private char sexe;
```

Mapping relationnel avec JPA

- Voici les principaux attributs pour @Column:
- **name** indique le nom de la colonne dans la table;
- **length** indique la taille maximale de la valeur de la propriété;
- **nullable** (avec les valeurs false ou true) indique si la colonne accepte ou non des valeurs à NULL (au sens « base de données » du terme: une valeur à NULL est une absence de valeur);
- **unique** indique que la valeur de la colonne est unique.

Mapping relationnel avec JPA

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;
2 usages
@Column(name = "NOM_AUTEUR", nullable = false, length = 50)
private String nom;
2 usages
@Column(name = "PRENOM_AUTEUR", nullable = false, length = 50)
private String prenom;
```

Mapping relationnel avec JPA

- La relation 1:1 (one to one) : L'annotation @OneToOne définit une relation 1:1 entre deux entités:

```java
@Entity
@Table(name = "Auteur")
public class Auteur {
    no usages
    @OneToOne
    private Photo photo;
```

```java
@Entity
public class Photo {
    no usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
}
```

- L'annotation **@OneToOne** implique que la table Auteur contient une colonne qui est une clé étrangère contenant la clé d'une photo. Par défaut, JPA s'attend à ce que cette colonne se nomme PHOTO_ID, mais il est possible de changer ce nom grâce à l'annotation **@JoinColumn** :

Mapping relationnel avec JPA

- La relation 1:1 (one to one) :

```
@Entity
@Table(name = "Auteur")
public class Auteur {
    no usages
    @JoinColumn(name = "FK_AUTEUR_PHOTO")
    @OneToOne
    private Photo photo;
```

Mapping relationnel avec JPA

- La relation n:1 (many to one) : L'annotation @ManyToOne définit une relation n:1 entre deux entités:

```
@Entity
public class Livre {

    no usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    no usages
    @ManyToOne
    private Auteur auteur;
}
```

Mapping relationnel avec JPA

- La relation n:1 (many to one) : L'annotation @ManyToOne définit une relation n:1 entre deux entités:
- L'annotation @ManyToOne implique que la table Livre contient une colonne qui est une clé étrangère contenant la clé d'un auteur. Par défaut, JPA s'attend à ce que cette colonne se nomme AUTEUR_ID, mais il est possible de changer ce nom grâce à l'annotation @JoinColumn.