

# Java - Spring

---

# Sommaire

- Introduction
- Première application
- Lombok
- Spring MVC Rest Services
- Thymeleaf
- Mockito
- Gérer les données
- Validation des données
- Routes avancées
- Relations de Données
- Spring Rest Template
- Spring Security Basics
- Spring Authorization Server

# Introduction

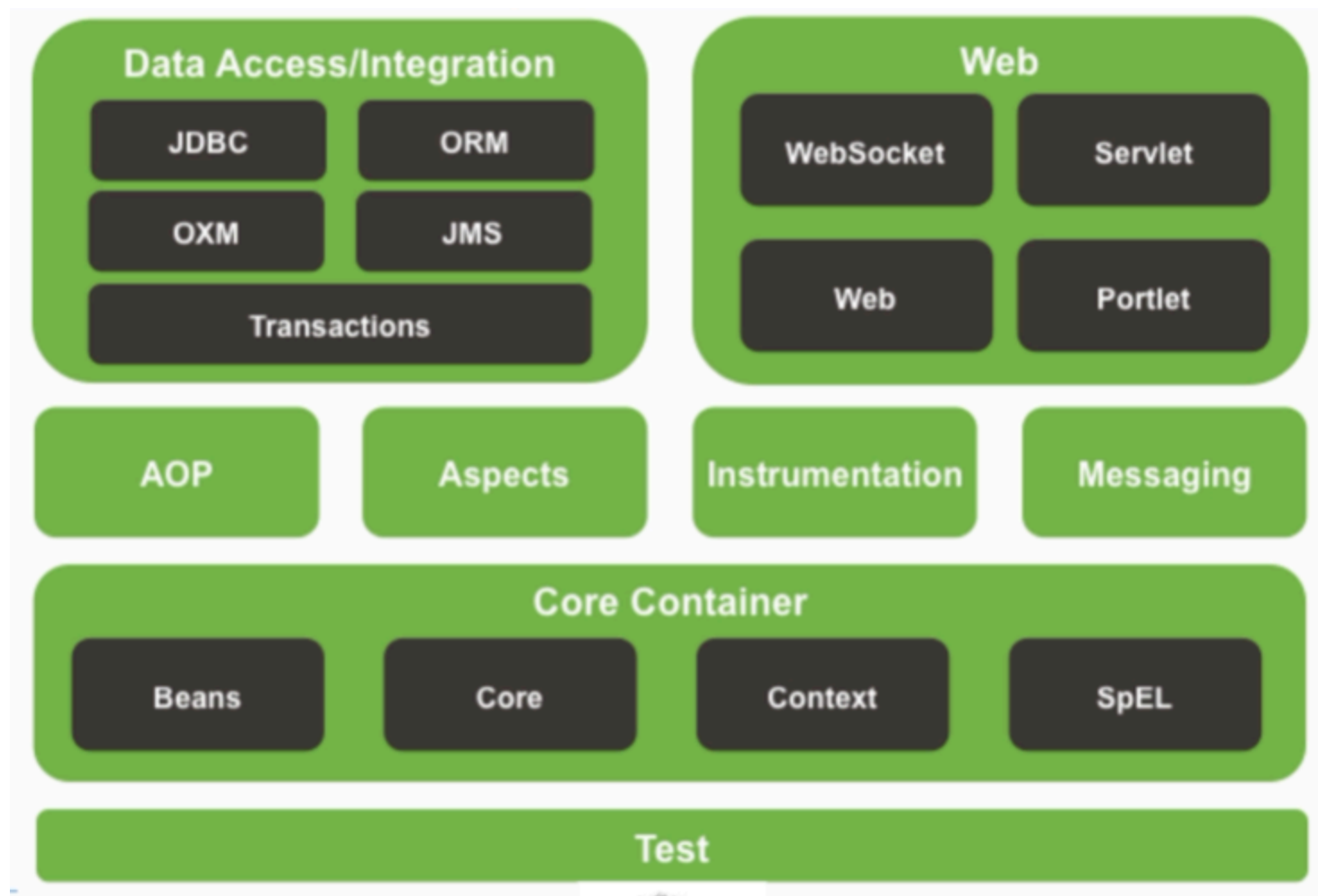
## Un peu d'Histoire

**Spring Framework** est l'un des framework les plus utilisés dans l'écosystème Java. Il est très utilisé dans les domaines de la finance et du retailing.

Ses capacités sont très variées et il est à la fois utile pour travailler des données, réaliser des applications front-end ou back-end, ce de façon monolithique ou sous forme de micro-services. Son origine date de **2003** où il fut introduit par **Rod Johnson**.

Il est important de noter cependant une différence entre **Spring Framework** et **Spring Boot**.

# Spring Framework



# Spring Boot

- Il comprend les dépendances de base pour la majorité des projets réalisables avec Spring
- Possède une configuration automatisée des éléments de type classe trouvés dans le **CLASSPATH**, comme par exemple une base de données H2
- Gestion des fichiers de configs et variables d'environnement
- Logging
- Amélioration globale de l'expérience de développement

# Quelques projets Spring

- **Spring Data:** Collection de projets pour gérer la persistance de données en SQL / NoSQL
- **Spring Cloud:** Outils pour les systèmes distribués
- **Spring Security:** Authentification et Autorisation
- **Spring Session:** Applications web distribuées usant de sessions
- **Spring Integration:** Patterns d'intégration d'entreprise
- **Spring Batch:** Batch processing
- **Spring State Machine:** Gestion d'état des machines (Open source)

# Le modèle MVC

Le modèle MVC est un **Design Pattern** reposant sur le principe de la séparation en trois domaines de notre application:

- **Model:** Les entités de données utilisées dans le cadre de notre application
- **View:** Les templates pré-remplies servant de base à l'interface web qui seront ensuite peuplées de données.
- **Controller:** Les classes servant à la récupération de la requête, son traitement, le peuplage des vues par des modèles et l'envoi de la réponse finale au client.



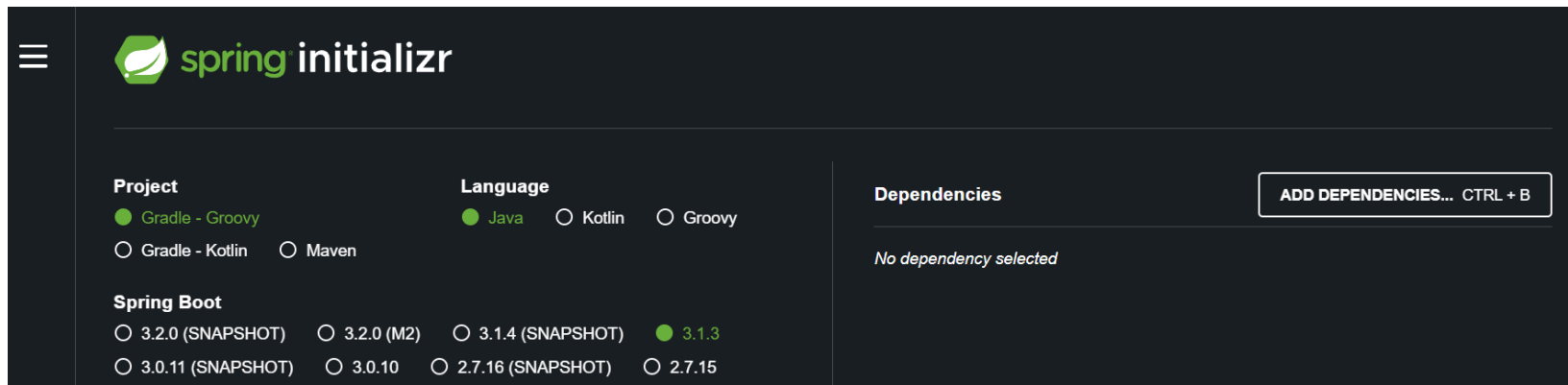
# Les verbes HTTP

- **GET:** Le client demande une ressource au serveur
- **HEAD:** Le client demande les méta-données au serveur
- **POST:** Le client poste des informations au serveur.
- **PUT:** Le client demande de stocker ou modifier une entité.
- **PATCH:** Le client demande une modification d'entité.
- **DELETE:** Le client demande une suppression d'entité au serveur
- **TRACE:** Le client demande un écho de la requête au serveur
- **OPTIONS:** Le client demande les verbes supportés à cet endpoint.

# Première application

# Spring Initializer

Lorsque l'on veut réaliser une application Spring, il est généralement plus simple de pré-configurer notre batterie de dépendances via l'utilisation de [Spring Initializer](#).



The screenshot shows the Spring Initializer web interface. It features a dark theme with a sidebar on the left containing a hamburger menu icon and the 'spring initializr' logo. The main content area is divided into three sections: 'Project', 'Language', and 'Dependencies'. The 'Project' section has radio buttons for 'Gradle - Groovy' (selected), 'Gradle - Kotlin', and 'Maven'. The 'Language' section has radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'. The 'Spring Boot' section has radio buttons for various versions: '3.2.0 (SNAPSHOT)', '3.2.0 (M2)', '3.1.4 (SNAPSHOT)', '3.1.3' (selected), '3.0.11 (SNAPSHOT)', '3.0.10', '2.7.16 (SNAPSHOT)', and '2.7.15'. The 'Dependencies' section has a button labeled 'ADD DEPENDENCIES... CTRL + B' and the text 'No dependency selected'.

L'outil est également disponible dans l'interface d'IntelliJ Ultimate si l'on crée un projet Spring.

# L'Injection de Dépendance

```
@Controller
public class BaseController {
    private MyServiceInterface myService;

    public BaseController(MyServiceInterface myService) {
        this.myService = myService;
    }
}
```

La dépendance est créée de par le framework et se verra envoyée à la demande dans le constructeur à condition d'utiliser les annotations.

# Les principes **SOLID**

Datant de Mars 1995, les principes **SOLID** furent relayés par **Robert Martin** et sont au coeur des notions d'injection de dépendance et de l'inversion de contrôle:

- **S**: Single Responsibility Principle
- **O**: Open Closed Principle
- **L**: Liskov Substitution Principle
- **I**: Interface Segregation Principle
- **D**: Dependency Inversion Principle

# Le Contexte Spring

```
@SpringBootApplication
public class SpringFramework01DependencyInjectionApplication {

    public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication.run(SpringFramework01DependencyInjectionApplication.class, args);

        MyController controller = ctx.getBean(MyController.class);

        System.out.println("I'm in the main method");
        System.out.println(controller.sayHello());
    }
}
```

Le contexte Spring est à la clé de la récupération des éléments. En fonction des annotations que l'on donne à nos classes, on peut définir le type de **Bean** qui seront instanciés et utilisés dans le cadre de notre application.

# @Controller

Cette annotation est utilisée à Spring pour construire un élément destiné à gérer les requêtes HTTP entrantes et pour répondre, entre autre, via des vues (dans le cadre d'une application Web) ou des objets JSON

```
@Controller
public class MyController {

    @RequestMapping(value = "/books", method = RequestMethod.GET)
    public String getBooks(Model model) {
    }
}
```

## @Autowired

Cette annotation est utilisée pour indiquer à Spring de réaliser une injection de dépendance pour une propriété ou un setter.

```
private MyService myService;

@Autowired
public void setMyService(MyService myService) {
    this.myService = myService;
}
```

Dans le cadre d'une injection par constructeur, celle-ci est automatique. Au moment où Spring instancie le contrôleur, il remarquera la dépendance et la fournira de lui-même.



## @Primary

Dans le cas où l'on a plusieurs implémentation d'une même interface, il est possible de dire à Spring laquelle est la configuration par défaut. Pour ce faire, il suffit d'ajouter l'annotation **@Primary**:

```
@Primary
@Service
public class MyService {
    // ...
}
```

# @Qualifier

```
@Service("serviceName")  
public class MyServiceImpl {  
    // ...  
}
```

```
@Controller  
public class MyController {  
    @Qualifier("serviceName")  
    @Autowired  
    MyService myService;  
  
    // ...  
}
```

# Les Profils Spring

Il est possible également de définir des profils pour mettre en commun toute une série de Beans:

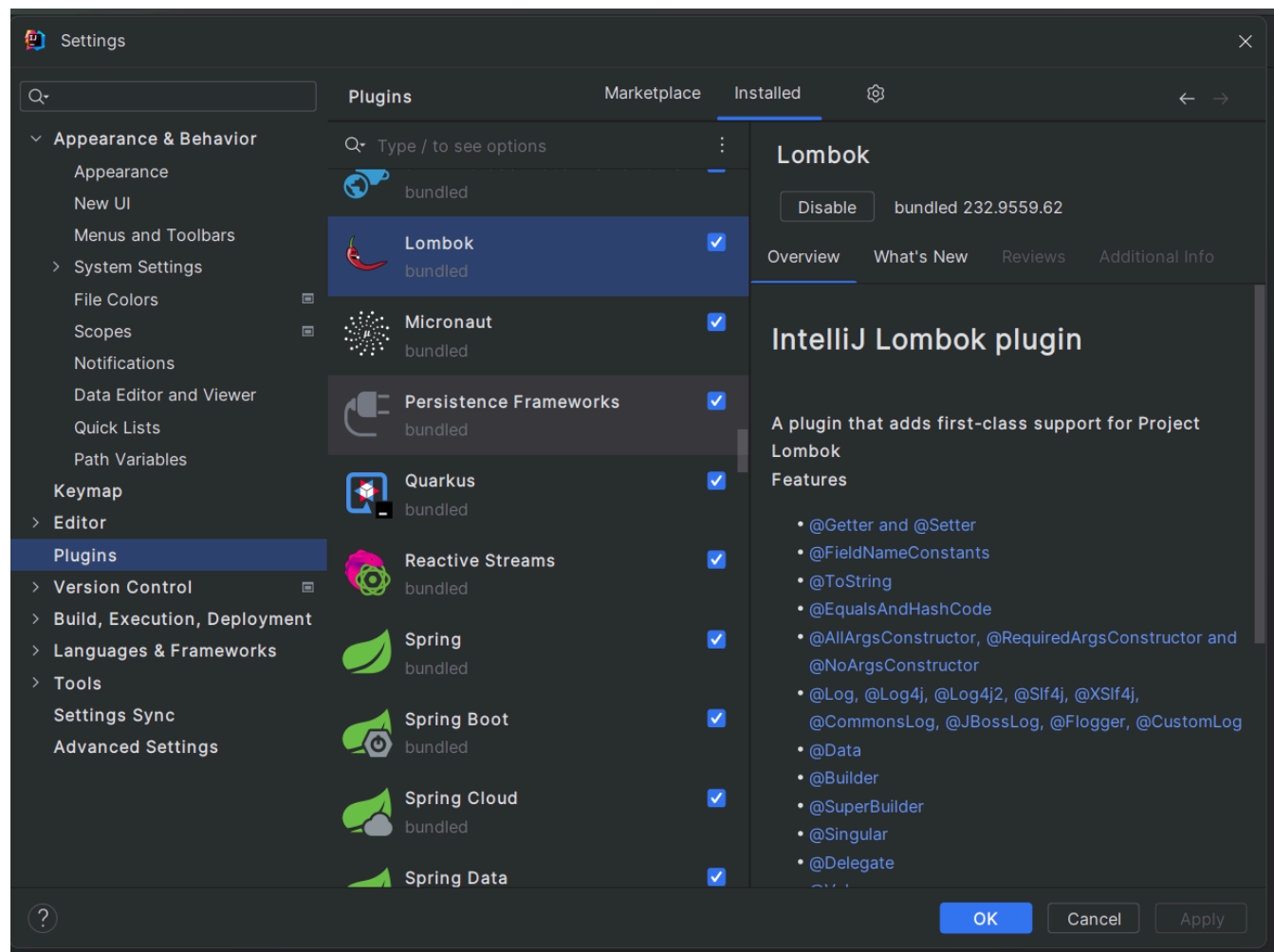
```
@Profile("EN")
@Service("serviceName")
public class MyServiceEnglish {
    // ...
}
```

Une fois le(s) profil(s) ajouté(s) à nos éléments, on peut définir lequel est actif dans le fichier **application.properties**:

```
spring.profiles.active=EN
spring.profiles.include=FR, BE
```

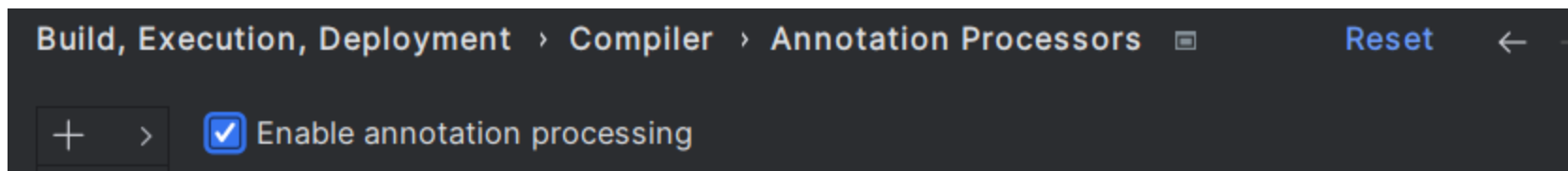
# Lombok

# Configuration d'IntelliJ IDEA



## A quoi ça sert ?

Lombok est un outil servant à la génération de code lors de la compilation. L'objectif de cet outil est de réduire la taille du code Java via des annotations amenant à la génération d'éléments tels que les **Getters / Setters, constructeurs, etc.**



Pour que cela fonctionne, il va nous falloir configurer notre environnement de développement pour qu'il prenne en compte les **annotations de traitement.**

# Les annotations

- **@Getter**: Génère les getters
- **@Setter**: Génère les setters
- **@ToString**: Génère la méthode dédiée (paramètres possibles)
- **@EqualsAndHashCode**: Génère les deux méthodes dédiées (paramétrage possible pour exclusion de propriétés)
- **@NoArgsConstructor**: Constructeur sans paramètres
- **@RequiredArgsConstructor**: Constructeur pour les champs non finaux ou nullables

## Les annotations (suite)

- **@Data**: Rassemble les annotations **@Getter**, **@Setter**, **@ToString**, **@EqualsAndHashCode** et **@RequiredArgsConstructor**
- **@Value**: Version immuable de **@Data**
- **@NonNull**: Empêche la nullabilité
- **@Builder**: Permet l'implémentation du **builder pattern**
- **@Synchronized**: Permet d'implémenter la synchronisation Java
- **@SneakyThrows**: Permet la levée d'exceptions dans déclarer la clause **throws** dans la méthode
- **@Log** / **@Slf4j**: Permet l'accès à un logger





# Spring MVC Rest Services

Spring Rest MVC est utilisé dans le cadre de la réalisation d'API. Une API est en général un service fourni par un serveur qu'il est possible d'interroger via le protocole HTTP et ses différents verbes. Pour qu'une API soit dite Rest, il nous faut implémenter les verbes principaux qui sont **GET, DELETE, POST** et **PUT / PATCH**.

# @RestController

Si l'on retourne un élément, le contrôleur va le transformer automatiquement en **JSON** pour être manipulable par le client Rest tel que **Postman**.

```
@RestController
public class ElementController {

    @GetMapping("/api/v1/element")
    public List<ClassName> getElements() {
        // ...
    }
}
```

# Utiliser les paramètres de la route

Si l'on le veut, il est possible de nous servir d'éléments de notre route que l'on peut extraire pour nous en servir au sein de notre code. Pour ce faire, notre route doit être exprimée de la sorte:

```
@RequestMapping("/api/v1/element")
@RestController
public class ElementController {

    @GetMapping("/{elementId}")
    public ClassName getElementById(@PathVariable("elementId") Long id) {
        // ...
    }
}
```

# Récupérer des objets

Il est possible dans Spring de réceptionner directement les objets envoyés par le client sous la forme d'un JSON. Cet objet sera alors transformé en élément Java manipulable dans notre code de contrôleur.

```
@PostMapping
public ResponseEntity handlePost(@RequestBody Classname obj) {
    Classname savedObj = repository.save(obj);

    return new ResponseEntity(HttpStatus.CREATED);
}
```

# Gérer les Headers de la réponse

Si l'on le veut, il nous est également possible de modifier les Headers de notre réponse via le constructeur de réponse :

```
@PostMapping
public ResponseEntity handlePost(Classname obj) {
    Classname savedObj = repository.save(obj);

    HttpHeaders headers = new HttpHeaders();
    headers.add("Location", "/api/v1/element/" + savedObj.getId().toString());

    return new ResponseEntity(headers, HttpStatus.CREATED);
}
```

# Spring Boot DevTools

En ajoutant cette dépendance, il devient possible de monitorer les classes afin d'analyser leur changement et de ne pas avoir à relancer à chaque fois notre application en cas de modification des classes et/ou de retour de focus de la fenêtre d'IntelliJ.

Pour ce faire, il nous faut éditer la configuration de notre application dans l'interface d'IntelliJ.

# Thymeleaf



# La création de Vues

Dans une application Web, il va nous falloir en général une interface agréable et pratique pour manipuler notre application. Dans le cadre de Spring, une dépendance, **Thymeleaf**, va grandement nous faciliter la chose.

Pour utiliser Thymeleaf dans votre HTML, tout ce qu'il suffit de faire, c'est de l'inclure via les namespaces XML:

```
<html xmlns:th="http://www.thymeleaf.org">
```

Une fois fait, il vous sera possible d'utiliser les fonction de Thymeleaf via l'utilisation d'attributs HTML de type **th:nomFonctionnalité**.

## Retourner une vue

Pour envoyer des pages HTML à notre navigateur, il faut que les contrôleurs Spring retourne des view. Via l'utilisation de Thymeleaf, ceci n'est pas très complexe. Il suffit de faire un retour de chaîne de caractère correspondant au chemin, sans le suffixe d'extension, vers notre page depuis le dossier templates (où doivent se trouver nos pages, d'ailleurs):

```
@GetMapping  
public String getPage() {  
    return "chemin/vers/page";  
}
```

## Le passage d'attributs à notre vue

Dans notre contrôleur Spring, il est possible d'injecter des attributs dans notre vue pour nous en servir par la suite. Par exemple, on voudrait pouvoir afficher les valeurs d'un objet Java ou rendre conditionnel le rendu de notre page. Pour cela, on va demander à Spring d'injecter un Model que l'on va manipuler. Ce dernier sera transmis à la vue automatiquement

```
@GetMapping
public String getPersonInfos(Model model) {
    PersonDTO person = personService.getRandomPerson();

    model.addAttribute("person", person);
}
```

# Affichage de valeurs

Pour ensuite récupérer nos éléments envoyés via les attributs du modèle dans notre vue, il suffit d'utiliser une syntaxe du type **`${nomAttribut}`** au sein de nos attributs Thymeleaf présent dans les balises HTML. Par exemple, pour afficher la valeur d'un message, on peut utiliser **`th:text`**:

```
<p>  
  La valeur du message est <b th:text="${message}"></b>  
</p>
```

# Affichage de collections

Si désormais on envoie à notre vue une collection d'objets, alors on peut les parcourir via une boucle de type `forEach()` dans notre template via l'attribut **th:each**:

```
<table>
  <tr th:each="p : ${persons}">
    <td th:text="${p.firstName}"></td>
    <td th:text="${p.lastName}"></td>
  </tr>
</table>
```

# Rendu conditionnel

Dans le cas où l'on souhaite provoquer ou non le rendu d'éléments dans notre template HTML, on peut utiliser encore une fois des attributs Thymeleaf dédiés à cela. On peut alors utiliser **th:if** pour afficher en cas d'évaluation à **True**, ou **th:unless** si l'on préfère travailler avec des valeurs **False**:

```
<p th:if="{persons.isEmpty()}">No person yet!</p>
<table th:unless="{persons.isEmpty()}">
  <tr th:each="p : {persons}">
    <td th:text="{p.firstName}"></td>
    <td th:text="{p.lastName}"></td>
  </tr>
</table>
```

# Attributs natifs conditionnels

Certains attributs dans notre code HTML devraient idéalement n'être présent dans le résultat final qu'en cas d'évaluation d'un booléen à **True**. Par exemple, dans un formulaire, on aimerai pouvoir rendre conditionnel le placement de l'attribut `readonly` sur un champ de type `<input>`. Pour ce faire, on peut utiliser encore une fois les pre-processor de Thymeleaf:

```
<input type="text" th:readonly="${isReadOnly}">
```

Si l'on veut contrôler le contenu de nos balises de façon plus personnalisées, on peut également utiliser **th:attr** ou **th:attrappend**.

## Les formulaires (1/2)

Pour gérer l'envoi de formulaires avec Thymeleaf, on peut utiliser deux autres symtaxes que sont `@{/url}` et `*{propertyName}`. Ces deux nouvelles syntaxes vont servir à respecter le routing relatif au chemin du contexte de notre artéfact ainsi qu'à peupler les champs de notre formulaire. Il ne faudra cependant pas oublier d'envoyer à notre template un attribut correspondant au type d'objet que l'on veut manipuler:

```
@GetMapping("/add")
public String addElementForm(Model model) {
    model.addAttribute("person", new Person());

    return "person/add";
}
```



## Les formulaires (2/2)

Au niveau de notre template HTML, il nous faut maintenant créer un formulaire exploitant l'attribut envoyé via **th:object** permettant ensuite d'extraire les propriétés de l'objet via **\*{propertyName}**:

```
<form action="#" th:action="@{/persons/add}" th:object="${person}" method="post">
  <div>
    <label for="firstName">Firstname: </label>
    <input type="text" id="firstName" th:field="*{firstName}">
  </div>
  <button>Submit</button>
</form>
```

# Les liens personnalisés

Si l'on veut par la suite faire en sorte de nous déplacer dans l'application Web, nous aurons besoin de liens de type `<a>`. Pour pouvoir peupler les chemins à atteindre, on va encore un fois utiliser la syntaxe offerte par thymeleaf, qui peut être:

```
<a th:href="@{/url}">
```

```
<a th:href="@{/url/with/optional/params(paramA='value')}">
```

```
<a th:href="@{/dynamic/url/{id}(id='value')}">
```

```
<a th:href="@{/dynamic/url/{id}(id=${elementId})}">
```

# Mockito

## Le Test et les Exceptions

# Tester nos applications Spring MVC

Si l'on souhaite réaliser des tests de nos applications Spring MVC, il est possible de le faire via **MockMVC**. En effet, pour tester les contrôleurs (et leurs dépendances), prendre en compte les requêtes et les réponses, **JUnit** n'est pas suffisant.

MockMVC peut être lancé avec ou sans contexte Spring (Un vrai test unitaire ne devrait pas utiliser le contexte).

# Les types de Mocks

- **Dummy**: Un objet utilisé pour la compilation du code
- **Fake**: Un objet possédant une implémentation mais n'étant pas prêt pour la production
- **Stub**: Un objet dont les méthodes ont des réponses pré-définies
- **Mock**: Un objet dont les méthodes ont des réponses pré-définies et des attentes vis à vis de leur exécution. Peut lever des exceptions si une invocation inattendue a lieu.
- **Spy**: Permet la création de Mock autour d'objets Java standard.

# Configurer MockMvc

```
@WebMvcTest(MyController.class)
class MyControllerTest {

    @Autowired
    MockMvc mockMvc;

    @MockBean
    MyService myService;

    @Test
    void getElementById() {
        mockMvc.perform(get("/api/v1/element/" + UUID.randomUUID()))
            .accept(MediaType.APPLICATION_JSON)
            .andExpect(status().isOk());
    }
}
```

# Retourner des données

```
MyServiceImpl elementServiceImpl = new MyServiceImpl();

@Test
void getElementById() {
    ClassName testElement = elementServiceImpl.listElements().get(0);

    given(elementService.getElementById(any(UUID.class))).willReturn(testElement);

    mockMvc.perform(get("/api/v1/element/" + UUID.randomUUID()))
        .accept(MediaType.APPLICATION_JSON)
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON));
}
```

# JSON Matchers

Si l'on veut tester l'objet de retour, il est possible d'utiliser une notation particulière, les **JSON Matchers**:

```
@Test
void getElementById() {
    // ...

    mockMvc.perform(get("/api/v1/element/" + testElement.getId()))
        .accept(MediaType.APPLICATION_JSON)
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.id", is(testElement.getId().toString())))
        .andExpect(jsonPath("$.firstName", is(testElement.getFirstName())));
}
```



# Jackson

Dans le cas où l'on veut tester les méthodes POST, PUT, PATCH, il sera nécessaire d'envoyer un objet JSON. Pour ce faire, on peut passer par un **objectMapper** :

```
@Autowired
ObjectMapper objectMapper;

@Test
void addElement() {
    ClassName newElement = ...

    given(elementService.saveElement(any(ClassName.class))).willReturn(elementService.listElements().get(1));

    mockMvc.perform(post("/api/v1/element"))
        .accept(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(newElement))
        .andExpect(status().isCreated())
        .andExpect(header().exists("Location"));
}
```

# Gérer les exceptions

Les exceptions et leur association aux code de status est géré automatiquement dans Spring.

Le **DefaultHandlerExceptionHandlerResolver** va lever les exceptions en fonction du problème, mais des annotations existent:

- **@ExceptionHandler**: A placer sur les contrôleurs pour gérer tels types d'exceptions.
- **@ControllerAdvice**: Pour implémenter le gestionnaire d'exception global
- **@ResponseStatus**: Pour donner le code status lié à une classe d'exception

# Utiliser **ExceptionHandler**

Dans le cas où une ligne d'instruction dans un contrôleur provoque une levée d'exception, il est possible de rediriger la chose vers une méthode dédiée à la gestion de cette exception. Pour cela, on passe par **ExceptionHandler** en lui indiquant quelle exception va être gérée par la méthode suivante:

```
@ExceptionHandler(NotFoundException.class)
public ResponseEntity handleNotFoundException() {
    // ...

    return ResponseEntity.notFound().build();
}
```

## @ControllerAdvice

Si l'on souhaite faire en sorte de gérer les exception au niveau de l'application et non indépendamment par contrôleur, alors il est possible d'utiliser l'annotation **@ControllerAdvice**:

```
@ControllerAdvice
public class ExceptionController {
    @ExceptionHandler(NotFoundException.class)
    public ResponseEntity handleNotFoundException() {
        // ...

        return ResponseEntity.notFound().build();
    }
}
```

## @ResponseStatus

Si l'on veut affecter à nos exceptions personnelles le status code qu'elles déclenchent, il est possible de les précéder par l'annotation **@ResponseStatus**:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Value not found")
public class NotFoundException extends RuntimeException {
    // ...
}
```

# Les Données dans Spring MVC

# Spring Data JPA

Dans le cadre de l'utilisation de Spring Data JPA, il est important d'utiliser les bonnes annotations dans nos classes Java:

- **@Entity**: Pour préciser que la classe est une entité de données
- **@Id**: Pour informer JPA de quelle propriété de la classe se retrouvera utilisée comme clé primaire
- **@Version**: Pour éviter les soucis de modification de l'entité par un autre thread

Concernant les IDs, il est utile d'informer Hibernate de quelle stratégie de génération sera utilisée. Pour ce faire, on utilise **@GeneratedValue** ou **@GenericGenerator**

# Les Repositories

Pour manipuler les données, il nous est possible d'utiliser l'interface générique **JpaRepository**:

```
public interface MyElementRepository extends JpaRepository<ClassName, UUID> {  
  
    // Pas besoin de code ici  
}
```

Le reste du travail se verra être réalisé automatiquement par JPA et nous fournira l'ensemble des méthodes utiles à notre CRUD.



# Le Testing avec JPA

Pour éviter de manipuler nos données dans le cadre de tests, il est possible d'utiliser des annotations dédiées telles que **@DataJpaTest**:

```
@DataJpaTest
class MyRepositoryTest {
    @Autowired
    MyRepository myRepository;

    @Test
    void myTest() {
        ClassName savedElement = myRepository.save(ClassName.builder().firstName("John").lastName("DUPONT").build());

        assertThat(savedElement).isNotNull();
        assertThat(savedElement.getId()).isNotNull();
    }
}
```

# MapStruct

Il est utile, lorsque l'on manipule des données, de posséder des classes structurelles permettant de rapidement affecter les valeurs et de les récupérer au niveau du serveur.

Pour ce faire, nous passons généralement par des **Data Transfer Objects** qui, contrairement aux classes POJOs de base, ne possèdent pas de fonctionnement propre et peuvent ou non posséder les jointures avec les autres classes (pour offrir plus de flexibilité aux transferts de données).

Pour aider à la sérialisation, **MapStruct** va offrir des implémentation de la même façon que Lombok.

# Notre premier Mapper

Un mappeur est un élément qui va transformer une classe en une autre, ce dans les deux sens. Nous allons nous en servir pour avoir facilement une conversion de nos POJOs en DTOs et vice-versa.

```
@Mapper
public interface PersonMapper {

    Person personDtoToPerson(PersonDTO dto);

    PersonDTO personToPersonDto(Person person);
}
```

La génération automatique de code va se servir du **builder pattern** pour réaliser le mapping.

## Les Services JPA

Il nous faut désormais faire en sorte d'utiliser en coordination nos mappeurs et nos repositories. Il peut être également utile d'utiliser le type **Optional** dans le cadre de notre code métier pour permettre la gestion des recherche d'éléments non existants par l'utilisateur (mauvais ID par exemple):

```
public Optional<PersonDTP> getPersonById(UUID id) {  
    // ...  
}
```

# Implémentation du service

Une implémentation possible des méthodes permettant le CRUD des éléments serait par exemple:

```
@Service
public class PersonServiceJPA implements PersonService {
    private final PersonRepository personRepository;
    private final PersonMapper personMapper;

    @Override
    public List<PersonDTO> listPersons() {
        return personRepository.findAll()
            .stream()
            .map(personMapper::personToPersonDto)
            .collect(Collectors.toList());
    }
}
```

# @Transactional / @Rollback

Lorsque l'on réalise des tests unitaires dans une application possédant une couche de données, il est important d'éviter les réelles manipulations de données:

```
@Rollback
@Transactional
@Test
void testEmptyList() {
    personRepository.deleteAll();

    List<PersonDTO> dtos = personRepository.listPersons();

    assertThat(dtos.size()).isEqualTo(0);
}
```

# Validation des données

## A quoi ça sert ?

Lorsque l'on travaille avec des API ou des WebApp, on a souvent recours à la validation pour éviter de pouvoir peupler notre base de données avec des informations n'étant pas voulues (un numéro de téléphone au mauvais format par exemple). La validation peut se faire à plusieurs niveaux:

- Validation dans l'**interface**
- Validation dans les **contrôleurs**
- Validation dans la **base de données**

Pour une application optimisée, il est parfois nécessaire d'avoir recours à de multiples niveaux de validation.



# Exemples d'annotations

- **@Null / @NotNull**: Vérifie si la valeur est nulle ou non
- **@AssertTrue / @AssertFalse**: Vérifie si la valeur est **true** ou **false**
- **@Min / @Max**: Vérifie si la valeur est égale ou supérieure / égale ou inférieure
- **@DecimalMin / @DecimalMax**: Vérifier si la valeur est supérieure ou inférieure
- **@Negative / @NegativeOrZero / @Positive / @PositiveOrZero**: Vérifie le signe de la valeur numérique
- **@Size**: Vérifie la longueur de la collection / chaîne de caractères

## Mise en place

Pour activer ensuite la validation dans nos requêtes, il va falloir ajouter l'annotation **@Validated** sur nos paramètres:

```
@PostMapping
public ResponseEntity handlePost(@Validated @RequestBody PersonDTO person) {
    // ...
}
```

Dans le cas où la validation ne passerait pas, alors un code d'erreur **400** correspondant à une **Bad Request** sera envoyé suite à la levée d'une exception et sa gestion par Spring.

# Retour d'erreur personnalisé

Si l'on laisse Spring répondre de lui même aux erreurs de validation, alors la plupart du temps, trop d'informations seront rendues publiques. Pour éviter cela, on peut faire notre propre retour:

```
@ControllerAdvice
public class CustomErrorController {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    ResponseEntity handleBindErrors(MethodArgumentNotValidException ex) {
        List errorList = exception.getFieldErrors().stream().map(fieldError -> {
            Map<String, String> errorMap = new HashMap();
            errorMap.put(fieldError.getField(), fieldError.getDefaultMessage());
            return errorMap;
        }).collect(Collectors.toList());

        return errorList;
    }
}
```

# Routes avancées

## A quoi ça sert ?

Un paramètre de requête peut nous être utile pour alimenter notre requête sous la forme de:

```
https://route/de/requête?paramA=valueA&paramB=valueB
```

Via leur utilisation, il est possible d'utiliser une même route de plusieurs façon en faisant en sorte que le passage de ce paramètre soit traité par notre application avant un retour personnalisé. Pour cela, il nous faut utiliser l'annotation **@RequestParam**.

# Mise en place

Dans notre contrôleur, il va nous falloir comme à notre habitude des routes avec les mappings adéquats. Cependant, cette fois-ci, notre mapping comportera un ou plusieurs paramètre(s):

```
@GetMapping(value = "/api/v1/person")  
public List<PersonDTO> listPersons(@RequestParam("personFirstName") String personFirstName) {  
    return personService.listPersons()  
}
```

# Récupérer des éléments de la route

Si l'on veut désormais réaliser des routes permettant d'avoir une manipulation spécifique de certaines données, de l'ordre de :

```
https://base/commune/paramA/paramB
```

Il est possible de réaliser ceci via l'annotation **@PathVariable** de sorte à extraire de notre route les éléments voulu:

```
@GetMapping(value = "/api/v1/person/{personId}")  
public List<PersonDTO> getPersonDetails(@PathVariable("personId") String personId) {  
    return personService.getById(personId);  
}
```

# Paging et Sorting

Le mécanisme du **Paging** et du **Sorting** a pour but de permettre aux utilisateurs de visionner les éléments sections par section, en fonction de critères de tri choisis par leurs soins.

Ce mécanisme a une importance particulière lorsqu'il s'agit d'optimiser l'application en évitant l'appel à des milliers de données alors que l'utilisateur ne compte au final uniquement visionner le sommaire des 100 premiers résultats de son tri.

Par défaut Spring Data JPA n'offre pas de paramètre pour le paging, ce qui peut causer un soucis d'allocation mémoire en cas de requête importante.



# Le Paging avec Spring Data JPA

Pour implémenter le mécanisme du Paging avec Spring Data JPA, nous allons faire appel aux **PageRequest**, qui incluent le numéro de page, la taille et potentiellement le mécanisme de tri. Le mécanisme de tri est un objet **Sort** décrivant le tri potentiel des données.

```
@GetMapping(value= "/api/v1/person")
public List<PersonDTO> listPersons(@RequestParam(required = false) Integer pageNumber,
    @RequestParam(required = false) Integer pageSize) {
    return personService.listPersons(pageNumber, pageSize)
}
```

# Utiliser l'objet PageRequest

Dans Spring Data JPA, il nous suffit de faire appel à la méthode **.findAll()** en lui passant en paramètre un objet de PageRequest pour que ce système de paging soit pris en compte au moment du requesting:

```
public PageRequest buildPageRequest(Integer pageNumber, Integer pageSize) {  
    return PageRequest.of(pageNumber, pageSize);  
}  
  
public List<PersonDTO> listPersons(pageNumber, pageSize) {  
    PageRequest pgRequest = buildPageRequest(pageNumber, pageSize);  
  
    personPage = personRepository.findAll(pgRequest);  
}
```

# Ajouter le Sorting

Pour ajouter le système de tri, il faut passer par l'utilisation d'un objet **Sort**.

```
public PageRequest buildPageRequest(Integer pageNumber, Integer pageSize, String propertySorting) {  
    Sort sorting = Sort.by(Sort.Order.asc(propertySorting));  
    return PageRequest.of(pageNumber, pageSize, sort);  
}
```

# Les relations de Données

# L'objectif

Lorsque l'on travaille avec des données de type SQL, il est intéressant de faire usage des liens entre nos entités. Ces liens peuvent se traduire de plusieurs façon en fonction de la cardinalité de nos liaisons (visibles sur le modèle UML de nos bases de données).

- One to One (**1:1**)
- One to Many (**1:n**)
- Many to Many (**n:n**)

# One to One

Pour réaliser une relation de type One to One dans Spring data JPA, il suffit d'utiliser les annotations dédiées dans nos deux classes:

```
@Entity
public class ClassA {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne
    private ClassB classB;
}

@Entity
public class ClassB {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

# One to Many

Dans le cadre d'une liaison **1:n**, on obtient un code de ce genre:

```
@Entity
public class ClassA {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToMany
    private List<ClassB> classesB = new ArrayList<>();
}

@Entity
public class ClassB {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

# Many to Many

```
@Entity
public class ClassA {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany
    private List<ClassB> classesB = new ArrayList<>();
}

@Entity
public class ClassB {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToMany
    @JoinTable(name = "classA_classB", joinColumns = @JoinColumn(name = "classB_id"),
    inverseJoinColumns = @JoinColumn(name = "classA_id"))
    private List<ClassA> classesA = new ArrayList<>();
}
```



# Spring Rest Template

# L'interface Client

Pour utiliser Rest Template, il nous faut créer une variable de type RestTemplateBuilder dans notre Service. Cette variable nous offre ensuite la possibilité de faire appel à des requête via ses méthodes dédiées telles que:

```
@Service
public class MyService {
    private final RestTemplateBuilder restTemplateBuilder;

    public Page<T> list() {
        RestTemplate restTemplate = restTemplateBuilder.build();

        ResponseEntity<String> stringResponse = restTemplate.getForEntity("url", String.class);

        // Suite du code...
    }
}
```

# L'utilisation de Jackson

Dans le cas où l'on choisi au contraire de faire appel à un objet de type **Map<K, V>**, alors Jackson sera employé automatiquement pour réaliser un parsing des valeurs afin que l'on puisse exploiter un peu plus notre élément (pour avoir les headers par exemple):

```
ResponseEntity<Map> stringResponse = restTemplate.getForEntity("url", Map.class);
```

Pour pousser la chose encore plus loin, on peut utiliser le type `JsonNode`. Ce type va amener les features de deserialisation de Jackson dans notre élément. Cet élément est compatible avec la notation du `JsonPath` vue précédemment:

```
ResponseEntity<JsonNode> stringResponse = restTemplate.getForEntity("url", JsonNode.class);
```

## Se lier à un POJO

Bien entendu, l'objectif serait plutôt de se lier directement à l'un des objets utiles à notre code. Pour pouvoir réussir cela, il nous faut utiliser certaines annotations sur nos classes:

```
@JsonIgnoreProperties(ignoreUnknown = true, value = "propD")
public class MyClass {
    @JsonCreator(mode = JsonCreator.Mode.PROPERTIES)
    public MyClass(@JsonProperty("propAName") List<T> propA,
        @JsonProperty("propBName") Integer propB,
        @JsonProperty("propCName") String propC) {
        // Code
    }
}
```

# Configurer le Builder

Pour ajouter la configuration de notre créateur Rest Template, on peut utiliser une classe possédant l'annotation **@Configuration**. Cette annotation, reconnue par Spring, va l'informer qu'il s'agit là d'une classe servant à configurer les **@Bean** de l'application:

```
@Configuration
public class RestTemplateBuilderConfig {
    @Bean
    RestTemplateBuilder restTemplateBuilder(RestTemplateBuilderConfigurer configurer) {
        RestTemplateBuilder builder = configurer.configure(new RestTemplateBuilder());

        DefaultUriBuilderFactory uriFactory = new DefaultUriBuilderFactory("http://localhost:8080");

        return builder.uriTemplateHandler(uriFactory);
    }
}
```

# Externaliser des valeurs

Pour éviter d'avoir des valeurs en dur dans notre code métier, il est possible de faire appel à l'annotation **@Value** et de spécifier la clé utilisée dans le fichier **application.properties** de Spring:

```
@Value("my.root.url")
String rootValue;

...

DefaultUriBuilderFactory uriFactory = new DefaultUriBuilderFactory(rootValue);
```

# Paramètres de requête

Via l'utilisation d'un UriComponentBuilder, il est possible de générer les URL de nos requêtes à partir des paramètres configurés précédemment. De plus, on peut ajouter des paramètres de requête grâce à lui:

```
public Page<ClassNameDTO> listElements(String paramA) {  
    RestTemplate restTemplate = restTemplateBuilder.build();  
  
    UriComponentBuilder uriComponentBuilder = UriComponentBuilder.fromPath("path-url");  
  
    if (paramA != null) {  
        uriComponentBuilder.queryParam("paramA", paramA);  
    }  
  
    ResponseEntity<PageClassNameDTO> stringResponse = restTemplate  
        .getForEntity(uriComponentBuilder.toUriString(), ClassName.class);  
  
    return stringResponse.getBody();  
}
```

# Paramètres de route

Si l'on a désormais besoin de gérer la route de notre requête via une route dynamique, on peut aussi le faire via la classe utilisée précédemment:

```
public ClassNameDTO getElementById(UUID id) {  
    RestTemplate restTemplate = restTemplateBuilder.build();  
  
    return restTemplate.getFromObject("path", ClassNameDTO.class, id);  
}
```



# Les autres verbes HTTP

Pour envoyer des requêtes faisant appel à un corps de requête, on peut utiliser la méthodologie ci-dessous:

```
public ClassNameDTO addNewElement(ClassNameDTO newDto) {  
    RestTemplate restTemplate = restTemplateBuilder.build();  
  
    // Si retour d'objet complexe directement  
    // RestTemplate<ClassNameDTO> responseEntity = restTemplate  
    // .postForEntity("path", newDto, ClassNameDTO.class);  
  
    // return responseEntity.body();  
  
    // Si retour d'un header portant l'Id de l'object inséré  
    URI uri = restTemplate.postForLocation("path", newDto);  
  
    return restTemplate.getForObject(uri.getPath(), ClassNameDTO.class);  
}
```

# Spring Security

# Attention

Afin de nous familiariser avec la sécurisation de notre application, nous allons commencer par réaliser une connexion via le protocole **HTTP**.

Attention, ce mode de procédé devra se faire *in fine* uniquement dans le cadre de l'utilisation de l'**HTTPS** qui offre une sécurisation des informations transmises suite à l'utilisation d'un **handshake** servant à l'encryptage des informations transmises entre le serveur et le client. Dès l'ajout de la dépendance, un utilisateur est créé pour nous ainsi que son mot de passe. Attention à bien le noter, ce dernier se trouvera dans la sortie console du lancement de l'application.

## Ajouter l'autorisation à nos requêtes

Par défaut, tous les endpoints de notre application se verront être sécurisés dès l'ajout de la dépendance.

Pour pouvoir de nouveau avoir accès à nos chemins, il nous faut donc désormais, au niveau de notre client HTTP, faire appel à une authentification usant un nom d'utilisation et un mot de passe (Basic Auth) et y placer **user** ainsi que le **mot de passe sorti en console précédemment**.

# Configurer les informations par défaut

Pour ajouter une configuration de l'username et du mot de passe, on peut le faire via le fichier `application.properties` en ajoutant les lignes suivantes:

```
spring.security.user.name=donald-duck  
spring.security.user.password=my-secret-password
```

Il est bien entendu possible de faire ensuite appel aux variables d'environnement pour éviter le passage de ces informations dans notre dépôt Git.

# Réaliser les tests

Pour éviter que nos tests ne se voient échouer suite à l'utilisation d'un mockMVC et donc des règles de sécurité qui en découlent, on peut ajouter une nouvelle dépendance à **Spring-Security-Test** pour nous aider à ajouter les credentials requis:

```
mockMVC.perform(get("url")
    .with(httpBasic("username", "password"))
    .accept(MediaType.APPLICATION_JSON)
    .andExpect(status().isOk()));
```

# Désactiver le jeton CSRF

Il se peut cependant que nos tests ne passent pas de part le filtre CSRF. Pour le configurer, il va nous falloir ajouter une classe de configuration et ne pas oublier d'appliquer la sécurité à notre mock:

```
@Configuration
public class SpringSecurityConfig {
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests()
            .anyRequest().authenticated()
            .and().httpBasic(Customizer.withDefaults())
            .csrf().ignoreRequestMatchers("/api/**");

        return http.build();
    }
}
```

# Refactoring de RestTemplate

Nous pouvons désormais faire un peu de refactoring sur notre configuration de RestTemplate en prenant en compte l'authentification de base:

```
@Configuration
public class RestTemplateBuilderConfig {
    @Bean
    RestTemplateBuilder restTemplateBuilder(RestTemplateBuilderConfigurer configurer) {
        return configurer
            .configure(new RestTemplateBuilder())
            .basicAuthentication(username, password)
            .uriTemplateHandler(new DefaultUriBuilderFactory(rootUrl));
    }
}
```



# JSON Web Token

## Késako ?

Spring Security est un ensemble de classes / interfaces que Spring va utiliser pour organiser la couche de sécurité de notre application. Par exemple, parmi ces éléments se trouvent:

- **UserDetails:** L'interface permettant la création d'utilisateur
- **UserDetailsService:** Le service de base qui se charge de récupérer les éléments de type UserDetails.
- **AuthenticationManager:** L'élément gérant la création de jetons d'authentification
- **SecurityContextHolder:** L'élément traitant les jetons

## SecurityFilterChain (1/2)

Lorsque l'on fait une requête à notre application, la requête doit passer par une série de filtre pour être admise de continuer ainsi que de procurer une réponse adéquate. Dans le cas contraire, une erreur d'authentification a lieu, provoquant un code de status **403**.

Pour passer d'un filtre à l'autre, la méthode à retenir se nomme **.doFilter()**.

# SecurityFilterChain (2/2)

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .httpBasic();

        return http.build();
    }
}
```

# UserDetails

Dans l'univers de Spring, la capacité de lier nos utilisateurs au système de connexion passe par l'implémentation de notre POJO de l'interface **UserDetails**:

```
@Bean
public UserDetailsService userDetailsService() {
    UserDetails admin = User.builder()
        .username("admin").password("password").roles("ADMIN").build();

    UserDetails user = User.builder()
        .username("user").password("password").roles("USER").build();

    return new InMemoryUserDetailsManager(admin, user);
}
```

## requestMatchers()

Les **requestMatchers** sont au final une façon de filter quelles requêtes doivent nécessiter l'authentification et quelles requêtes peuvent être permises sans connexion.

- Il est alors possible de trier par méthode HTTP:

```
.requestMatchers(HttpMethod.GET).permitAll()
```

- Ou par route dans notre application, ainsi que par rôle:

```
.requestMatchers("/api/v1/private/**").hasRole("ADMIN")
```

# Les Roles

La sécurisation d'une application passe généralement par deux étapes:

- L'**Authentication**, qui permet la connexion des utilisateurs, évitant ainsi les erreurs de type **403**.
- L'**Autorisation**, qui permet la validation des droits spécifiques à un utilisateur, évitant ainsi les erreurs de type **401**.

Pour gérer la seconde partie, il est courant dans une application de faire appel aux **Roles** ou aux **GrantedAuthorities** (formes plus spécifiques encore) se trouvant bien souvent dans un objet de type **Claims**.

## La couche Données (1/3)

Il nous faut désormais avoir en Base de données nos utilisateurs et nos rôles pour pouvoir ajouter ou connecter les utilisateurs de notre application:

```
public interface UserRepository extends JpaRepository<UserEntity, UUID> {  
    Optional<UserEntity> findByUsername(String username);  
    Boolean existsByUsername(String username);  
}
```

```
public interface RoleRepository extends JpaRepository<RoleEntity, UUID> {  
    Optional<RoleEntity> findByName(String name);  
}
```



# La couche Données (2/3)

```
@Entity
@Data
@NoArgsConstructor
@Table(name = "users")
public class UserEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    private String username;
    private String password;

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinTable(name = "user_roles", joinColumns = @JoinColumn(name = "user_id", referencedColumnName = "id"),
        inverseJoinColumns = @JoinColumn(name = "role_id", referencedColumnName = "id"))
    private List<RoleEntity> roles = new ArrayList<>();
}
```

## La couche Données (3/3)

```
@Entity
@Data
@NoArgsConstructor
@Table(name = "roles")
public class RoleEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    private String name;
}
```

# Notre propre service UserDetails

Pour pouvoir gérer nous même la récupération des utilisateurs, il nous faut créer une implémentation de l'interface UserDetailsService afin d'informer Spring de la méthode à suivre:

```
@Service
@RequiredArgsConstructor
public class CustomUserDetailsService implements UserDetailsService {
    private final UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        UserEntity user = userRepository.findByUsername(username).orElseThrow(() -> new UsernameNotFoundException("Username not found!"));

        return new User(
            user.getUsername(),
            user.getPassword(),
            mapRolestoAuthorities(user.getRoles())
        );
    }

    private Collection<GrantedAuthority> mapRolestoAuthorities(List<RoleEntity> roles) {
        return roles.stream()
            .map(role -> new SimpleGrantedAuthority(role.getName()))
            .collect(Collectors.toList())
    }
}
```

# Configuration de l'AuthenticationManager

Pour permettre l'authentification, Spring se sert d'un élément appelé AuthenticationManager. Il est possible de le configurer manuellement, ainsi que le mécanisme d'encryptage de notre mot de passe. Ici, nous utiliserons cependant les valeurs par défaut:

```
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration configuration) throws Exception {
    return configuration.getAuthenticationManager();
}
```

```
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

# Le Registering

```
@PostMapping("/register")
public ResponseEntity<String> register(@RequestBody RegisterRequest request) {
    if (userRepository.existsByUsername(request.getUsername())) {
        return new ResponseEntity<>("Username is taken!", HttpStatus.BAD_REQUEST);
    }

    UserEntity user = new UserEntity();
    user.setUsername(request.getUsername());
    user.setPassword(passwordEncoder.encode(request.getPassword()));

    Optional<RoleEntity> role = roleRepository.findByName("USER");

    if (role.isEmpty()) {
        RoleEntity newUserRole = new RoleEntity();
        newUserRole.setName("USER");
        user.setRoles(Collections.singletonList(newUserRole));
    } else {
        user.setRoles(Collections.singletonList(role.get()));
    }

    userRepository.save(user);

    return new ResponseEntity<>("User registered successfully!", HttpStatus.CREATED);
}
```

# Le Login

```
@PostMapping("/authenticate")
public ResponseEntity<AuthenticationResponse> register(@RequestBody AuthenticateRequest request) {
    Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(
            request.getUsername(),
            request.getPassword()
        )
    );

    SecurityContextHolder.getContext().setAuthentication(authentication);
    String token = tokenGenerator.generateToken(authentication);
    AuthenticationResponse response = new AuthenticationResponse(token);
    return new ResponseEntity<>(response, HttpStatus.OK);
}
```

# Ajout du JWT (1/2)

```
@Component
public class TokenGenerator {
    public String generateToken(Authentication authentication) {
        String username = authentication.getName();
        Date expirationDate = new Date(new Date().getTime() + SecurityConstants.JWT_EXPIRATION);

        String token = Jwts.builder()
            .setSubject(username)
            .setExpiration(expirationDate)
            .setIssuedAt(new Date())
            .signWith(SignatureAlgorithm.HS256, SecurityConstants.JWT_SECRET)
            .compact();

        return token;
    }

    ...
}
```

# Ajout du JWT (2/2)

```
...

public String getUsernameFromToken(String token) {
    Claims claims = Jwts.parserBuilder()
        .setSigningKey(SecurityConstants.JWT_SECRET)
        .build()
        .parseClaimsJws(token)
        .getBody();
    return claims.getSubject();
}

public Boolean validateToken(String token) {
    try {
        Jwts.parserBuilder().setSigningKey(SecurityConstants.JWT_SECRET).build()
            .parseClaimsJws(token);
        return true;
    } catch (Exception ex) {
        throw new AuthenticationCredentialsNotFoundException("JWT was expired or incorrect!");
    }
}
}
```



# OncePerRequestFilter

```
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final TokenGenerator tokenGenerator;
    private final CustomUserDetailsService userDetailsService;
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
        String token = getJWTFromRequest(request);
        if (StringUtils.hasText(token) && tokenGenerator.validateToken(token)) {
            String username = tokenGenerator.getUsernameFromToken(token);

            UserDetails userDetails = userDetailsService.loadUserByUsername(username);

            UsernamePasswordAuthenticationToken authenticationToken = new UsernamePasswordAuthenticationToken(
                userDetails.getUsername(),
                null,
                userDetails.getAuthorities());

            authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authenticationToken);
        }

        filterChain.doFilter(request, response);
    }

    private String getJWTFromRequest(HttpServletRequest request) {
        String bearerToken = request.getHeader("Authorization");
        if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring("Bearer ".length());
        }

        return null;
    }
}
```

# Modification de la FilterChain

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .exceptionHandling()
        .authenticationEntryPoint(authEntryPoint)
        .and()
        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
        .authorizeRequests()
        .requestMatchers("/api/v1/auth/**").permitAll()
        .anyRequest().authenticated()
        .and()
        .addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

