



API REST – SPRING BOOT

B3 - PGL

2022 - 2023

[Chap. 1] API REST

Rappel des notions de JEE vues en B2

Concept API

Principes d'une API REST

[Chap. 2] Spring Boot

Spring Framework et Spring Boot

Développement de l'architecture d'une API REST

Front Controller pattern

WebClient

[Chap. 3] Spring Data

Présentation d'Hibernate et de Spring Data

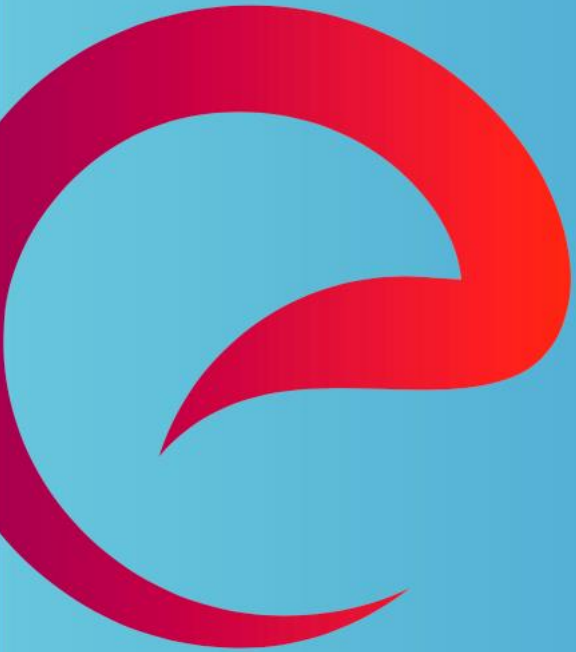
Entités

Relations

[Chap. 4] Spring Security

Sécurisation

Déploiement sur un serveur

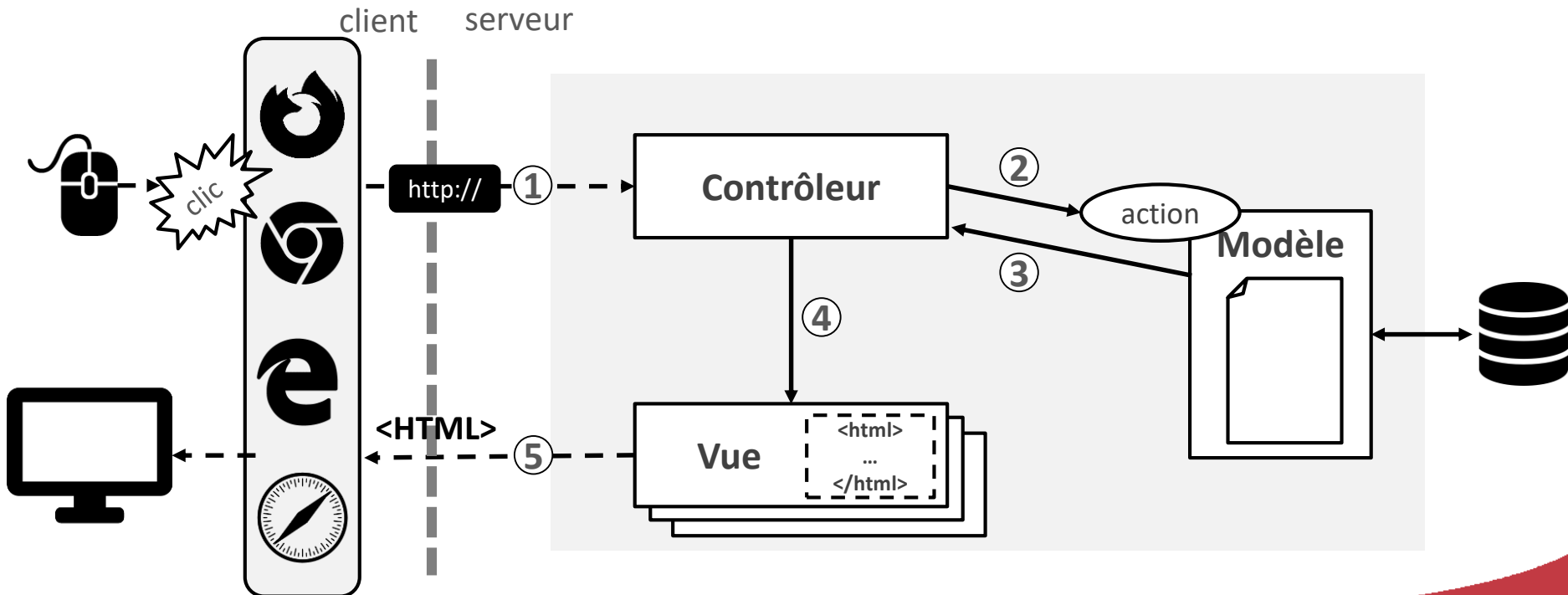


API REST

RAPPEL DES NOTIONS DE JEE VUES EN B2

Le MVC (Modèle Vue Contrôleur) pour le Web est un design pattern qui découpe l'application en 3 couches principales :

- **Modèle** : contient la logique et les informations métiers nécessaires pour répondre aux demandes de l'utilisateur
- **Vue** : affiche les informations attendues par l'utilisateur final
- **Contrôleur** : reçoit les demandes en provenant de l'utilisateur final et contrôle le comportement de l'application interactive pour répondre à ces demandes



RAPPEL DES NOTIONS DE JEE VUES EN B2

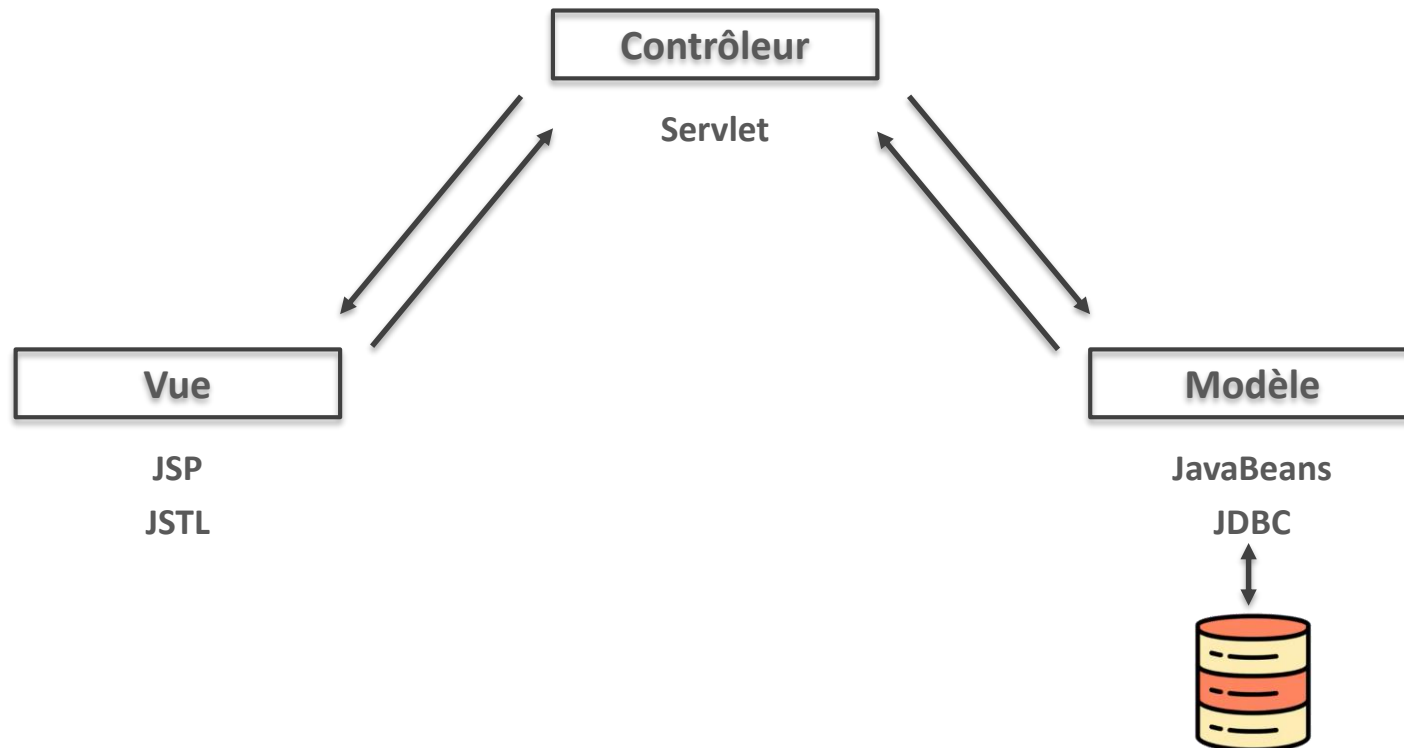
Servlet : programme Java sur le serveur qui route une requête HTTP et un vers autre service HTTP

JSP : permet de générer dynamiquement des pages web

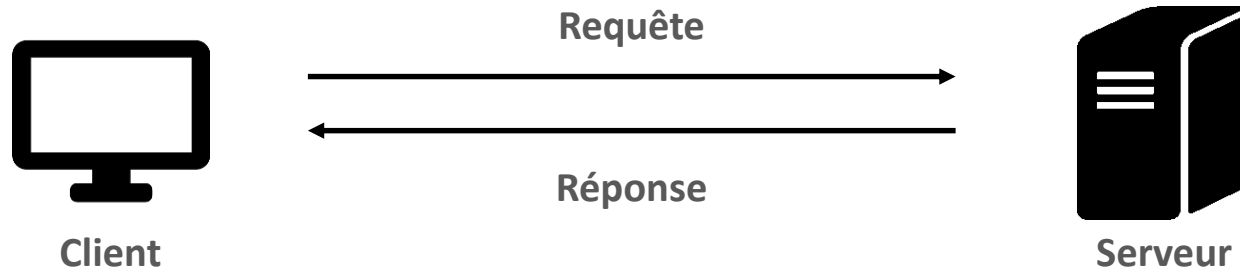
JSTL : permet de développer une page JSP sans incorporer directement code Java

JavaBeans : classe Java qui expose des propriétés et des listeners en respectant les conventions de codage

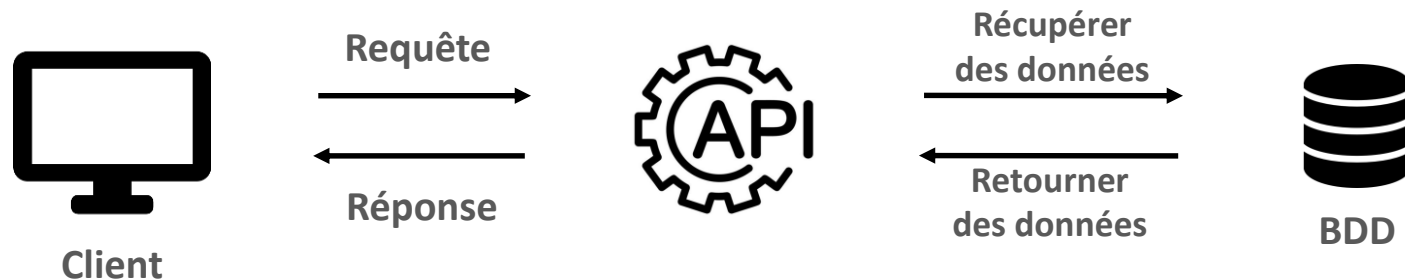
JDBC : permet l'accès aux bases de données



CONCEPT D'API



Les API (Application Programming Interface) servent à communiquer entre plusieurs applications, elles agissent comme un intermédiaire qui transmet des messages à travers un système de requêtes et de réponses.



CONCEPT D'API

Les principaux avantages des API sont :

 la **standardisation**

 la **réutilisabilité**

 la **maintenance**



Un WS (Web Service) et une API sont tous les deux des moyens de communication.

Un WS facilite seulement la communication entre des machines via un réseau alors qu'une API facilite l'interaction entre des applications différentes.

Toutes les API ne sont pas des WS, mais tous les WS sont des API.

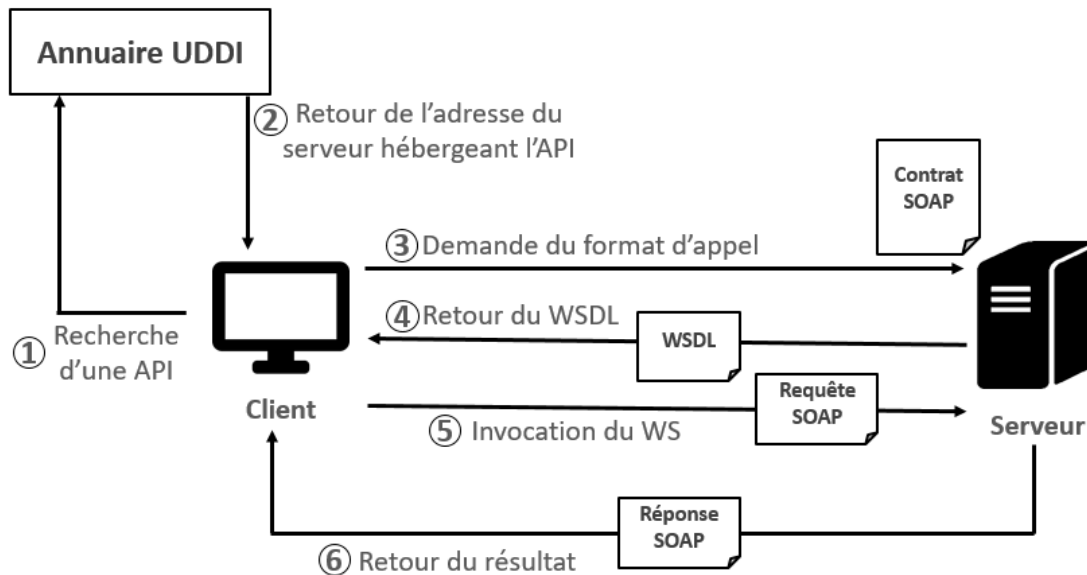
Les API peuvent être publiques (accessibles par tous) ou privées (accès restreint).

CONCEPT D'API

Les deux principales technologies pour mettre en œuvre les API sont :

➤ SOAP

Reposent sur un ensemble de protocoles et de standards pour pouvoir échanger des données en exposant des fonctionnalités sous la forme de services exécutables à distance : SOAP (standard pour l'échange de messages XML) ; WSDL (grammaire XML permettant de description) ; UDDI (service d'annuaire).



👍 Adapté dans le cadre d'un contrat strict (e.g. pour la communication interbancaire)

👎 Rigidité de la solution (e.g. une modification côté serveur entraînera le dysfonctionnement du client)

Toutes les communications entre l'annuaire, le client et le serveur se font via le protocole SOAP.

CONCEPT D'API

Les deux principales technologies pour mettre en œuvre les API sont :

➤ REST

Architecture définissant un ensemble de contraintes à utiliser pour créer des API. Les requêtes seront effectuées sur des URI de ressources et des réponses (formatées en HTML, XML, JSON, ...) seront renvoyées. REST est indépendant du protocole pourvu qu'il possède un schéma standard pour une URI.



👍 découplage client-serveur (en versionnant l'API les clients peuvent continuer à utiliser des anciennes versions)

👍 API REST sont plus légères que SOAP et donc plus adaptées l'Internet des objets (IoT), les applications mobiles et l'informatique sans serveur.

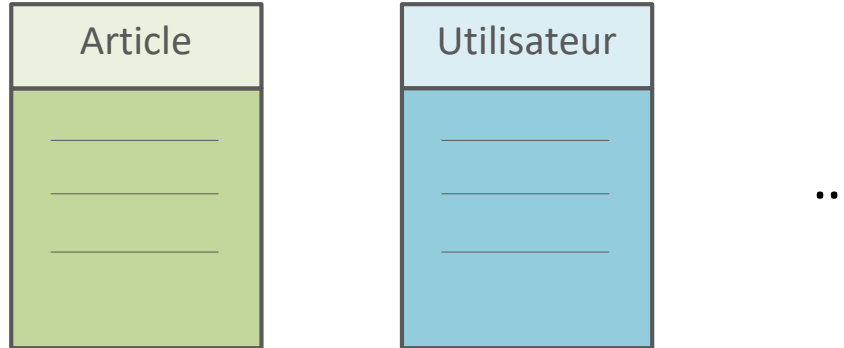
PRINCIPES D'UNE API REST

Une application REST respecte six recommandations architecturales (dans les faits il est rare qu'une application respecte les six...) :

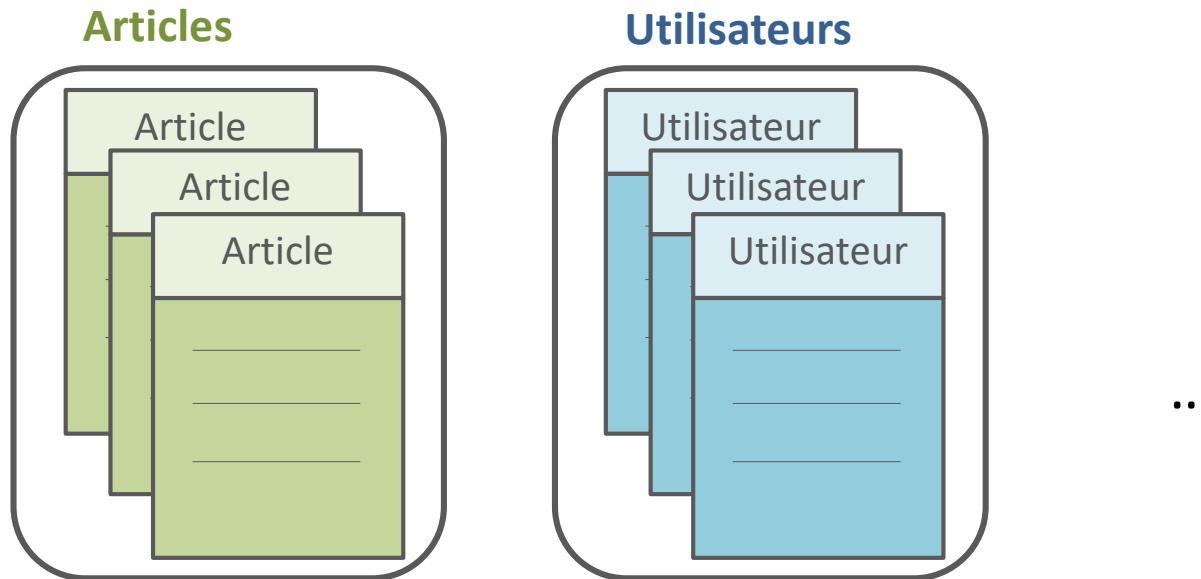
- **séparation entre du client et du serveur**
- communications **sans état** (garantir qu'aucune information client n'est stockée sur le serveur)
- la plateforme doit pouvoir **mettre en cache** les réponses pour augmenter la vitesse
- **interface uniforme** : utilisation de standards partagés
- **système à couches** : ne pas accéder au-delà du composant précis avec lequel on interagit
- **code à la demande** qui permet au serveur d'étendre la fonctionnalité d'un client en transférant le code exécutable

PRINCIPES D'UNE API REST

Les données échangées via une API REST sont représentées sous forme de **ressources**.



Les ressources sont regroupées dans un groupe appelé une **collection**.



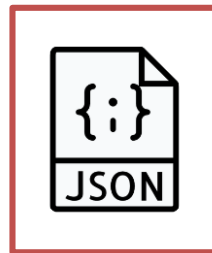
PRINCIPES D'UNE API REST

L'API connaîtra l'emplacement des ressources demandées par le client grâce au **URL de la requête** qui lui sera fourni : **URL / URI (appelé aussi endpoint)**

↓ ↓
https://monsite.fr/articles liste tous les articles

https://monsite.fr/articles/6 pour obtenir qu'uniquement l'article dont l'ID est 6

Les données échangées seront stockées dans un fichier de manière organisée. Pour cela deux formats de données sont généralement utilisés :



Plus rapide



Moins verbeux



Plus simple à analyser

PRINCIPES D'UNE API REST

Association des méthodes HTTP (les requêtes des clients) aux opérations CRUD

Méthodes HTTP	Opérations CRUD
POST : créer une ressource	Create
GET : récupérer une ressource	Read
PUT : remplacer une ressource	Update
DELETE : supprimer une ressource	Delete

Passer un paramètre

GET <https://monsite.fr/articles?type=desktop>



? permet de passer un paramètre

PRINCIPES D'UNE API REST

Agréger

GET <https://monsite.fr/articles?sort=nom&order=asc>



& permet d'agréger des paramètres

Versionner

GET <https://monsite.fr/v2/articles>




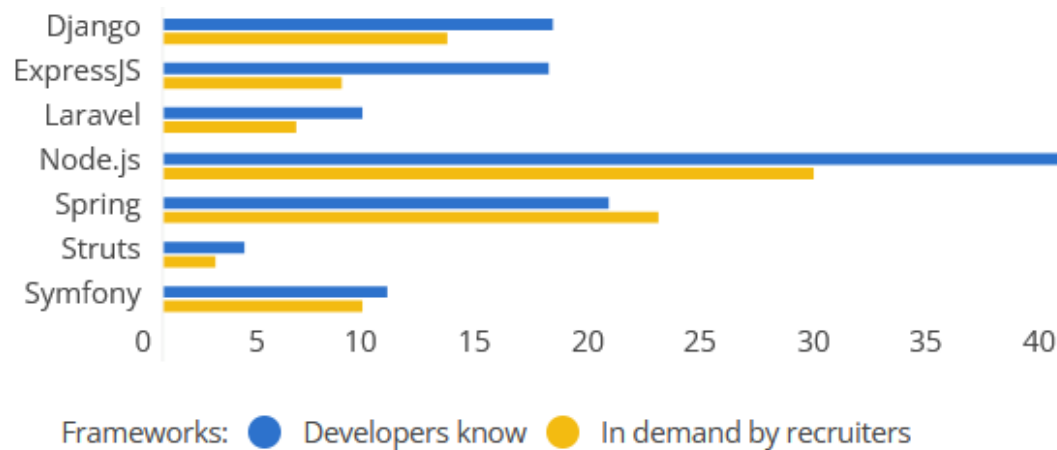
il est possible de préciser la version de l'API à utiliser



SPRING BOOT

SPRING FRAMEWORK ET SPRING BOOT

 **spring**® est un Framework pour faciliter et automatiser le développement d'application en Java (et particulièrement utilisé en JEE).

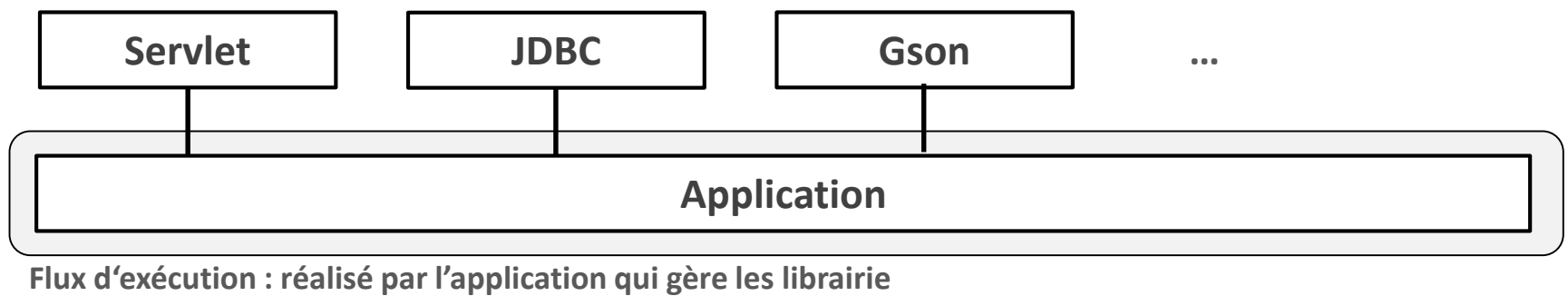


Étude 2022 de CodinGame sur 14 000 développeurs et recruteurs du monde entier.

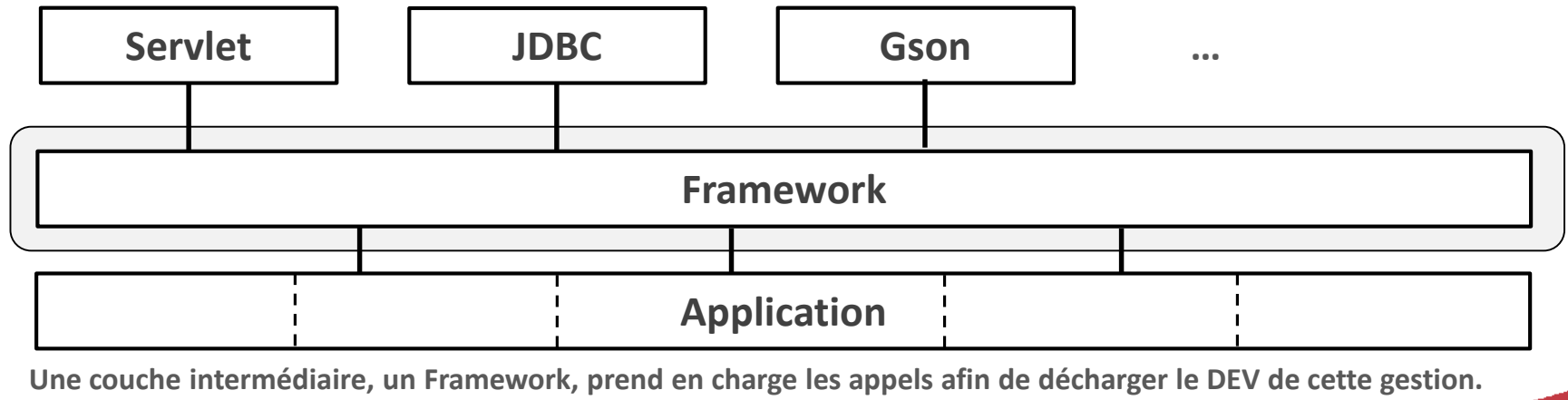
SPRING FRAMEWORK ET SPRING BOOT

Inversion du contrôle

Programmation classique



Avec Spring

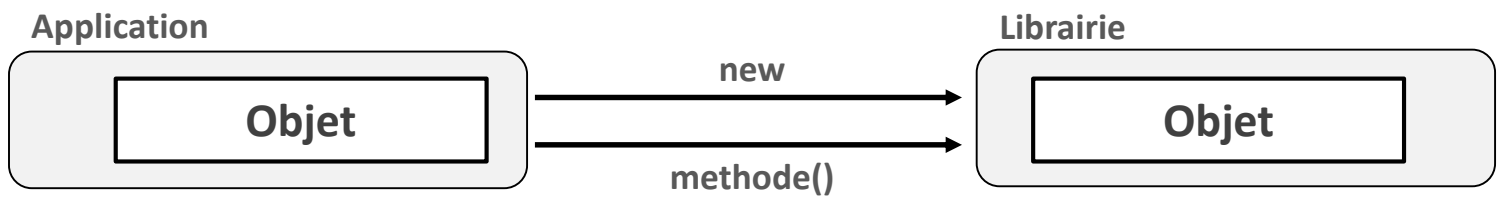


SPRING FRAMEWORK ET SPRING BOOT

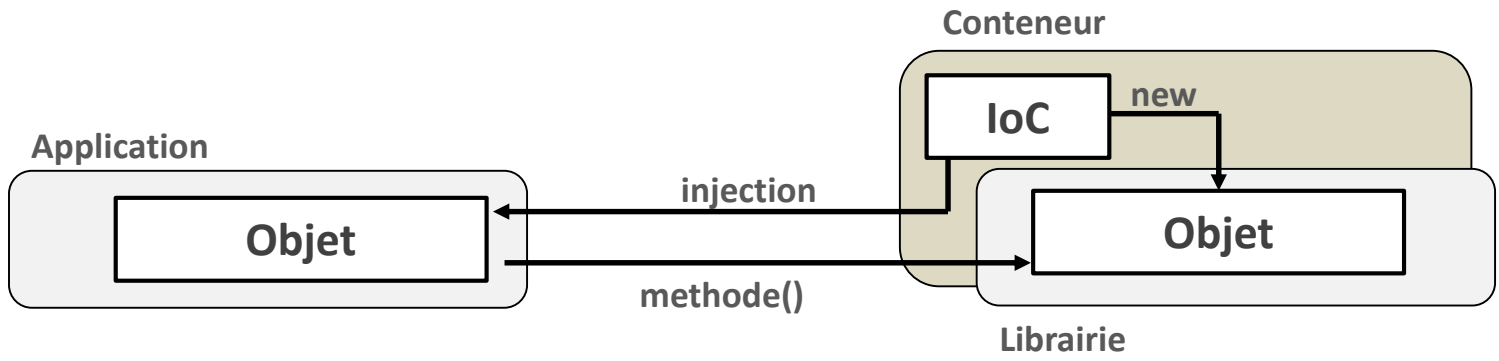
Injection de dépendances

 Réduire les dépendances

Programmation classique

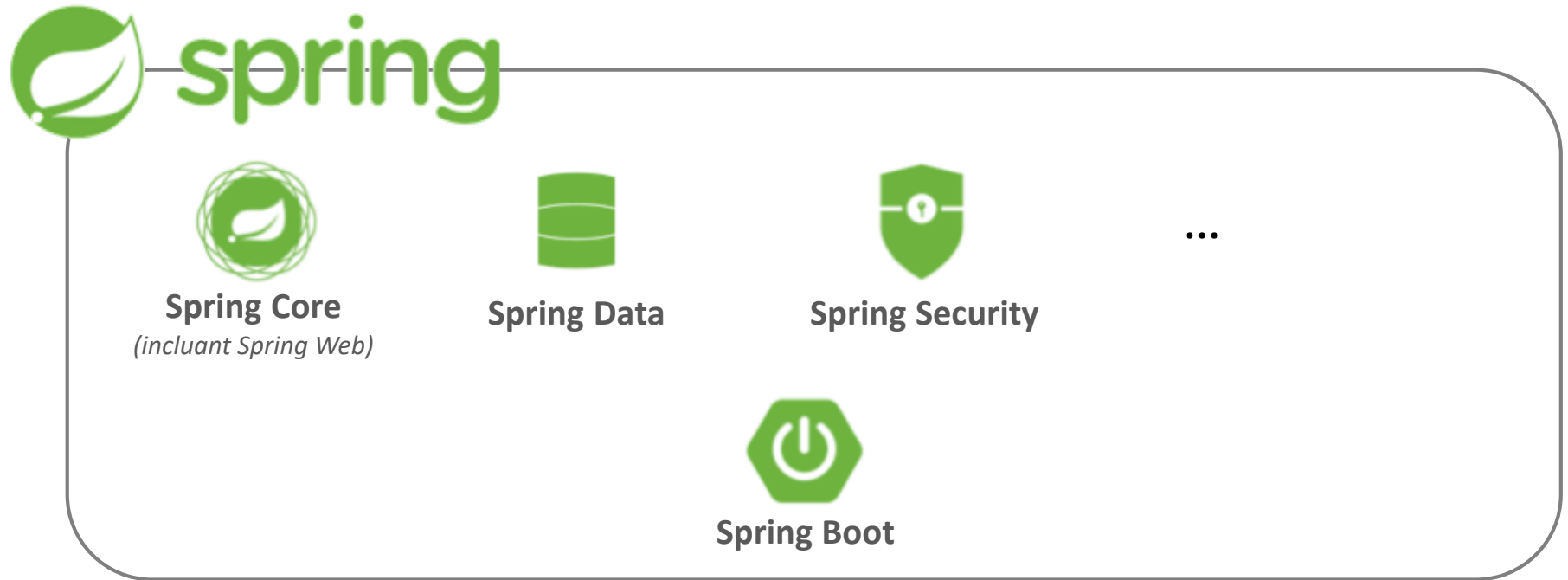


Avec Spring



Le conteneur, via le Framework, va se charger de créer l'objet de la librairie et fournira à l'application la référence de celui-ci (principe d'injection).
 Le DEV est donc affranchi de créer l'objet de la librairie.

SPRING FRAMEWORK ET SPRING BOOT



Spring Boot est un framework, construit sur le framework Spring, où tout est configuré automatiquement. Spring Boot est particulièrement utilisé pour développer une **API REST**.

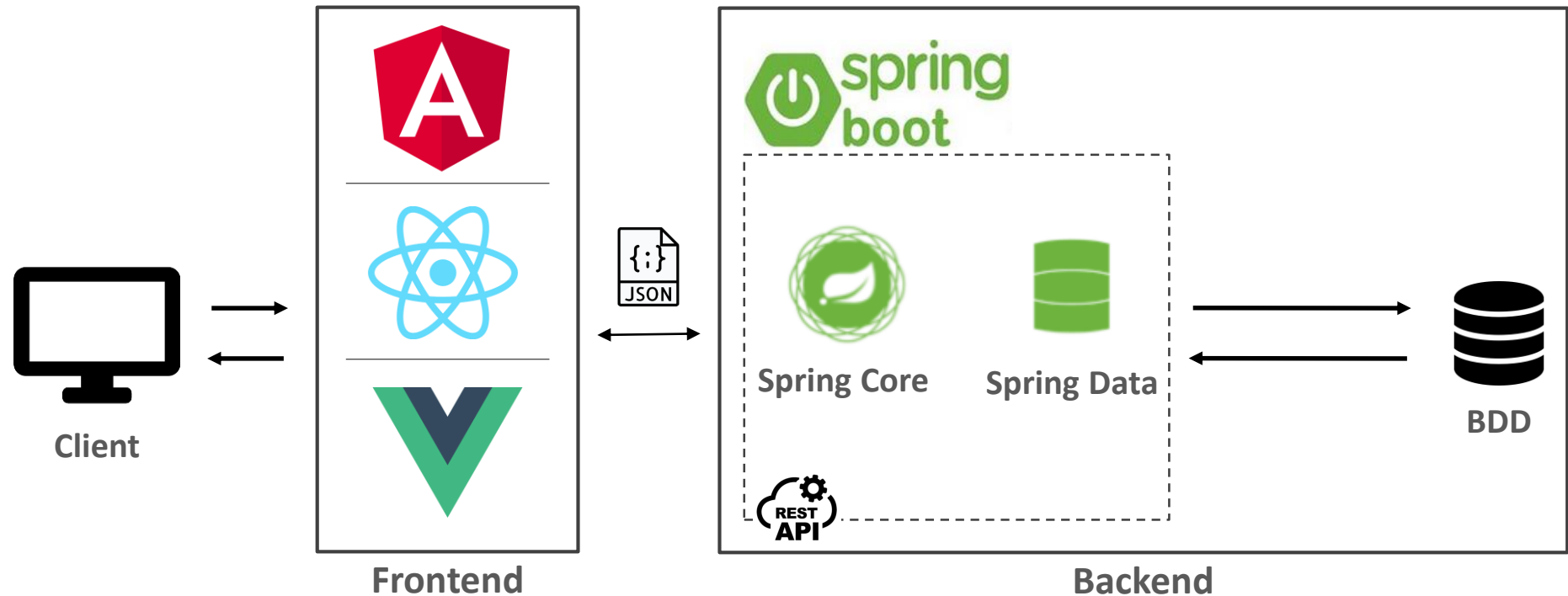
Il permet de mettre en œuvre de tous les autres frameworks de Spring.

SPRING FRAMEWORK ET SPRING BOOT

Spring Boot va réduire la complexité de Spring Framework.

- 👍 Gestion des dépendances : les starters, ensemble de dépendances homogénéisées
- 👍 Auto configuration
- 👍 Déploiement : Tomcat est embarqué, ...
- ...

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST



Application Web implémentée en SPA (Single-page application)

- 👍 Fluidité et temps de chargement réduit
- 👍 Maintenabilité accrue grâce aux Framework

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Partant des maquettes suivantes, nous allons construire une API REST permettant la communication avec une base de données.

Frontend :   

Backend : 

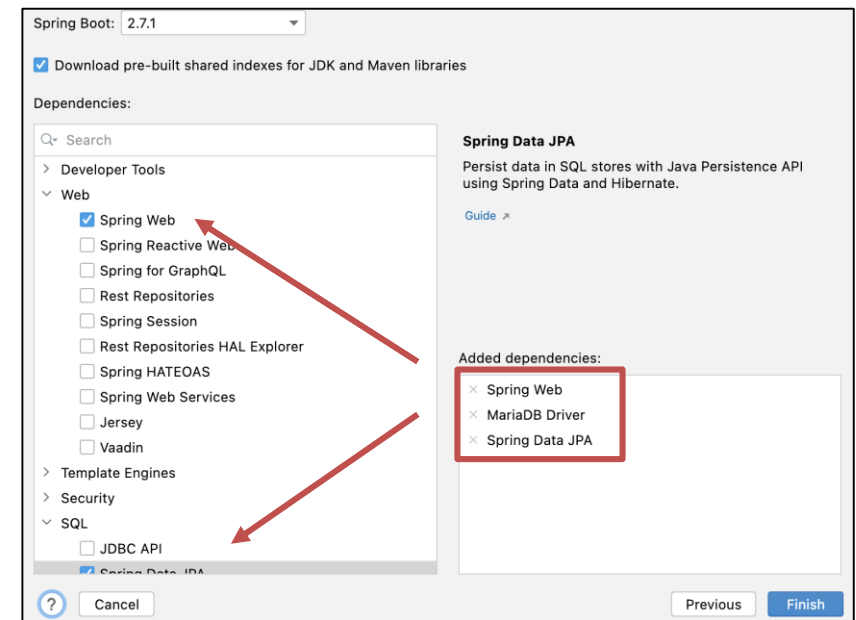
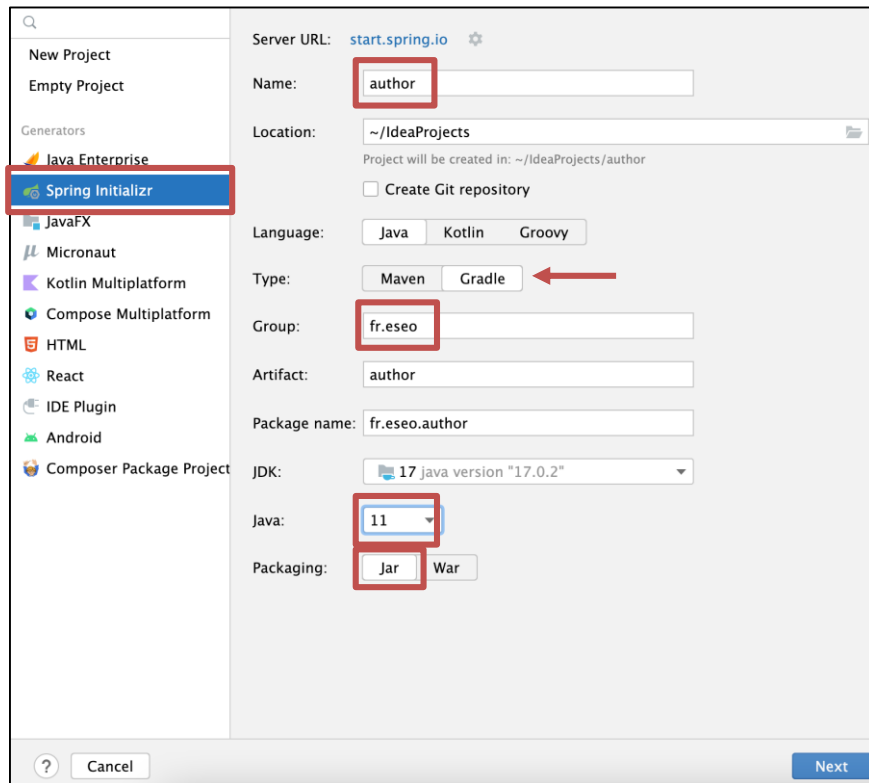
Base de données :  MariaDB

NB : nous n'utiliserons pas de Framework Frontend pour faciliter la compréhension, mais il est conseillé d'en utiliser dans vos projets :



DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Dans IntelliJ IDEA, créez un nouveau projet « Spring Initializr » et choisissez les starters.



Le téléchargement des dépendances et la construction du projet peuvent prendre un certain temps.


NB : Tomcat est directement intégré et l'application exécutera simplement avec le JAR.

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

La classe **AuthorApplication** sera automatiquement générée, elle lancera (entre autres) la **SpringApplication** qui est responsable du démarrage de l'application Spring.

@SpringBootApplication permet :

- de définir des configurations
- de générer automatiquement les configurations nécessaires en fonction des dépendances
- d'indiquer qu'il faut scanner les classes pour trouver des Beans de configuration



```
@SpringBootApplication
public class AuthorApplication {

    public static void main(String[] args) {
        SpringApplication.run(AuthorApplication.class, args);
    }

}
```


DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Le fichier `application.properties` permet de modifier des configurations liées à Spring Boot.

 `application.properties` ✕

```
server.port=8080

spring.datasource.url=jdbc:mariadb://localhost:3306/spring_example
spring.datasource.username=root
spring.datasource.password=MySQL2021
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.jpa.database-platform = org.hibernate.dialect.MySQL5Dialect
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto = update
```

Nous avons défini le port de Tomcat et l'accès à la base de données.

 Pensez à créer une base de données nommée `spring_example`

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Nous allons exposer une API REST qui pourra être appelée sur les URL suivantes :

- **GET /authors** : affiche la liste de tous les auteurs
- **GET /authors/{id}** : affiche l'auteur via son id
- **PUT /authors/{id}** : met à jour un auteur via son id
- **POST /authors** : ajoute un auteur
- **DELETE /authors/{id}** : supprime un auteur via son id

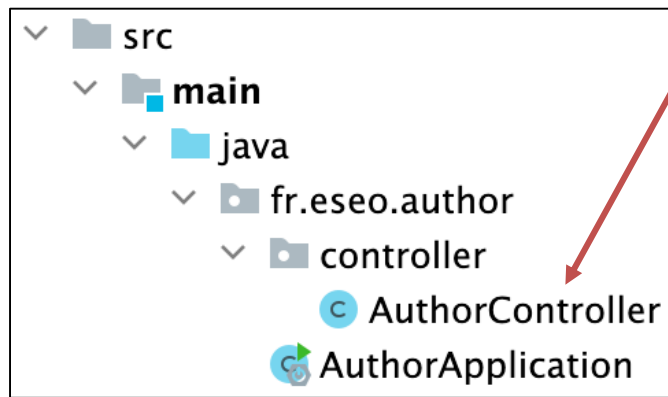
Cette API respectera donc les opérations CRUD.

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Créer le contrôleur REST

Créer un paquetage **controller** inclus dans le paquetage fr.eseo.author

Puis créer une classe nommée **AuthorController** dans ce nouveau paquetage.



```
@RestController
public class AuthorController {
}
```

Ajouter l'annotation **@RestController** qui permet :

- d'indiquer que cette classe va pouvoir traiter les requêtes
- d'indiquer que chaque méthode va renvoyer directement la réponse JSON

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Méthodes pour GET

Dans un premier temps nous n'allons pas relier les méthodes à la base de données, mais juste mettre la structure en place (les méthodes renverront un exemple de données).

```
@RestController
public class AuthorController {

    @GetMapping("/authors")
    public List<Author> getAuthors() {
        return null;
    }

    @GetMapping("/authors/{id}")
    public Optional<Author> getAuthor(@PathVariable int id) {
        return null;
    }
}
```

@GetMapping permet de faire le lien entre l'URI /authors invoquée via GET et la méthode getAuthors

L'ajout d'un identifiant permettra de différencier les requêtes pour un auteur spécifique.

@PathVariable permet de lire la variable dans l'URI.

Optional permet d'encapsuler un objet dont la valeur peut être null : [documentation](#)

2 – SPRING BOOT

```
public class Author {

    private int id;
    private String firstName;
    private String lastName;

    public Author() {
    }

    public Author(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int getId() {
        return id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Nous allons maintenant mettre en place une réponse au format JSON en créant un modèle.

Créer un paquetage **model** dans le paquetage `fr.eseo.author`, puis une classe `Author` dans celui-ci.

Cette classe doit avoir un constructeur sans paramètre et des getters / setters.

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Méthodes pour GET

Nous allons mettre en place une DAO qui permettra de communiquer avec la base de données.

Spring se charge de fabriquer une instance du fait de la déclaration en private final.

```
@RestController
public class AuthorController {

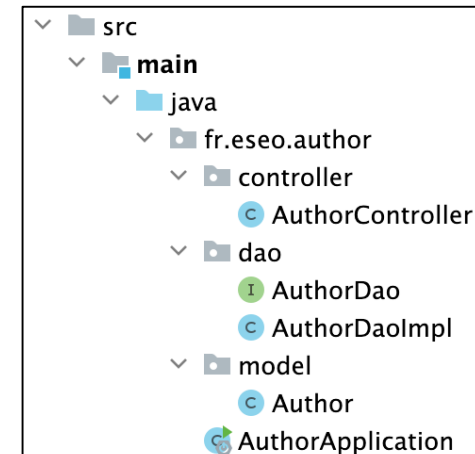
    private final AuthorDao authorDao;

    public AuthorController(AuthorDao authorDao) {
        this.authorDao = authorDao;
    }

    @GetMapping("/authors")
    public List<Author> getAuthors() {
        return authorDao.findAll();
    }

    @GetMapping("/authors/{id}")
    public Optional<Author> getAuthor(@PathVariable int id) {
        return authorDao.findById(id);
    }
}
```

```
public interface AuthorDao {
    List<Author> findAll();
    Optional<Author> findById(int id);
    void save(Author author);
}
```



DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

@Repository

Permet d'indiquer que cette classe gère les données.

```
public class AuthorDaoImpl implements AuthorDao {

    private List<Author> dummy = new ArrayList<>(Arrays.asList(
        new Author(1, "George", "Orwell"),
        new Author(2, "J. R. R.", "Tolkien")
    ));

    @Override
    public List<Author> findAll() {
        return dummy;
    }

    @Override
    public Optional<Author> findById(int id) {
        for (Author author : dummy) {
            if (author.getId() == id) {
                return Optional.of(author);
            }
        }
        return null;
    }

    @Override
    public void save(Author author) {
        dummy.add(author);
    }
}
```

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Compilez et lancez l'application, puis testez la grâce à un navigateur.

<http://localhost:8080/authors>

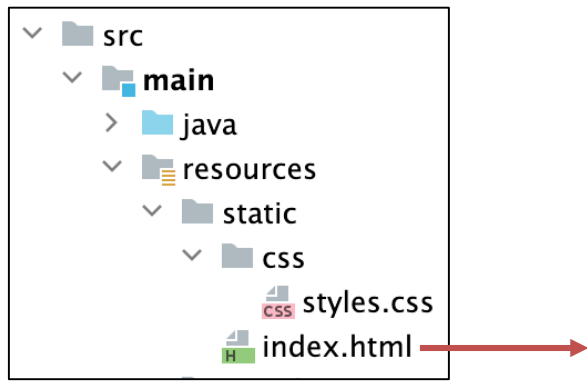
```
{"id":2,"firstName":"J. R. R. ","lastName":"Tolkien"}
```

<http://localhost:8080/authors/2>

```
[{"id":1,"firstName":"George","lastName":"Orwell"}, {"id":2,"firstName":"J. R. R. ","lastName":"Tolkien"}]
```


DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Pour ajouter un Frontend au projet, déposez les fichiers dans le dossier static.



```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="css/styles.css">
    <title>Auteurs</title>
  </head>
  <body>
    <table>
      <thead>
        <td>Prénom</td>
        <td>Nom</td>
      </thead>
      <tbody id="authors"></tbody>
    </table>
    <script>
      fetch('/authors')
        .then(response => response.json())
        .then(data => data.forEach(element => {
          let row = '<tr><td>' + element.firstName + '</td>' +
            '<td>' + element.lastName + '</td></tr>';
          document.getElementById("authors").innerHTML += row;
        })
      </script>
    </body>
  </html>
```

*Si vous souhaitez faire communiquer deux domaines entre eux il faudra ajouter l'annotation **@CrossOrigin** au contrôleur.*

Prénom	Nom
George	Orwell
J. R. R.	Tolkien

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Méthode pour GET

Pour récupérer des paramètres passés dans l'URI vous pourrez utiliser **@RequestParam**, par exemple :

Appel à `http://localhost:8080/weather?latitude=47.4739884&longitude=-0.5515588`

```
@GetMapping("/weather")
public Weather getWeather(@RequestParam(name = "latitude") Double latitude,
    @RequestParam(name = "longitude") Double longitude) {

    // ...

}
```

DÉVELOPPEMENT DE L'ARCHITECTURE D'UNE API REST

Méthode pour POST

```
@RestController
public class AuthorController {

    // les autres méthodes...

    @PostMapping("authors")
    public void addAuthor(@RequestBody Author author) {
        authorDao.save(author);
    }
}
```

@PostMapping permet de faire le lien avec les requêtes POST.

@RequestBody demande à Spring que le JSON contenu dans la partie body de la requête HTTP soit converti en objet Java.

2 – SPRING BOOT

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="css/styles.css">
    <title>Auteurs</title>
  </head>
  <body>
    <table>
      <thead>
        <td>Prénom</td>
        <td>Nom</td>
      </thead>
      <tbody id="authors"></tbody>
    </table>
    <script>
      fetch('/authors')
        .then(response => response.json())
        .then(data => data.forEach(element => {
          let row = '<tr><td>' + element.firstName + '</td>' +
            '<td>' + element.lastName + '</td></tr>';
          document.getElementById("authors").innerHTML += row;
        }
      ))
    </script>

    <h1>Ajouter un auteur</h1>
    <form action="javascript:addAuthor();">
      <input type="text" id="firstName" name="firstName" placeholder="Prénom" required>
      <input type="text" id="lastName" name="lastName" placeholder="Nom" required>
      <input type="submit" value="Valider">
    </form>
    <script>
      function addAuthor() {
        fetch('/authors', {
          method: 'POST',
          headers: {'content-type': 'application/json'},
          body: JSON.stringify({
            firstName:document.getElementById("firstName").value,
            lastName:document.getElementById("lastName").value
          })
        })
        .then(response => {
          response.text();
          window.location.reload();
        })
      }
    </script>
  </body>
</html>
```



POSTMAN

Application permettant de tester des API :

- choisir le type de requête à envoyer
- choisir l'URL de l'API
- les paramètres à passer
- ...

Vous verrez ainsi les requêtes / les retours et vous pourrez les analyser.

FRONT CONTROLLER PATTERN

Spring MVC intègre un DispatcherServlet qui agit comme un Front Controller pattern.

Le DispatcherServlet va gérer les requêtes HTTP entrantes en les déléguant selon les « instructions » implémentées.

Vous n'aurez qu'à définir les contrôleurs en définissant un mappage des requêtes HTTP spécifique pour chacun d'entre eux.

```
@RestController
@RequestMapping("/item")
public class ItemController {

    @GetMapping("/items")
    public String getItems() {
        return "Affichage des articles";
    }
}
```

<http://localhost:8080/item/items>

```
@RestController
@RequestMapping("/auth")
public class AuthController {

    @GetMapping("/message")
    public String getItems() {
        return "Vérification identité";
    }
}
```

<http://localhost:8080/auth/message>

WEBCIENT



Gradle implementation 'org.springframework.boot:spring-boot-starter-webflux'

WebClient est une interface permettant l'exécution de requêtes HTTP, nous pourrons ainsi faire communiquer notre API REST avec d'autres API REST.

GET

```
WebClient client = WebClient.create();
```

Obtention d'un auteur

```
Author response = client.get()
    .uri("URL/authors/1")
    .retrieve()
    .bodyToMono(Author.class)
    .block();
```

```
WebClient client = WebClient.create();
```

Obtention d'une liste d'auteurs

```
List<Author> response = client.get()
    .uri("URL/authors")
    .retrieve()
    .bodyToMono(new ParameterizedTypeReference<List<Author>>() {} )
    .block();
```

WEBCIENT

POST

```
WebClient client = WebClient.create();
```

```
client.post()  
    .uri("URL/authors")  
    .contentType(MediaType.APPLICATION_JSON)  
    .body(Mono.just(author), Author.class)  
    .retrieve()  
    .bodyToMono(Void.class)  
    .block();
```

Envoyer un auteur, sans retour attendu

```
WebClient client = WebClient.create();
```

```
String response = client.post()  
    .uri("URL/authors")  
    .contentType(MediaType.APPLICATION_JSON)  
    .body(Mono.just(author), Time.class)  
    .retrieve()  
    .bodyToMono(String.class)  
    .block();
```

Envoyer un auteur, avec un message de retour attendu

SWAGGER

Swagger est un ensemble d'outils permettant, entre autres, de documenter les APIs.



Gradle implementation group: 'io.springfox', name: 'springfox-boot-starter', version: '3.0.0'



application.properties spring.mvc.pathmatch.matching-strategy=*ant_path_matcher*

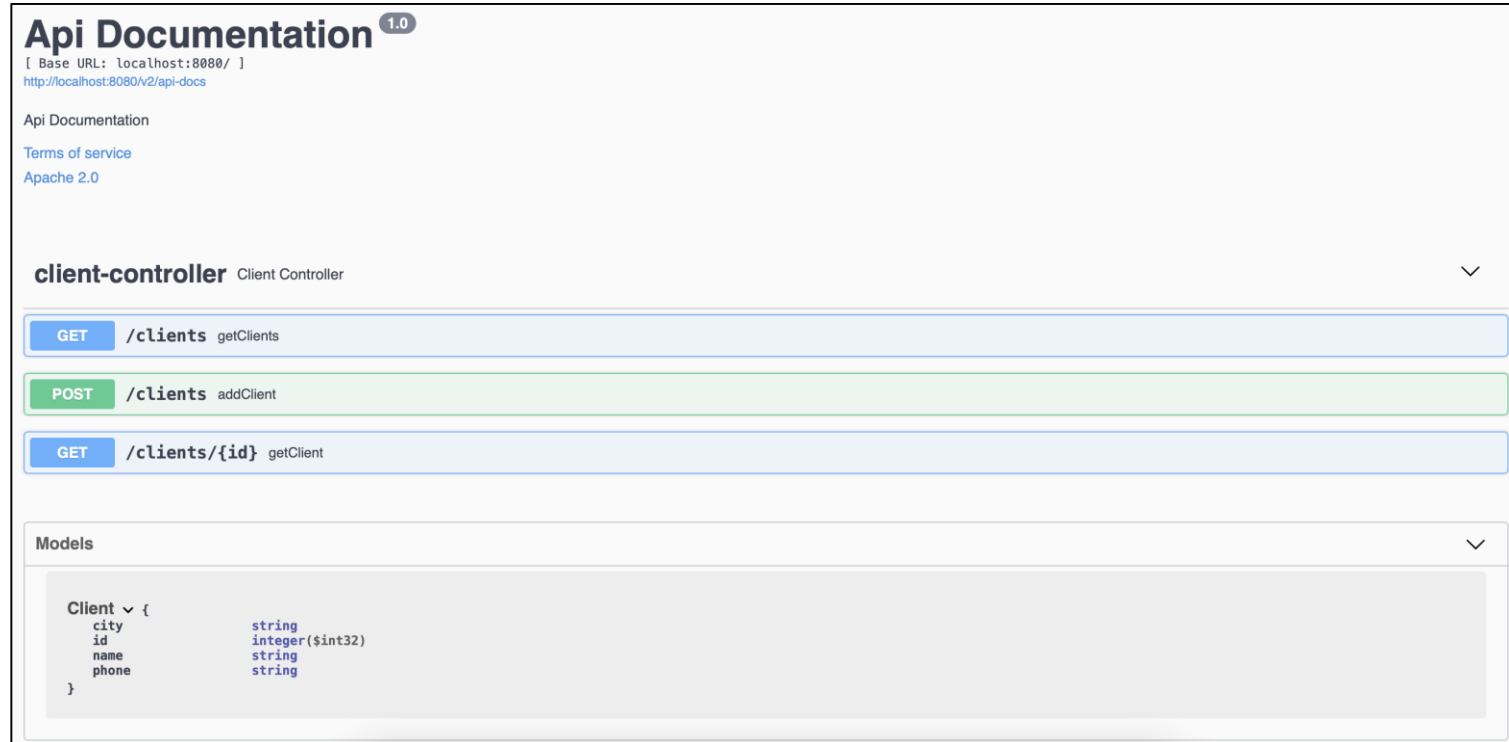
```
@SpringBootApplication
@EnableSwagger2
public class TdSpringApplication {
    public static void main(String[] args) {
        SpringApplication.run(TdSpringApplication.class, args);
    }
}
```

SwaggerConfig.java

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("fr.eseo.td_spring"))
            .paths(PathSelectors.any())
            .build();
    }
}
```


SWAGGER

Une fois l'API compilée et lancée, allez sur : <http://localhost:8080/swagger-ui/index.html>



The screenshot shows the Swagger API Documentation interface. At the top, it says "Api Documentation" with a version "1.0" badge. Below this, it lists the base URL as "localhost:8080/" and provides a link to "http://localhost:8080/v2/api-docs". There are also links for "Terms of service" and "Apache 2.0".

The main section is titled "client-controller" and "Client Controller". It lists three API endpoints:

- GET** `/clients` `getClients`
- POST** `/clients` `addClient`
- GET** `/clients/{id}` `getClient`

Below the endpoints, there is a "Models" section. It shows a "Client" model with the following fields:

```
Client {
  city      string
  id        integer($int32)
  name      string
  phone     string
}
```

Si votre API possède un système d'authentification, référez-vous à la documentation de Swagger :

<https://swagger.io/docs/specification/authentication/>



SPRING DATA

PRÉSENTATION D'HIBERNATE ET SPRING DATA


Un **ORM** (Object Relational Mapping) procure un niveau d'abstraction plus élevé que l'utilisation de JDBC pour assurer la persistance des objets dans un SGBD.

Un ORM fera l'interface entre l'application et la base de données relationnelle pour simuler une base de données orientée objet (grâce à une conversion de la base de données sous forme d'un graphe d'objet).

L'API **JPA** (Java Persistence API) est un standard d'ORM proposé par Java EE. Elle repose sur l'utilisation des annotations qui permettent de définir facilement des objets métier servant d'interface entre la base de données et l'application.



réduit la quantité de code standard requis par JPA.

 **HIBERNATE** est un framework de persistance qui va nous permettre de respecter JPA afin d'implémenter un ORM dans un environnement Java EE.

PRÉSENTATION D'HIBERNATE ET SPRING DATA

Une classe Java appelée Entité	↔	Une table SQL
Un attribut d'une classe	↔	Une colonne dans une table SQL
Une référence entre classes	↔	Une relation entre tables SQL
Un objet Java	↔	Un enregistrement dans une table SQL

Spring Data permet d'écrire des interfaces qui permettront de manipuler les entités d'une façon simplifiée.

Spring Data se base sur le design pattern DAO.



```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
runtimeOnly 'org.mariadb.jdbc:mariadb-java-client'
```

ENTITÉS

Déclaration de l'entité

Association à la table

L'attribut clé primaire est annoté par @Id

Pour produire automatiquement les valeurs d'identifiant :
 GenerationType.AUTO : géré par Hibernate
 GenerationType.IDENTITY : mécanisme du SGBD

```
@Entity
@Table(name = "author")
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(name = "firstName", length = 30)
    private String firstName;
    @Column(length = 20, nullable = false)
    private String lastName;

    public Author() {
    }

    public int getId() {
        return id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Attributs correspondant aux colonnes de la table, les attributs possibles sont :

name : précise le nom de la colonne dans la table si différent

length : indique la taille maximale de la valeur de la propriété

nullable (avec les valeurs false ou true) : indique si la colonne accepte des valeurs à NULL

...

Si vous définissez des constructeurs avec des paramètres vous devrez nécessairement définir un constructeur sans paramètres pour qu'Hibernate puisse utiliser la réflexivité.

Lorsqu'un attribut ne doit pas être persistant dans la base il faut l'annoter avec @Transient :

```
@Transient
private boolean isReady;
```

NB : le constructeur Author(int, String, String) a été supprimé car il ne servira plus pour boucher des données.

ENTITÉS

L'interface AuthorDao va directement gérer les données, elle sera donc annotée avec @Repository et AuthorDaoImpl peut être supprimée.

```
@Repository
public interface AuthorDao extends JpaRepository<Author, Integer> {

}
```

AuthorDao va hériter de la classe **JpaRepository** qui fournit un ensemble de méthodes pour interagir avec une base de données.

De nombreuses méthodes ([cf. documentation de JpaRepository](#)) sont déjà implémentées :

- findAll : lister toutes les lignes de la base de données
- findById : retrouver une ligne grâce à son ID
- deleteById : supprimer une ligne grâce à son ID
- count : retourner le nombre de lignes disponibles
- ...

ENTITÉS

```
@RestController
public class AuthorController {
    @Autowired
    private AuthorDao authorDao;

    @GetMapping("/authors")
    public List<Author> getAuthors() {
        return authorDao.findAll();
    }

    @GetMapping("/authors/{id}")
    public Optional<Author> getAuthor(@PathVariable int id) {
        return authorDao.findById(id);
    }

    @PostMapping("authors")
    public void addAuthor(@RequestBody Author author) {
        authorDao.save(author);
    }

    @DeleteMapping("/authors/{id}")
    public void deleteAuthor(@PathVariable int id) {
        if(authorDao.existsById(id)) {
            authorDao.deleteById(id);
        }
    }
}
```

Nous remplaçons la précédente déclaration de AuthorDao.

@Autowired indique à Spring qu'il doit réaliser une injection de dépendance.

Nous pouvons implémenter dans le contrôleur l'opération de suppression d'un auteur.

ENTITÉS

Les instructions suivantes, ajoutées au fichier index.html, permettront d'obtenir une IHM de la suppression.

```
<h1>Supprimer un auteur</h1>
<form action="javascript:deleteAuthor();">
  <input type="number" id="id" name="id" placeholder="Identifiant" required>
  <input type="submit" value="Valider">
</form>
<script>
  function deleteAuthor() {
    fetch('/authors/'+document.getElementById("id").value, {
      method: 'DELETE'
    })
    .then(response => {
      response.text();
      window.location.reload();
    })
  }
</script>
```


ENTITÉS

Vous pourrez aussi ajouter des opérations de modification des auteurs (PUT)

```
@PutMapping ("/authors")
public void updateAuthor(@RequestBody Author author) {
    authorDao.save(author);
}
```

Pour implémenter des requêtes spécifiques il est conseillé d'utiliser le JPQL ([wiki sur le sujet](#)).

AuthorController

```
@GetMapping ("/authors/lastnames")
public List<String> getLastNamesAuthors() {
    return authorDao.listAllLastNames();
}
```

AuthorDao

```
@Query("Select a.lastName FROM Author a")
List<String> listAllLastNames();
```

L'annotation @Modifying doit être ajoutée à @Query si la requête n'est pas en SELECT (e.g. INSERT, DELETE, ...)

RELATIONS

Relation de type 1 – 1 (unidirectionnelle)



```

@Entity
@Table(name="passeport")
public class Passeport {

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

    private String numero;

    @OneToOne
    private Personne proprietaire;

    public Passeport() {}

    public Passeport(String numero, Personne proprietaire) {
        this.numero = numero;
        this.proprietaire = proprietaire;
    }
}
  
```

id	numero	proprietaire_id
1	FR49BX	1

```

@Entity
@Table(name="personne")
public class Personne {

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

    private String nom;

    public Personne() {}

    public Personne(String nom) {
        this.nom = nom;
    }
}
  
```

id	nom
1	Paul

RELATIONS

Relation de type 1 – 1 (bidirectionnelle)



```

@Entity
@Table(name="passeport")
public class Passeport {

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

    private String numero;

    @OneToOne
    private Personne proprietaire;

    public Passeport() {}

    public Passeport(String numero, Personne proprietaire) {
        this.numero = numero;
        this.proprietaire = proprietaire;
    }
}
  
```

```

@Entity
@Table(name="personne")
public class Personne {

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

    private String nom;

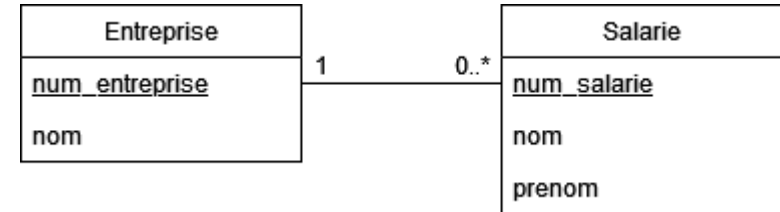
    @OneToOne(mappedBy = "proprietaire")
    private Passeport passeport;

    public Personne() {}

    public Personne(String nom) {
        this.nom = nom;
    }
}
  
```

RELATIONS

Relation de type 1 – n



```

@Entity
@Table(name="salarie")
public class Salarie {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "num_salarie")
    private int id;

    @Column(length = 30, nullable = false)
    private String nom;

    @Column(length = 30, nullable = false)
    private String prenom;

    public Salarie() {}

    public Salarie(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
}

```

```

@Entity
@Table(name="entreprise")
public class Entreprise {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "num_entreprise")
    private int id;

    @Column(length = 30, nullable = false)
    private String nom;

    @OneToMany(cascade = { CascadeType.PERSIST })
    @JoinColumn(name="num_entreprise")
    private List<Salarie> salaries;

    public Entreprise() {
        salaries = new ArrayList<>();
    }

    public Entreprise(String nom) {
        this();
        this.nom = nom;
    }





    public void ajouterSalarie(Salarie salarie) {
        salaries.add(salarie);
    }
}



```

Il est possible de propager des modifications à tout ou partie des entités liées. Les annotations permettant de spécifier une relation possèdent un attribut cascade permettant de spécifier les opérations concernées (ALL, DETACH, MERGE, PERSIST, REFRESH ou REMOVE).

RELATIONS

```
Salarie s1 = new Salarie("Martin", "Paul");  
Salarie s2 = new Salarie("Bonier", "Marie");  
Entreprise entreprise = new Entreprise("Google");  
entreprise.ajouterSalarie(s1);  
entreprise.ajouterSalarie(s2);  
entrepriseDao.save(entreprise);
```

 num_salarie	 nom	 prenom	 num_entreprise
1	Martin	Paul	1
2	Bonier	Marie	1

 num_entreprise	 nom
1	Google

RELATIONS

Relation de type n – 1

```
@Entity
@Table(name="salarie")
public class Salarie {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "num_salarie")
    private int id;

    @Column(length = 30, nullable = false)
    private String nom;

    @Column(length = 30, nullable = false)
    private String prenom;

    @ManyToOne(cascade = {CascadeType.ALL})
    @JoinColumn(name="num_entreprise")
    private Entreprise entreprise;

    public Salarie() {}

    public Salarie(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }

    public void ajouterEntreprise(Entreprise entreprise) {
        this.entreprise = entreprise;
    }
}
```

```
@Entity
@Table(name="entreprise")
public class Entreprise {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "num_entreprise")
    private int id;

    @Column(length = 30, nullable = false)
    private String nom;





    public Entreprise() {}



    public Entreprise(String nom) {
        this();
        this.nom = nom;
    }
}
```

RELATIONS

Relation de type n – 1

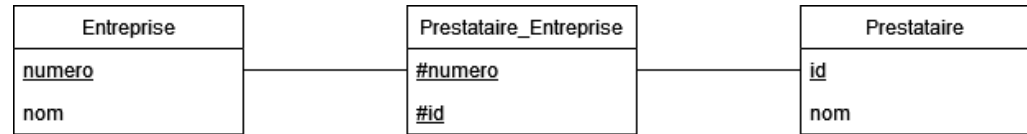
```
Entreprise entreprise = new Entreprise("Google");  
Salarie s1 = new Salarie("Martin", "Paul");  
Salarie s2 = new Salarie("Bonier", "Marie");  
s1.ajouterEntreprise(entreprise);  
s2.ajouterEntreprise(entreprise);  
salarieDao.save(s1);  
salarieDao.save(s2);
```

 num_salarie ▾	 nom ▾	 prenom ▾	 num_entreprise ▾
1	Martin	Paul	1
2	Bonier	Marie	1

 num_entreprise ▾	 nom ▾
1	Google

RELATIONS

Relation de type n – n



```

@Entity
@Table(name="entreprise")
public class Entreprise {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int numero;

    @Column(length = 30, nullable = false)
    private String nom;

    @ManyToMany(mappedBy = "entreprises", cascade = CascadeType.ALL)
    private Set<Prestataire> prestataires;

    public Entreprise() {
        prestataires = new HashSet<>();
    }

    public Entreprise(String nom) {
        this();
        this.nom = nom;
    }

    public void ajouterPrestataire(Prestataire prestataire) {
        prestataires.add(prestataire);
    }
}
  
```

```

@Entity
@Table(name="prestataire")
public class Prestataire {

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

    @Column(length = 30, nullable = false)
    private String nom;

    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name = "prestataire_entrepise",
        joinColumns = @JoinColumn(name = "id"),
        inverseJoinColumns = @JoinColumn(name = "numero"))
    private Set<Entreprise> entreprises;

    public Prestataire() {
        entreprises = new HashSet<>();
    }

    public Prestataire(String nom) {
        this();
        this.nom = nom;
    }

    public void ajouterEntreprise(Entreprise entreprise) {
        entreprises.add(entreprise);
    }
}
  
```


RELATIONS

Relation de type n – n

```

Entreprise e1 = new Entreprise("Google");
Entreprise e2 = new Entreprise("Apple");
Prestataire p1 = new Prestataire("Martin");
Prestataire p2 = new Prestataire("Bonier");
Prestataire p3 = new Prestataire("Dupont");

```

```

p1.ajouterEntreprise(e1);
p1.ajouterEntreprise(e2);

```

```

p3.ajouterEntreprise(e2);

```

```

prestataireDao.save(p1);
prestataireDao.save(p2);
prestataireDao.save(p3);

```

id	nom
1	Martin
2	Bonier
3	Dupont

numero	nom
1	Google
2	Apple

id	numero
1	1
1	2
3	2

RELATIONS

Héritage



Une table contenant tous les attributs de Vehicule, ceux spécifiques à Avion et à Voiture, ainsi que la colonne discriminante, sera créée.

```
@Entity
@Inheritance(strategy= InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="vehicule_type")
abstract class Vehicule implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected int id;

    @Column(length = 30, nullable = false)
    protected String modele;

    public Vehicule() {}

    public Vehicule(String modele) {
        this.modele = modele;
    }
}
```

Permet de préciser la valeur dans la colonne discriminante identifiant un objet du type de cette classe. Cette valeur doit être unique pour l'ensemble des classes de l'héritage.

```
@Entity
@DiscriminatorValue("V")
public class Voiture extends Vehicule {

    public Voiture() {
        super();
    }

    public Voiture(String modele) {
        super(modele);
    }
}
```

```
@Entity
@DiscriminatorValue("A")
public class Avion extends Vehicule {




    public Avion() {
        super();
    }

    public Avion(String modele) {
        super(modele);
    }
}
```

RELATIONS

Héritage

```
Voiture voiture1 = new Voiture("Peugeot 508");  
Voiture voiture2 = new Voiture("Tesla Model X");  
Avion avion = new Avion("Airbus A380");  
voitureDao.save(voiture1);  
voitureDao.save(voiture2);  
avionDao.save(avion);
```

 vehicule_type	 id	 modele
V	1	Peugeot 508
V	2	Tesla Model X
A	3	Airbus A380

La stratégie SINGLE_TABLE permet de représenter en base de données un héritage avec une seule table, mais il existe d'autres stratégies (JOINED et TABLE_PER_CLASS).



SPRING SECURITY

SÉCURISATION



```
implementation 'org.springframework.boot:spring-boot-starter-security'  
implementation group: 'io.jsonwebtoken', name: 'jjwt', version: '0.9.1'
```

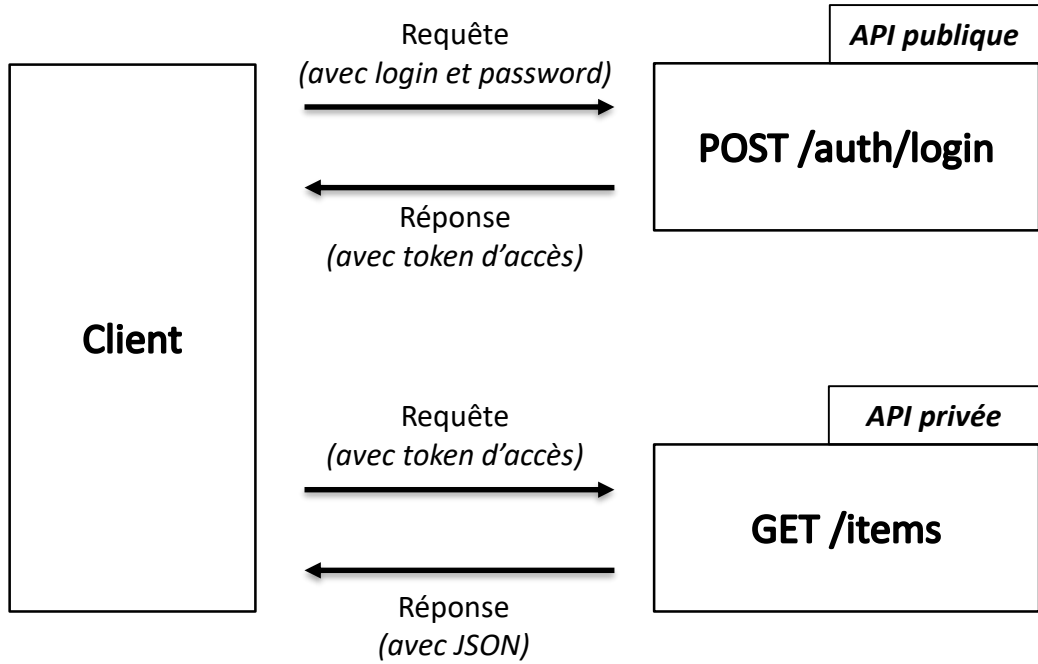


User
<u>login</u>
password



Pensez à créer une base de données nommée spring_security

- src
 - main
 - java
 - fr.eseo.token
 - controller
 - AuthController
 - ItemController
 - dao
 - ItemDao
 - UserDao
 - model
 - Item
 - User
 - security
 - ApplicationSecurity
 - AuthRequest
 - AuthResponse
 - JwtTokenFilter
 - JwtTokenUtil
 - TokenApplication



SÉCURISATION

application.properties

```
server.port=8080

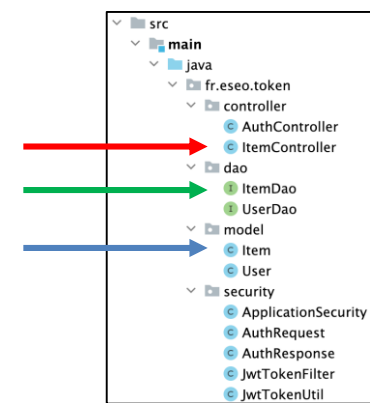
spring.datasource.url=jdbc:mariadb://localhost:3306/spring_security
spring.datasource.username=root
spring.datasource.password=MySQL2021
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.jpa.database-platform = org.hibernate.dialect.MySQL5Dialect
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto = update

app.jwt.secret=customizeMe
```

Un token JWT permet d'échanger des informations de manière sécurisée.

Afin de renforcer la sécurité, le token sera généré à l'aide d'une clé secrète stockée dans appsettings.json

SÉCURISATION



```

@Entity
@Table(name = "items")
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, length = 100)
    private String name;

    private float price;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }
}
  
```

```

@RestController
@CrossOrigin
@RequestMapping("/items")
public class ItemController {

    @Autowired
    private ItemDao itemDao;

    @GetMapping
    public List<Item> list() {
        return itemDao.findAll();
    }

    @PostMapping
    public void addItem(@RequestBody Item item) {
        itemDao.save(item);
    }
}
  
```

```

@Repository
public interface ItemDao extends JpaRepository<Item,Integer> {

}
  
```

4 – SPRING SECURITY

```

@Entity
@Table(name = "users")
public class User implements UserDetails {
    @Id
    @Column(length = 30, nullable = false)
    private String login;

    @Column(length = 60, nullable = false)
    private String password;
  
```

```

    public User() {}
  
```

```

    public User(String login, String password) {
        this.login = login;
        this.password = password;
    }
  
```

```

    // Ajouter les getters et setters
  
```

```

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return null;
    }
  
```

```

    @Override
    public String getUsername() {
        return this.login;
    }
  
```

```

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }
  
```

```

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
  
```

```

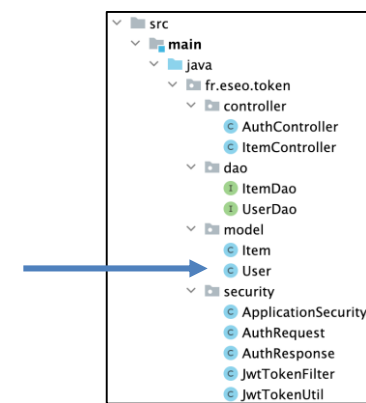
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }
  
```

```

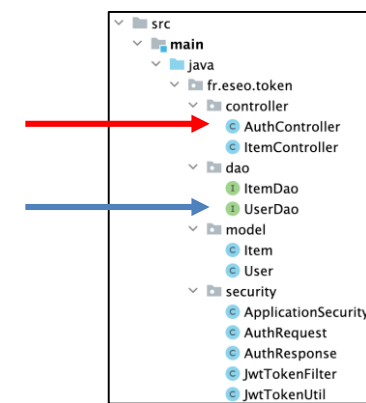
    @Override
    public boolean isEnabled() {
        return true;
    }
  
```

```

}
  
```



SÉCURISATION



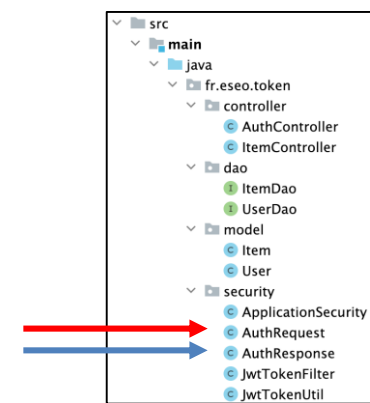
```
@Repository
public interface UserDao extends JpaRepository<User, String> {
    Optional<User> findByLogin(String login);
}
```

```
@RestController
@CrossOrigin
@RequestMapping("/auth")
public class AuthController {
    @Autowired
    AuthenticationManager authenticationManager;

    @Autowired
    JwtTokenUtil jwtUtil;

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody AuthRequest request) {
        try {
            Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                    request.getLogin(), request.getPassword()
                )
            );
            User user = (User) authentication.getPrincipal();
            String accessToken = jwtUtil.generateAccessToken(user);
            AuthResponse response = new AuthResponse(user.getLogin(), accessToken);
            return ResponseEntity.ok().body(response);
        } catch (BadCredentialsException ex) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
        }
    }
}
```

SÉCURISATION



```
public class AuthResponse {

    private String login;
    private String accessToken;

    public AuthResponse() {}

    public AuthResponse(String login, String accessToken) {
        this.login = login;
        this.accessToken = accessToken;
    }

    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public String getAccessToken() {
        return accessToken;
    }

    public void setAccessToken(String accessToken) {
        this.accessToken = accessToken;
    }

}
```

```
public class AuthRequest {

    private String login;
    private String password;

    public String getLogin() {
        return login;
    }

    public void setLogin(String login) {
        this.login = login;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

}
```

4 – SPRING SECURITY

Indique à Spring de chercher des injections de dépendance

← @Component

```
public class JwtTokenFilter extends OncePerRequestFilter {
    @Autowired
    private JwtTokenUtil jwtUtil;

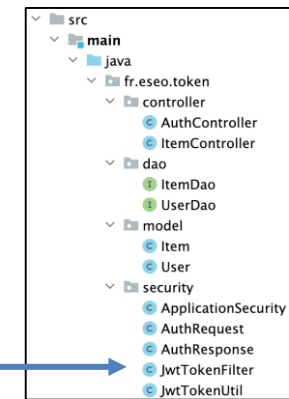
    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        if (!hasAuthorizationBearer(request)) {
            filterChain.doFilter(request, response);
        } else {
            String token = getAccessToken(request);
            if (!jwtUtil.validateAccessToken(token)) {
                filterChain.doFilter(request, response);
            } else {
                setAuthenticationContext(token, request);
                filterChain.doFilter(request, response);
            }
        }
    }

    private boolean hasAuthorizationBearer(HttpServletRequest request) {
        String header = request.getHeader("Authorization");
        return !ObjectUtils.isEmpty(header) && header.startsWith("Bearer");
    }

    private String getAccessToken(HttpServletRequest request) {
        String header = request.getHeader("Authorization");
        return header.split(" ")[1].trim();
    }

    private void setAuthenticationContext(String token, HttpServletRequest request) {
        UserDetails userDetails = getUserDetails(token);
        UsernamePasswordAuthenticationToken
            authentication = new UsernamePasswordAuthenticationToken(userDetails, null, null);
        authentication.setDetails(
            new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(authentication);
    }

    private UserDetails getUserDetails(String token) {
        User userDetails = new User();
        String[] jwtSubject = jwtUtil.getSubject(token).split(",");
        userDetails.setLogin(jwtSubject[0]);
        return userDetails;
    }
}
```



SÉCURISATION

@Component

```
public class JwtTokenUtil {
    private static final long EXPIRE_DURATION = 24 * 60 * 60 * 1000;

    @Value("${app.jwt.secret}")
    private String SECRET_KEY;

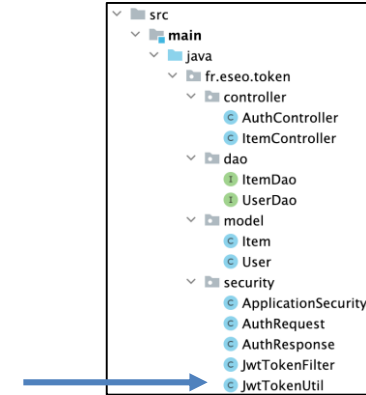
    public String generateAccessToken(User user) {
        return Jwts.builder()
            .setSubject(String.format("%s", user.getLogin()))
            .setIssuer("ESEO")
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + EXPIRE_DURATION))
            .signWith(SignatureAlgorithm.HS512, SECRET_KEY)
            .compact();
    }

    public boolean validateAccessToken(String token) {
        try {
            Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return false;
    }

    public String getSubject(String token) {
        return parseClaims(token).getSubject();
    }

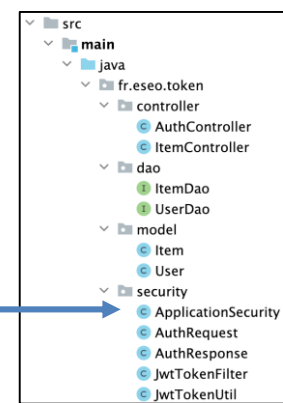
    private Claims parseClaims(String token) {
        return Jwts.parser()
            .setSigningKey(SECRET_KEY)
            .parseClaimsJws(token)
            .getBody();
    }
}
```

Permet d'injecter une valeur par défaut



4 – SPRING SECURITY

Permet de déclarer une classe servant à la configuration



```

@Configuration
public class ApplicationSecurity {
    @Autowired
    private UserDao userDao;
    @Autowired
    private JwtTokenFilter jwtTokenFilter;

    @Bean
    public UserDetailsService userDetailsService() {
        return username -> userDao.findByLogin(username)
            .orElseThrow(
                () -> new UsernameNotFoundException("User " + username + " not found");
            );
    }
  
```

Permet de déclarer une injection de dépendance

```

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

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

```

@Bean
public SecurityFilterChain configure(HttpSecurity http) throws Exception {
    http.csrf().disable();
    http.cors();
    http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.authorizeRequests()
        .antMatchers("/auth/login").permitAll()
        .anyRequest().authenticated();
    http.exceptionHandling()
        .authenticationEntryPoint(
            (request, response, ex) -> response.sendError(
                HttpServletResponse.SC_UNAUTHORIZED,
                ex.getMessage()
            )
        );
    http.addFilterBefore(jwtTokenFilter, UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
  
```

SÉCURISATION

Pour tester le tout...

```
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
@Rollback(false)
public class UserDaoTests {

    @Autowired
    private UserDao userDao;

    @Autowired
    private ItemDao itemDao;

    @Test
    public void testCreateUser() {
        BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
        String password = passwordEncoder.encode("b3");

        User newUser = new User("eseo", password);
        userDao.save(newUser);
    }

    @Test
    public void testCreateItemss() {

        Item phone = new Item();
        phone.setName("Pixel 6");
        phone.setPrice(600);

        Item laptop = new Item();
        laptop.setName("MacBook Pro 13");
        laptop.setPrice(1590);

        itemDao.save(phone);
        itemDao.save(laptop);
    }
}
```

Créez une classe de « test » qui va être chargée de remplir les deux tables avec des données.

```

<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8">
    <title>Sécurité / Token</title>
  </head>
  <script>
    function checkUser() {
      fetch("http://localhost:8080/auth/login", {
        method: 'POST',
        headers: {
          Authentication: 'Bearer Token',
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          "login":document.getElementById("login").value,
          "password":document.getElementById("password").value,
        })
      })
      .then(response => response.json())
      .then(data => {
        document.getElementById("password").style.backgroundColor = "#ffffff";
        sessionStorage.setItem("accessToken", data.accessToken);
      })
      .catch(err => {
        document.getElementById("password").value = "";
        document.getElementById("password").style.backgroundColor = "#ff0000";
      })
    }

    function getItems() {
      fetch("http://localhost:8080/items", {
        headers: { 'Authorization': 'Bearer ' + sessionStorage.getItem("accessToken") }
      })
      .then(response => response.json())
      .then(data => {
        document.getElementById("items").innerHTML = '<table style="width:50%"><thead><td>' +
          'Nom</td><td>Prix</td></thead><tbody id="listItems">';
        data.forEach(element => {
          let row = '<tr><td>' + element.name + '</td>' +
            '<td>' + element.price + '</td></tr>';
          document.getElementById("listItems").innerHTML += row;
        })
        document.getElementById("items").innerHTML += '</table>';
      })
      .catch(err => document.getElementById("items").innerHTML = "Vous n'êtes pas identifié.")
    }
  </script>
  <body>
    <form action="javascript:checkUser()">
      <input type="text" id="login" name="login" placeholder="Login" required>
      <input type="password" id="password" name="password" placeholder="Password" required>
      <input type="submit" value="Valider">
    </form>
    <button onclick="getItems()">Lister les articles</button>
    <div id="items"></div>
  </body>
</html>

```

DÉPLOIEMENT SUR UN SERVEUR

Ajouter dans le fichier build.gradle l'instruction war :

```
plugins {  
    id 'org.springframework.boot' version '2.7.1'  
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'  
    id 'java'  
    id 'war' ←  
}
```

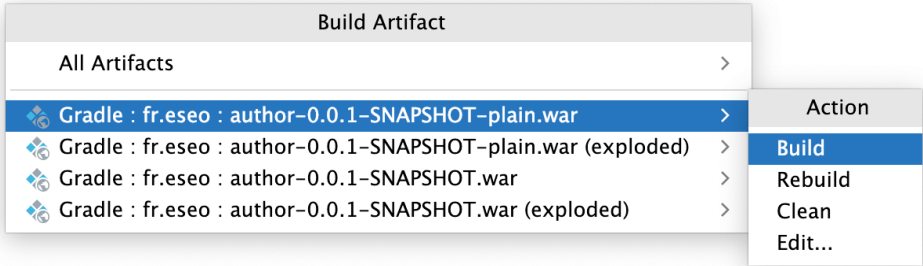
Créez une classe **ServletInitializer** dans la paquetage fr.eseo.author contenant :

```
package fr.eseo.author;  
  
import org.springframework.boot.builder.SpringApplicationBuilder;  
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;  
  
public class ServletInitializer extends SpringBootServletInitializer {  
    @Override  
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {  
        return application.sources(AuthorApplication.class);  
    }  
}
```

Cette classe hérite de SpringBootServletInitializer qui est une interface permettant d'exécuter l'application via un WAR.

DÉPLOIEMENT SUR UN SERVEUR

Générez le war : **Build** → **Build Artifacts...**



Chargez sur le serveur le WAR à déployer.

