

# REST API

## Conception, Architecture et sécurité.



# Tour de table des stagiaires.



- Nom & prénom
- Expériences
- Précisez vos attentes pour cette formation

# Présentation du formateur - Didier Curvier MALEMBE.

---

- **Ingénieur Logiciel en ESN**
  - > Capgemini Technology Services
    - > Projet SNCF Connect
    - > Projet RM (Revenue Management) , etc...
- **Consultant Freelance**
  - > Secteur public / TPE-PME
- **Formateur partenaire**
  - > ORSYS Formation
  - > FITEC
  - > AFPA
  - > PMN
  - > Cyborg Intelligence - IB Cegos...
- **Co-founder ELITIS CONSULTING**

# Les objectifs pédagogiques

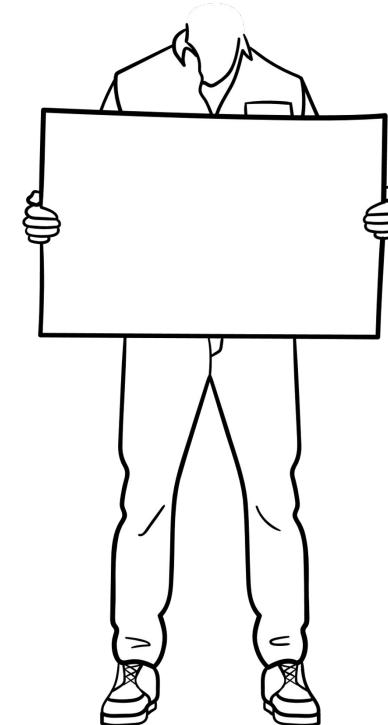
À l'issue de la formation, le participant sera en mesure de :

- Prendre en main les outils qui vous accompagneront de la conception au déploiement et la supervision de vos APIs
- Appréhender les menaces auxquelles s'exposent vos API
- Identifier les vulnérabilités les plus fréquentes
- Repérer les points faibles d'une API puis la protéger
- Maîtriser les bonnes pratiques de conception, de développement et d'architecture des APIs ReST

# Table des matières

---

- I. Introduction aux APIs ReST
- II. Bonnes pratiques
- III. La boîte à outils
- IV. Rappels sur la sécurité
- V. Authentification et autorisation
- VI. Middleware et JWT (JSON Web Token)
- VII. Les Tests d'API
- VIII. API Management





# Chapitre 1 : Introduction aux APIs REST.

Architecture n-tiers – API REST – HATEOAS Gestion des ressources et liens hypermédias.

## L'architecture n-tiers.

---

L'approche **client-serveur** à deux niveaux a été pendant longtemps la principale solution préconisée pour la construction d'applications. Aujourd'hui, **les architectures n-tiers**, grâce à leur maturité sont devenues une alternative efficientes pour la réalisation d'applications robustes et complexes.

**Les architectures n-tiers** ont émergées en réponse à la complexité croissante des besoins des entreprises et des technologies disponibles.

Elles offrent une séparation claire des préoccupations, **une meilleure gestion des ressources, une scalabilité accrue et une facilité de maintenance.**

## L'architecture n-tiers.

---

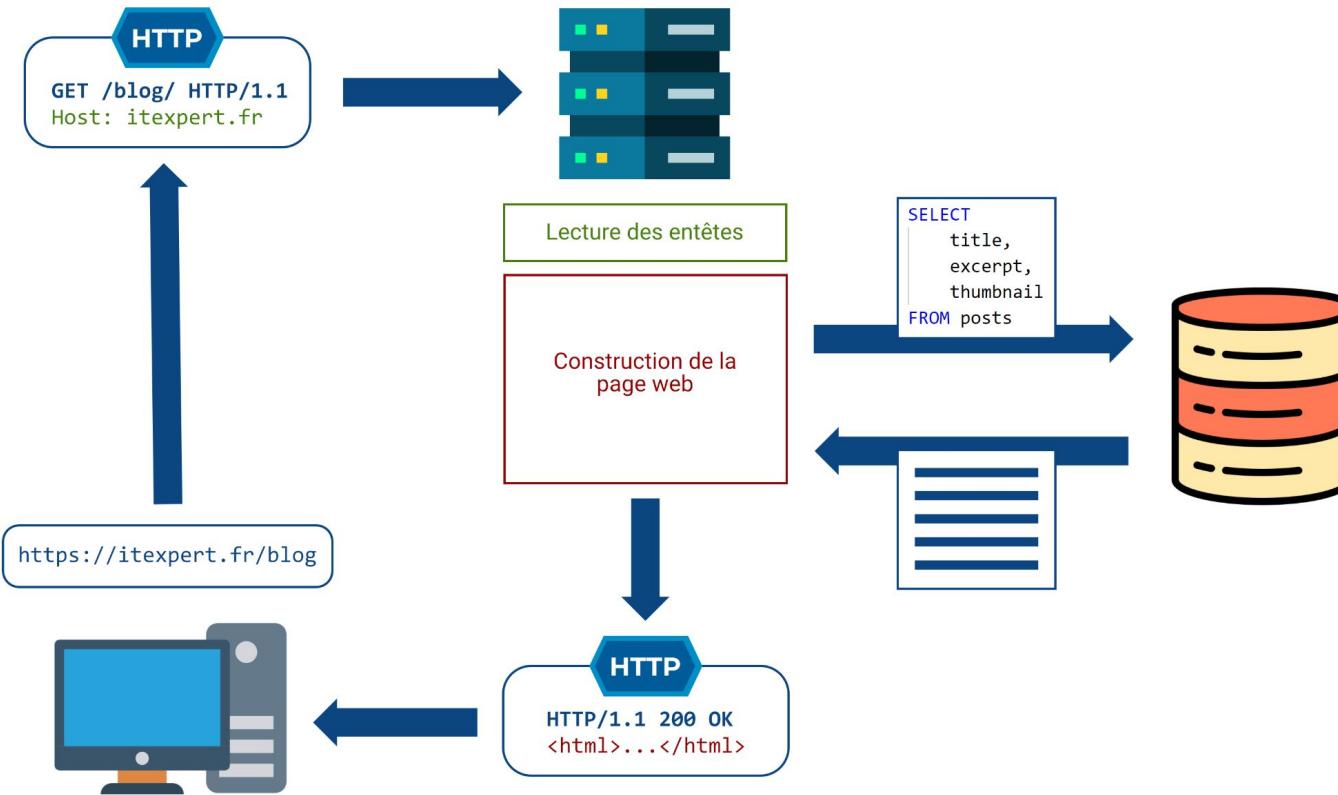
**L'architecture n-tiers** (tier : étage,niveau) , encore appelée multi-tiers, une architecture client-serveur dans laquelle une application est exécutée par plusieurs composants logiciels distincts.

Considérons par exemple les architectures 3 tiers :

- **Tier présentation** : application en charge de la présentation des données à l'utilisateur (interface utilisateur sur PC, interface web, ... )
- **Tier traitement** : application serveur qui contient la logique de gestion (les traitements sur les données) et accèdent aux données stockées dans les bases de données.
- **Tier base de données** : Serveurs de bases de données.

# Architecture n-tiers.

---

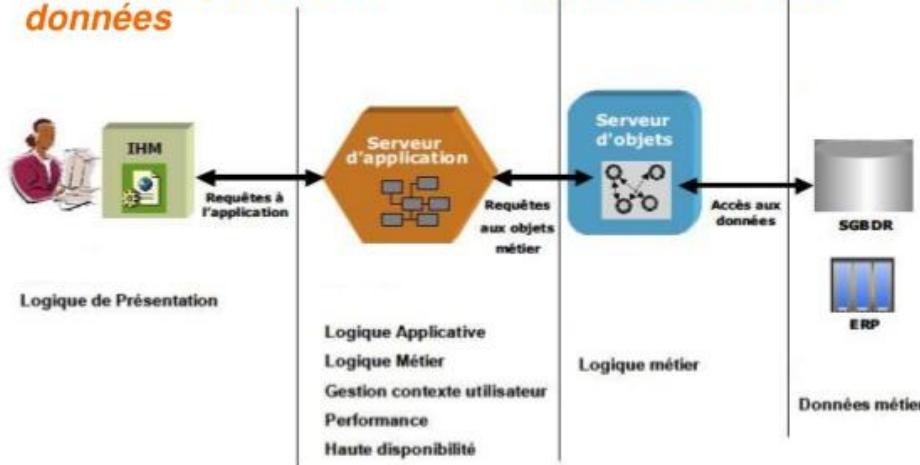


# Architecture n-tiers.

## Les couches de l'arch. n-tiers

6

- Une architecture n-tiers comprend généralement une **couche de présentation**, une **couche applicative**, une **couche objets métier** et une **couche d'accès aux données**



## Les avantages et inconvénients des architectures n-tiers.

---

Les architectures n-tiers présentent de nombreux avantages :

- La réutilisation des services existants.
- Couplage faible entre les services: Ce qui permet de faire évoluer les services un par un sans modification du reste de l'application.
- Modularité et indépendance technologique et topologique de chaque niveau.

Les inconvénients :

- Nécessité de gérer l'interopérabilité.
- Utilisation de framework / outils supplémentaires.

## Qu'est-ce qu'une API ?

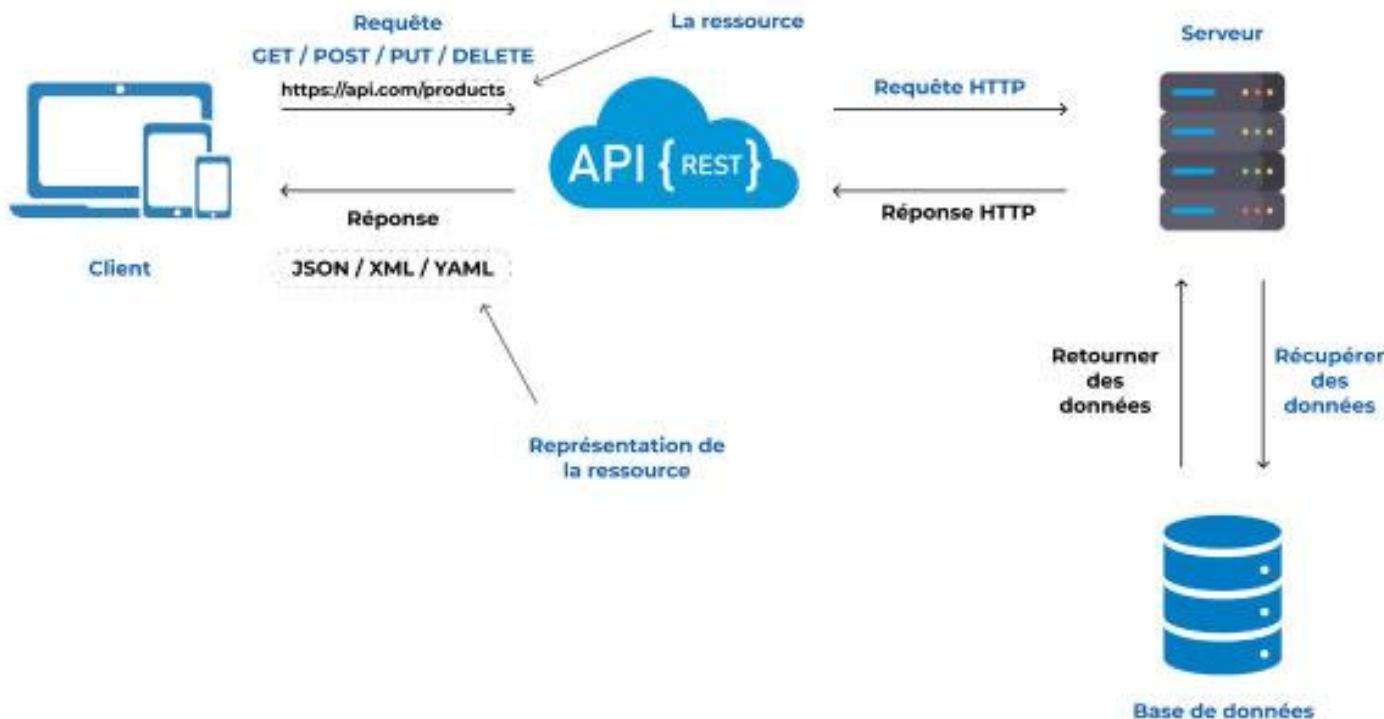
---

Le terme **API** est l'acronyme de “Application Programming Interface” qui signifie “Interface de programmation applicative”.

En termes simples, les APIs sont des “passerelles” qui permettent simplement à deux ou plusieurs applications (ex. *application mobile ou site Internet*) de **communiquer entre eux** et de permettre et **faciliter les échanges de données**.

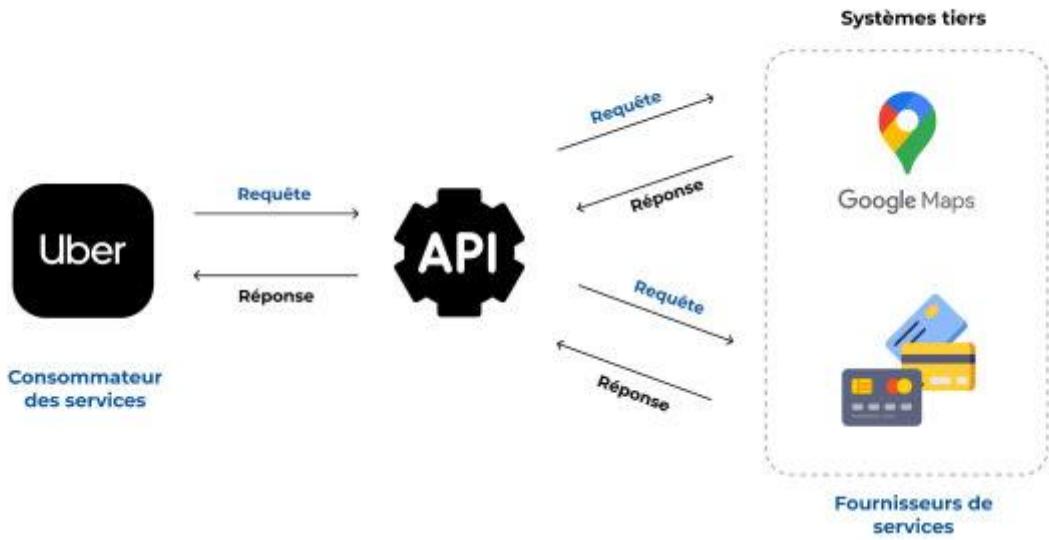
# API - Application Programming Interface.

---



# Cas d'utilisation API.

---



Fonctionnement de l'**application Uber** avec API Source.

# Les API REST.

---

- **REST :**
  - REST est un **style architectural** pour les services web. Il se concentre sur les ressources et la manière dont l'état de ces ressources est transféré sur le réseau.
  - Les services **REST** sont conçus autour de ressources (comme des données ou des objets) et sont accessibles via des **URI (Uniform Resource Identifiers)**.
  - REST utilise des méthodes **HTTP standard (GET, POST, PUT, DELETE, etc.)** et est étatique du point de vue du client (le serveur ne conserve pas l'état du client entre les requêtes).

# Les API REST.

---

- **REST :**
  - Utilise principalement **HTTP/HTTPS**.
  - L'accent est mis sur l'utilisation des méthodes **HTTP** standards et des codes d'état pour communiquer l'état des requêtes.
  - Flexible en termes de formats de données (**JSON, XML, HTML, texte brut, etc.**).
  - **JSON** est le format le plus couramment utilisé en raison de sa simplicité et de sa légèreté.

# Différences entre API REST et API SOA.

	<b>API REST</b>	<b>SOA(Service-Oriented Architecture)</b>
<b>Définition</b>	Modèle architectural conçu pour les services Web basés sur HTTP	Approche architecturale plus large basée sur l'intégration de services autonomes.
<b>Principes fondamentaux</b>	Basé sur les principes du REST : ressources, verbes HTTP, stateless.	Centré sur des principes comme les services indépendants et les contrats explicites.
<b>Technologies utilisées</b>	Principalement HTTP, JSON, XML, ou autres formats légers.	Utilise souvent des protocoles comme SOAP, WSDL, XML.
<b>Granularité</b>	Souvent plus fine, expose des ressources spécifiques.	Souvent plus grossière, expose des services complexes.

# Différences entre API REST et API SOA.

---

	<b>API REST</b>	<b>SOA(Service-Oriented Architecture)</b>
<b>Facilité d'intégration</b>	Plus simple grâce aux standards du Web.	Peut nécessiter plus d'efforts à cause des outils et protocoles spécifiques.
<b>Performance</b>	Haute performance grâce à sa légèreté et son optimisation HTTP.	Peut être plus lent à cause des surcharges liées à SOAP et XML.
<b>Cas d'usage principal</b>	Applications Web, microservices, systèmes mobiles.	Intégration d'applications complexes, systèmes d'entreprise (ERP, CRM).
<b>Gestion de sécurité</b>	Basée sur HTTPS, OAuth, JWT, etc.	Utilise des mécanismes comme WS-Security pour des scénarios complexes.

# Quelles sont les exigences d'un projet informatique ?



# **Exigences d'un projet informatique.**

---

## **Exigences fonctionnelles**

- Satisfaction des attentes fonctionnelles (besoins métiers).

## **Exigences techniques**

- Performance
- Sécurité
- Maintenabilité
- Gestion des données
- Portabilité

# Comment surmonter ces contraintes ?



## Amélioration par empirisme.

---

- Recourir à une architecture d'entreprise (**JEE,...**) pour bâtir l'application.
- Utiliser un **framework**.
  - pour la création d'applications côté client (Angular, VueJS,...)
  - pour la création d'application côté serveur (**Node JS, Spring-boot,...**)
  - pour le mapping objet-relationnel
- Utiliser des modèles de conception confirmés (**design pattern**).
- Utiliser un **middleware** pour des applications éprouvées.

# Programmation orientée aspect.

KESAKO ?



## La programmation orientée aspect.

---

Paradigme de programmation qui permet de traiter séparément les préoccupations transversales qui relèvent de la technique des préoccupations "métiers" qui sont au cœur de l'application.

Ce paradigme peut être appliqué sur des langages tels que le **JAVA**, **C++** ou le **C#**.

## **La programmation orientée aspect.**

---

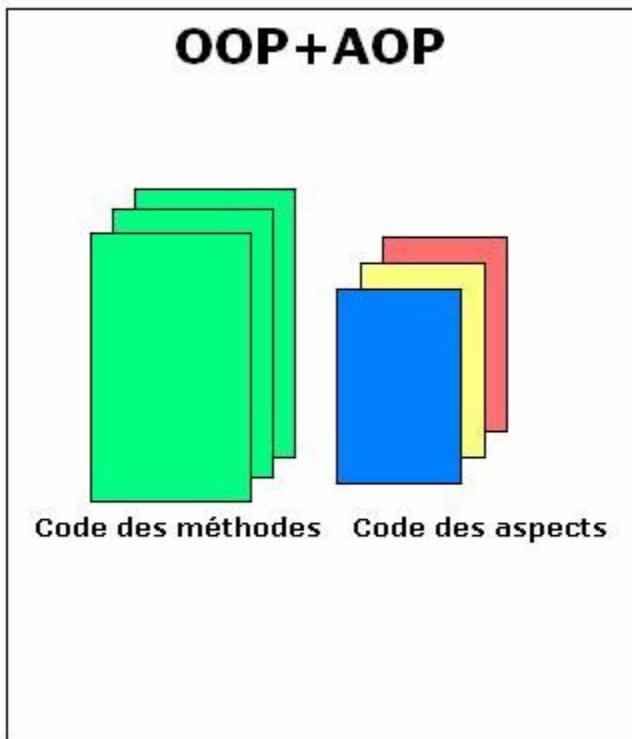
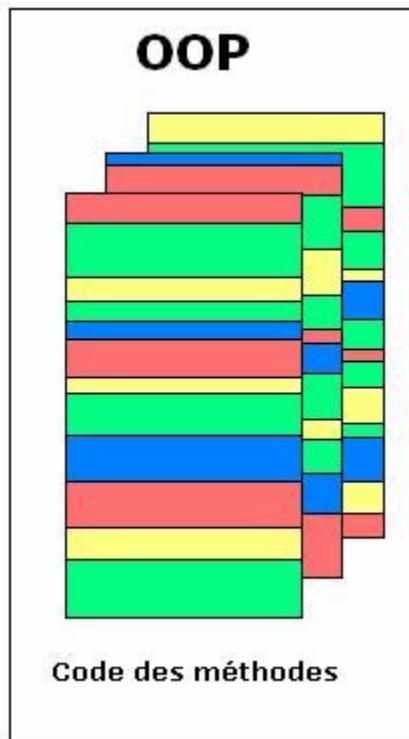
L'objectif de la **POA** n'est pas de se substituer aux autres paradigmes de programmation tel que l'orienté objet mais de les améliorer pour résoudre les problématiques liées à ces paradigmes.

La POA présente les avantages suivants :

- Gain de productivité
- Amélioration de la lisibilité du code

# La programmation orientée aspect.

---



# Exemple sans inversion de contrôle.

---

```
public void virement(int c1, int c2, double mt) {  
    /* Création d'une transaction */  
    EntityTransaction transaction=entityManager.getTransaction();  
    /* Démarrer la transaction */  
    transaction.begin();  
    try {
```

Code Technique

```
    /* Code métier */  
    retirer(c1,mt);  
    verser(c2,mt);
```

Code Métier

```
    /* Valider la transaction */  
    transaction.commit();  
} catch (Exception e) {  
    /* Annuler la transaction en cas d'exception */  
    transaction.rollback();  
    e.printStackTrace();  
}
```

Code Technique

## Exemple avec inversion de contrôle.

---

```
@Transactional  
public void virement(int c1, int c2, double mt) {  
    retirer(c1,mt);  
    verser(c2,mt);  
}
```

Code Métier

L'annotation **@Transactional**, permet de déléguer la gestion des transactions au conteneur Spring IOC

## Inversion de contrôle et Injection de dépendances.

---

L'inversion de contrôle est le design pattern commun à tous les frameworks .

Le principe sous-jacent est le suivant :

**Le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du framework.**

L'injection de dépendance est l'une des formes les plus connues d'inversion de contrôle.

## Qu'est-ce qu'un framework ?

---

- Le framework prend en charge l'essentiel des aspects techniques, ce qui permet aux programmeurs de se concentrer sur les aspects métier.
- Il existe des frameworks de toutes sortes. On en trouve notamment pour la réalisation de tests unitaires (*JUnit, Jasmin, etc.*), pour la persistance des données (*JPI, Hibernate, Entity Framework, etc.*) et, bien sûr, pour la réalisation d'applications.

## Qu'est-ce qu'un framework ?

---

Un framework d'application est un framework qui dispose d'une classe qui représente une application et qui en fournit une implémentation de base.

La réalisation d'une application consiste alors, pour l'essentiel, à étendre d'une manière ou d'une autre l'application de base fournie par le framework.

Cela peut se faire à l'aide de l'héritage ou par tout autre moyen.

# A la découverte du framework **SPRING-BOOT**



## Spring-Boot KESAKO ?

---

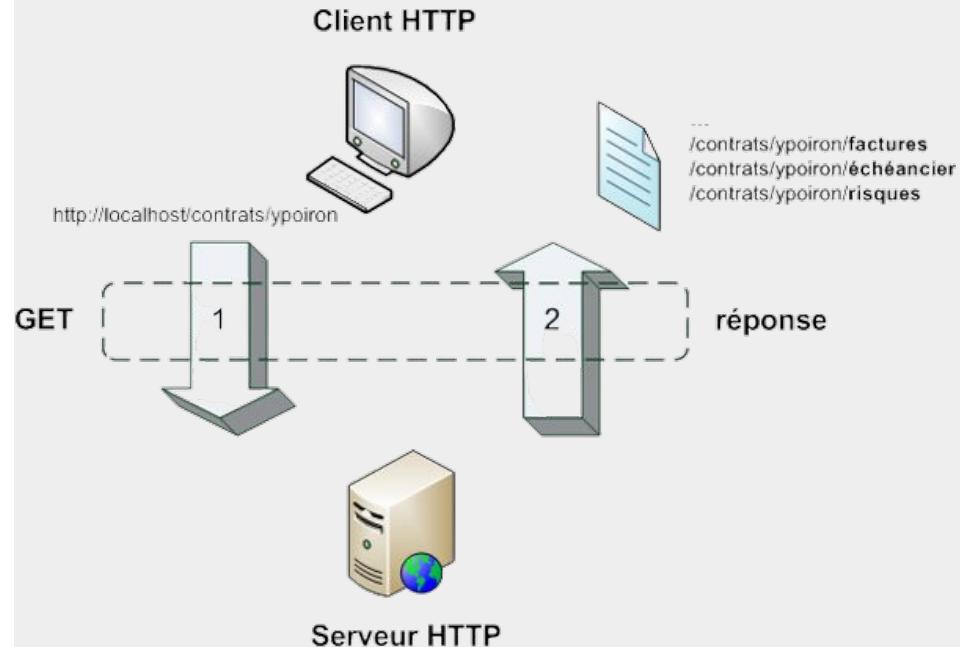
Spring Boot est un **framework** de développement applicatif Java open source.

Il est particulièrement recommandé pour le développement **d'API**.

Ses atouts sont nombreux :

- Facilite le développement d'applications complexes
- Facilite l'extrême injection de dépendances
- Réduire au maximum les fichiers de configuration
- Faciliter la gestion des dépendances Maven
- Créer des applications autonomes
- Fournir un conteneur embarqué (Tomcat)

# Architecture des API



## **SOAP (Simple Object Access Protocol)**

---

**Simple Object Access Protocol (SOAP)** est un protocole d'échange d'informations dans un environnement distribué basé sur **XML**.

**SOAP** prend en charge de nombreux protocoles (**HTTP, SMTP, POP**) et se compose de trois parties : Les enveloppes SOAP (ou Message), les règles d'encodage et la représentation **RPC**.

## REST (Representational State Transfer)

---

**REST (Representational State Transfer)** est un style d'architecture de services web fortement basé sur le protocole **HTTP** et le principe client-serveur.

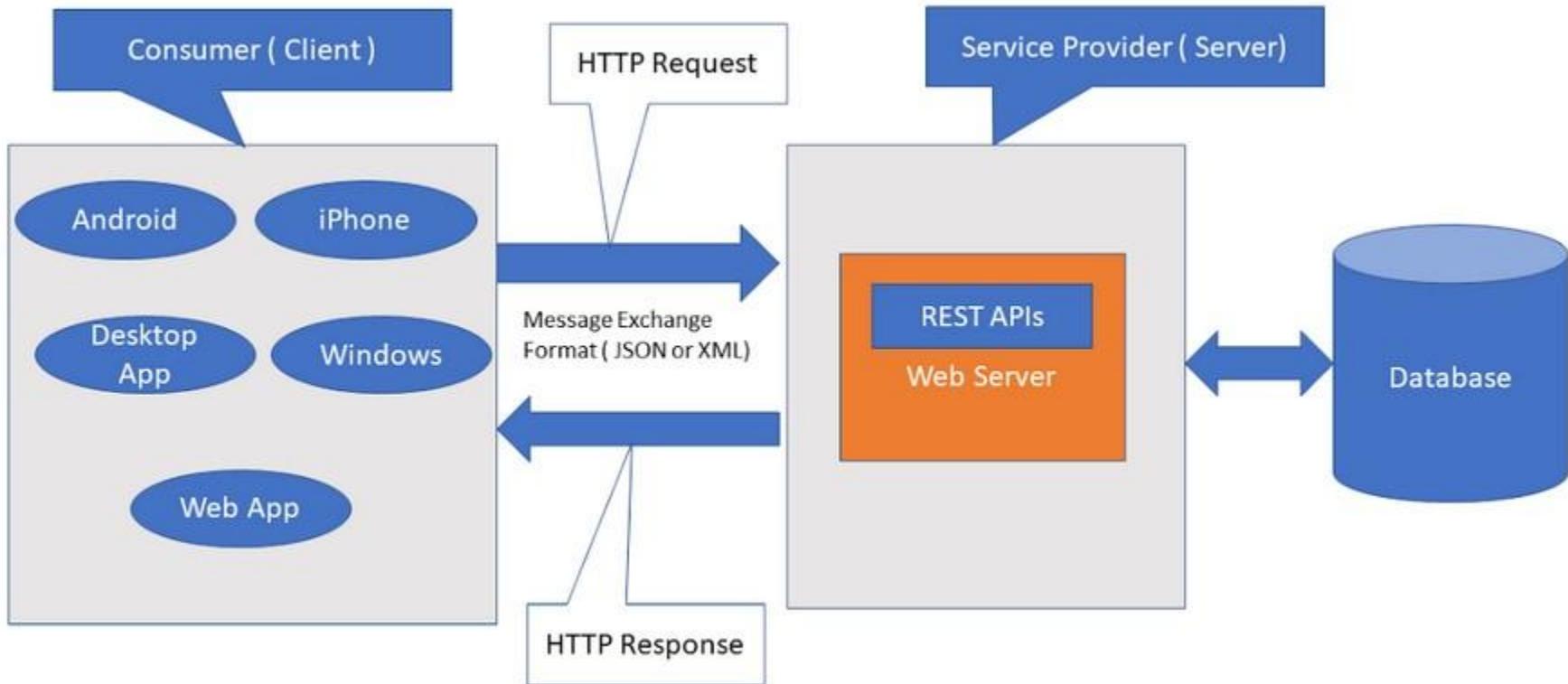
La méthode définit un ensemble de contraintes pour l'accès et la manipulation des données.

REST en quelques mots :

- Roy FIELDING, dans sa thèse soutenue en 2000.
- <http://www.ics.uci.edu/-fielding/pubs/dissertation/top.html>
- C'est un style d'architecture
- C'est une émanation de la façon dont fonctionne le web.
- C'est une alternative à **SOAP**

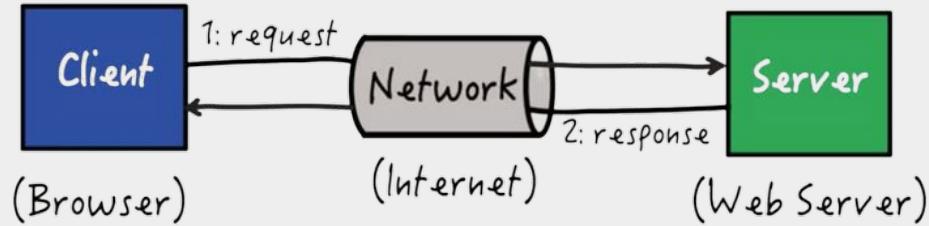
# REST Architecture

---



# Comprendre le protocole HTTP.

Hypertext Transfer Protocol (HTTP)



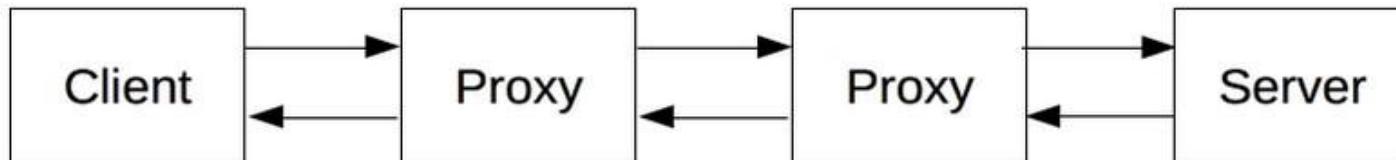
## HTTP (Hypertext Transfer Protocol).

---

La requête émanant de l'agent utilisateur (*généralement le navigateur web, ou le proxy délégué*) est envoyée vers le serveur qui la traite et fournit une réponse.

Les proxys sont les nombreux ordinateurs (ou machines) qui relaient les **messages HTTP** entre le navigateur client et les serveurs.

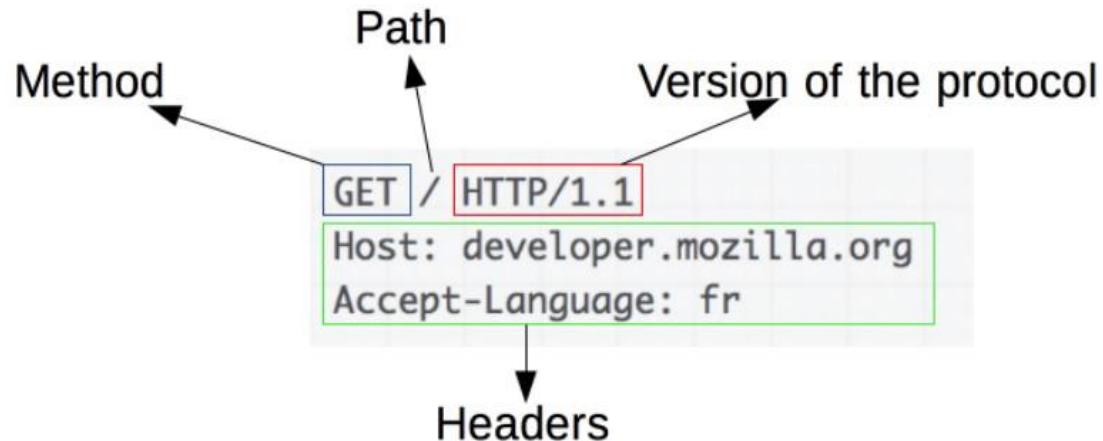
Il existe deux types de **messages HTTP**.



## Les requêtes HTTP

---

- Une méthode HTTP
- Le chemin de la ressource à extraire.
- La version du protocole HTTP
- Les headers

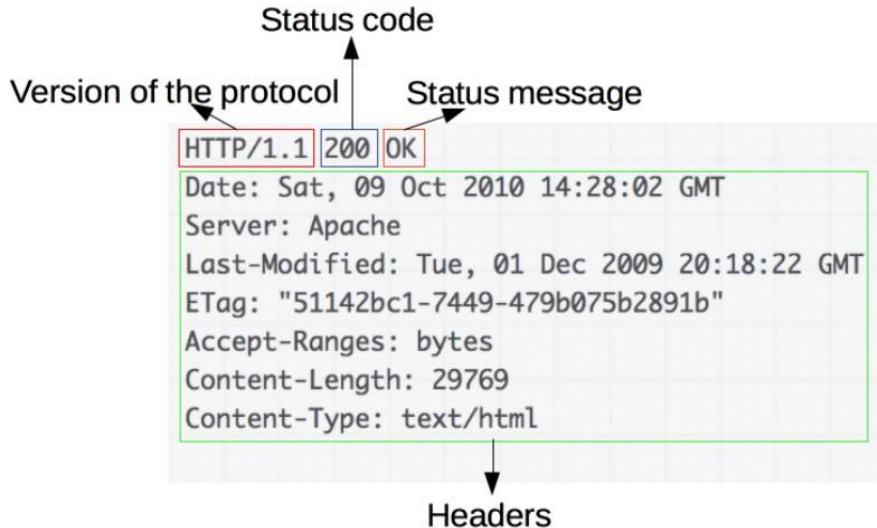


# Les réponses HTTP

---

Une réponse HTTP comprend les éléments suivants :

- La version du protocole HTTP qu'elle suit
- Un code de statut, qui indique si la requête a réussi ou non
- Un message de statut qui est une description rapide, informelle, du code de statut
- Les en-têtes HTTP
- Éventuellement un corps contenant la ressource récupérée.



# Atelier 1

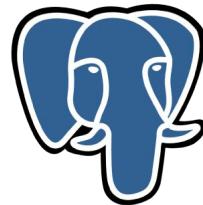
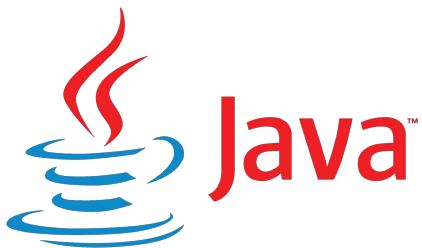
## Création API REST

## Java

# Spring-Boot



# Outils requis



PostgreSQL



# Atelier 1 - Conception d'une API flexible.

---

On souhaite créer une application qui permet de gérer des **Livres**.

Chaque **Livre** est défini par :

- Son titre de type String
- Sa description de type String
- un booléen pour renseigner s'il est publié ou pas

L'application doit permettre de :

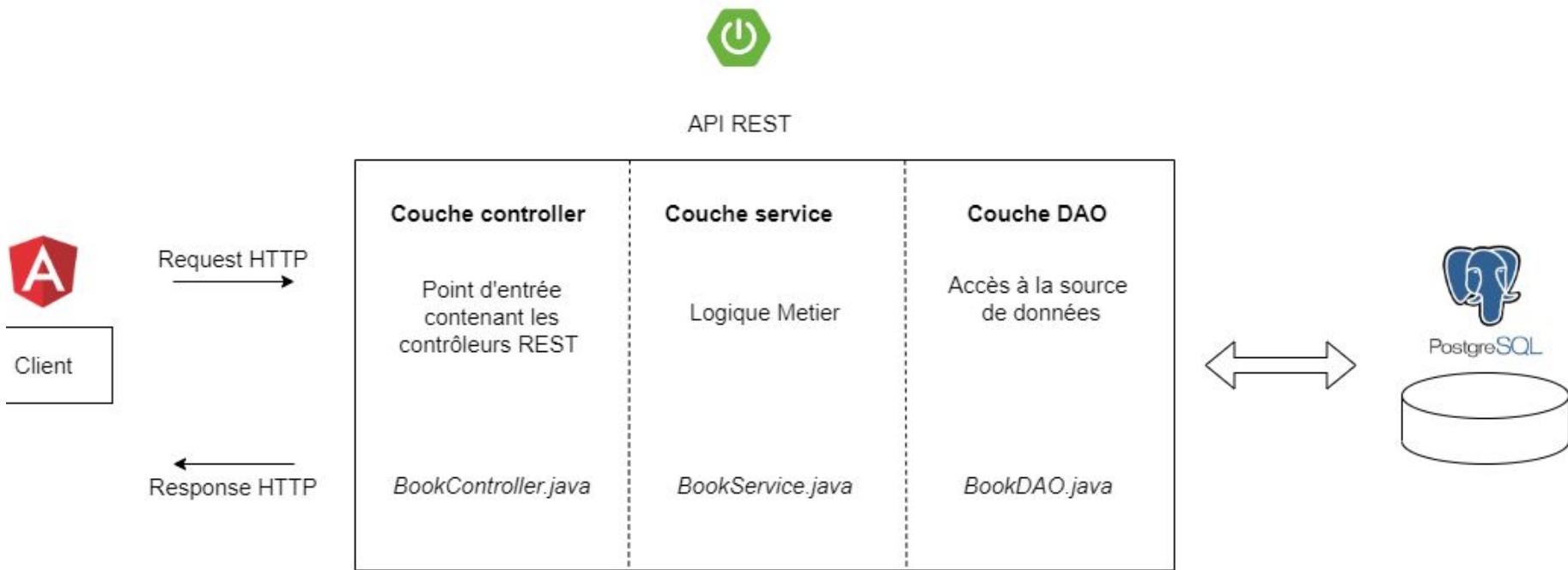
- Ajouter de nouveaux Livre
- Consulter les Livres
- Consulter un Livre
- Mettre à jour un Livre
- Supprimer un Livre
- Supprimer tous les Livre

Les données sont stockées dans une base de données **Postgresql/MySQL**.

L'application est un service Restful basée sur **Spring-Boot**.

# Architecture de l'API REST à construire.

---



# Endpoints de l'API à construire.

---

Méthodes	urls	Description
POST	/api/books	Création d'un nouveau livre
GET	/api/books	Retourne tous les livres
GET	/api/books/:id	Retourne un livre au moyen de son ID
DELETE	/api/books/:id	Supprimer un livre au moyen de son ID
DELETE	/api/books	Supprimer tous les livres

# Initialisation d'un projet Spring.

<https://start.spring.io/>



## Project

Gradle - Groovy

Gradle - Kotlin     Maven

## Language

Java

Kotlin     Groovy

## Dependencies

**ADD DEPENDENCIES...** CTRL + B

No dependency selected

## Spring Boot

3.0.2 (SNAPSHOT)     3.0.1     2.7.8 (SNAPSHOT)     2.7.7

## Project Metadata

Group com.example

Artifact demo

Name demo

Description Demo project for Spring Boot

Package name com.example.demo

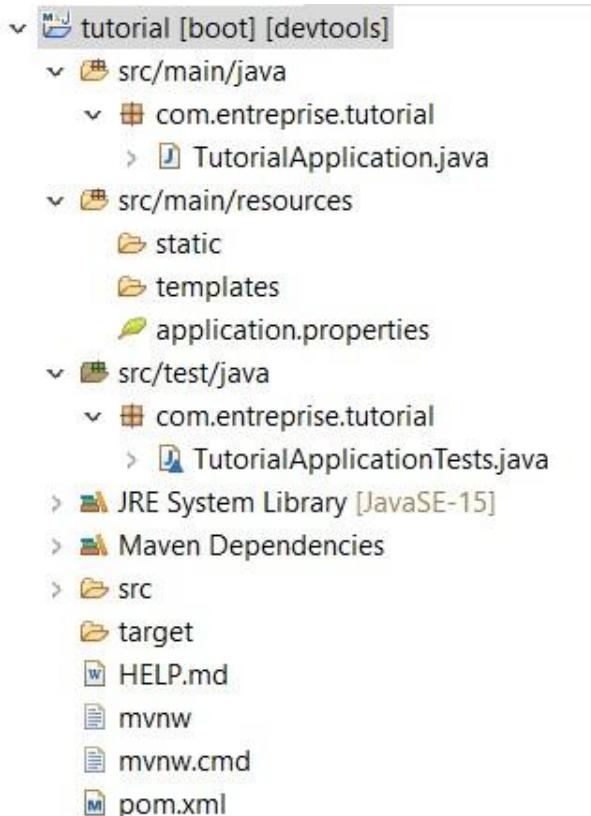
**GENERATE** CTRL + ↵

**EXPLORE** CTRL + SPACE

**SHARE...**

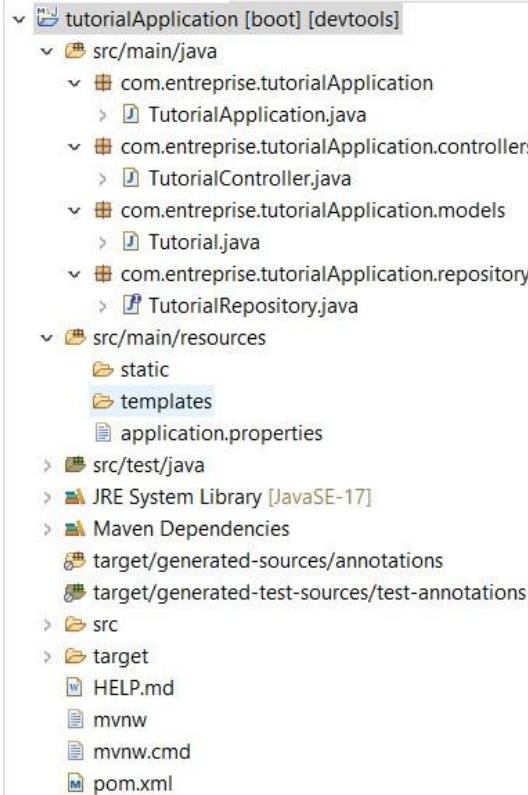
# Structure de base d'un projet.

---



# Structure du projet.

---



# Fichier pom.xml

---

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.1</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.entreprise</groupId>
  <artifactId>tutorial</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>tutorial</name>
  <description>Spring Boot project tuto</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

# Fichier application.properties

---

Configuration de l'accès à la base de données postgresql.

```
#DataSource Settings
spring.datasource.url= jdbc:postgresql://localhost:5432/spring-boot-jpa-postgresqlDB
spring.datasource.username= postgres
spring.datasource.password= xxxxxxxx

spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation= true
spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.PostgreSQLDialect

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto= create
```

## Fichier application.properties

---

Configuration de l'accès à la base de données MySQL.

```
spring.datasource.url=jdbc:mysql://localhost:3306/testdb?useSSL=false
spring.datasource.username=root
spring.datasource.password=123456

spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
```

# Spring-Boot Application

---

Fichier principal d'application.

```
@SpringBootApplication
public class TutorialApplication {

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

}
```

# Fichier entité représentant la classe Tutorial

---

```
package com.entreprise.tutorialApplication.models;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import jakarta.validation.constraints.Size;

@Entity
@Table(name = "tutorial")
public class Tutorial {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Column(name = "title")
    @Size(max = 120)
    private String title;

    @Column(name = "auther")
    @Size(max = 50)
    private String auther;

    @Column(name = "published")
    private boolean published;

    public Tutorial() {

    }

    public Tutorial(String title, String auther) {
        this.title = title;
        this.auther = auther;
        this.published = false;
    }

    public Tutorial(long id, @Size(max = 120) String title, @Size(max = 50) String auther, boolean published) {
        super();
        this.id = id;
        this.title = title;
        this.auther = auther;
        this.published = published;
    }
}
```

# Fichier représentant la l'interface Tutorial Repository

---

```
package com.entreprise.tutorialApplication.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.entreprise.tutorialApplication.models.Tutorial;

@Repository
public interface TutorialRepository extends JpaRepository<Tutorial, Long> {
    List<Tutorial> findByPublished(boolean published);
    List<Tutorial> findByTitle(String title);
}
```

# Fichier représentant la classe Tutorial Controller

---

```
package com.entreprise.tutorialApplication.controllers;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.entreprise.tutorialApplication.models.Tutorial;
import com.entreprise.tutorialApplication.repository.TutorialRepository;

@RestController
@RequestMapping(path = "/api/")
public class TutorialController {

    @Autowired
    TutorialRepository tutorialRepository;

    @GetMapping(path = "/tutorials")
    public List<Tutorial> getTutorials(){
        List<Tutorial> liste_de_tutoriels = (List<Tutorial>) tutorialRepository.findAll();
        if (!liste_de_tutoriels.isEmpty()) {
            return liste_de_tutoriels;
        }

        return null;
    }

    @GetMapping(path = "/tutorials/{id}")
    public Tutorial getTutorialByID(@PathVariable Long id) {
        Optional<Tutorial> retrievedTutorial = tutorialRepository.findById(id);
        if(retrievedTutorial.isPresent()) {
            return retrievedTutorial.get();
        }
        return null;
    }

    @PostMapping(path = "/tutorials")
    public Tutorial addTutorial(@RequestBody Tutorial tutorial) {
        Tutorial _tutorial = tutorialRepository.save(new Tutorial(tutorial.getTitle(), tutorial.getAuthor()));
        return _tutorial;
    }

    @DeleteMapping(path = "/tutorials/{id}")
    public HttpStatus deleteTutorialByID(@PathVariable Long id) {
        tutorialRepository.deleteById(id);
        return HttpStatus.OK;
    }

    @DeleteMapping(path = "/tutorials")
    public HttpStatus deleteTutorials() {
        tutorialRepository.deleteAll();
        return HttpStatus.OK;
    }
}
```

# Présentation de Postman

You are using the Lightweight API Client, sign in or create an account to work with collections, environments and **unlock all free features in Postman.**

History      New Import      GET localhost:8080/api/      +      **localhost:8080/api/**      Save      </>

GET      localhost:8080/api/      Send

Params      Authorization      Headers (6)      Body      Pre-request Script      Tests      Settings      Cookies

Query Params

	Key	Value	Bulk Edit
	Key	Value	

Response

Could not send request

Error: connect ECONNREFUSED 127.0.0.1:8080 | [View in Console](#)

[Learn more about troubleshooting API requests](#)

Console      Not connected to a Postman account



# Chapitre 2 : Les bonnes pratiques.

Conventions et bonnes pratiques – Techniques de Versioning – Bonnes approches de conception et de développement.

# Conventions et bonnes pratiques.

---

## 1. Conception RESTful (pour les API REST) :

- Utiliser les Méthodes HTTP Appropriées : **GET** pour récupérer des données, **POST** pour créer, **PUT/PATCH** pour mettre à jour, et **DELETE** pour supprimer.
- Ressources Nommées de Manière Claire : Utiliser des noms de ressources (au pluriel) clairs et descriptifs dans les **URLs**.
- Gestion des Réponses : Retourner des codes de réponse HTTP appropriés (par exemple, **200 OK**, **404 Not Found**).

# Conventions et bonnes pratiques.

---

## 2. Standardisation :

- **Formats de Réponse Consistents** : Utiliser un format standard (*comme JSON*) pour toutes les réponses.

## 3. Sécurité

- **Authentification et Autorisation** : Implémenter des mécanismes robustes comme OAuth, JWT.
- **Validation et Sanitisation des Données** : Toujours valider les données entrantes pour prévenir les attaques telles que l'injection SQL.

## Conventions et bonnes pratiques.

---

### 4. Documentation:

- **Documenter l'API** : Fournir une documentation complète et à jour, idéalement avec des exemples d'utilisation.
- **Utiliser des Outils de Documentation** : Comme Swagger ou Redoc pour générer une documentation interactive.

### 5. Gestion des Erreurs

- **Messages d'Erreur Clairs** : Fournir des messages d'erreur descriptifs avec des codes d'erreur standardisés.
- **Documentation des Erreurs** : Documenter les types d'erreurs possibles et leur signification.

# Conventions et bonnes pratiques.

---

## 6. Performances et Scalabilité

- **Pagination** : Pour les grandes collections de données, utiliser la pagination pour limiter la charge du serveur et améliorer la réactivité.
- **Mise en Cache** : Utiliser des mécanismes de mise en cache pour améliorer les performances.

## 7. Versioning

- **Versionner l'API** : Prévoir des versions pour l'API pour gérer les changements sans perturber les utilisateurs existants.
- **Stratégie de Versioning** : Utiliser des URL ou des en-têtes pour gérer différentes versions.

## Conventions et bonnes pratiques.

---

### 8. Respect des Limites de Taux

- **Limitation des Taux** : Mettre en place des limites pour prévenir l'abus et la surcharge des serveurs.

### 9. Test et Monitoring

- **Tests Automatisés** : Écrire des tests pour chaque aspect de l'API.
- **Surveillance de l'API** : Utiliser des outils de monitoring pour surveiller les performances et la santé de l'API.

### 10. Prise en Charge des Clients : Etre réceptif aux retours utilisateurs - produire des guides.

# API REST avec Swagger

---

- Pour Spring Boot 3 :

Pour utiliser Swagger 3 dans votre projet Maven, vous devez ajouter la dépendance ***springdoc-openapi-starter-webmvc-ui*** au fichier *pom.xml* de votre projet :

```
<!-- Ajout swagger-->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.0.3</version>
</dependency>
```

<http://localhost:8080/swagger-ui/index.html>

# Refactoring de l'API.

---

```
package com.entreprise.tutorialApplication.controllers;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import com.entreprise.tutorialApplication.models.Tutorial;
import com.entreprise.tutorialApplication.repository.TutorialRepository;

@RestController
@RequestMapping(path = "/api")
public class TutorialController {

    @Autowired
    TutorialRepository tutorialRepository;

    @GetMapping(path = "/tutorials", produces = "application/json")
    public ResponseEntity<List<Tutorial>> getTutorials(@RequestParam(required = false) String title){
        try {
            List<Tutorial> list_of_tutorials = new ArrayList<Tutorial>();
            if (title == null) {
                tutorialRepository.findAll().forEach(list_of_tutorials::add);
            } else {
                tutorialRepository.findByTitle(title).forEach(list_of_tutorials::add);
            }
            if (list_of_tutorials.isEmpty()) {
                return new ResponseEntity<List<Tutorial>>(HttpStatus.NO_CONTENT);
            }
            return new ResponseEntity<>( list_of_tutorials, HttpStatus.OK);
        } catch (Exception e) {
            // TODO: handle exception
            return new ResponseEntity<List<Tutorial>>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}
```

# Refactoring de l'API.

---

```
@GetMapping(path = "/tutorials/{id}", produces = "application/json")
public ResponseEntity<Tutorial> getTutorialByID(@PathVariable Long id){
    try {
        Optional<Tutorial> retrievedTutorial = tutorialRepository.findById(id);
        if(retrievedTutorial.isPresent()) {
            return new ResponseEntity<Tutorial>(retrievedTutorial.get(),HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    } catch (Exception e) {
        // TODO: handle exception
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

@PostMapping(path = "/tutorials", produces = "application/json")
public ResponseEntity<Tutorial> addTutorial(@RequestBody Tutorial tutorial){
    try {
        Tutorial _tutorial = tutorialRepository.save(new Tutorial(tutorial.getTitle(), tutorial.getAuthor()));
        return new ResponseEntity<Tutorial>(_tutorial,HttpStatus.CREATED);
    } catch (Exception e) {
        // TODO: handle exception
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

# Refactoring de l'API.

---

```
    @DeleteMapping(path = "/tutorials/{id}", produces = "application/json")
    public ResponseEntity<HttpStatus> deleteByID(@PathVariable Long id){
        try {
            tutorialRepository.deleteById(id);
            return new ResponseEntity<HttpStatus>(HttpStatus.OK);
        } catch (Exception e) {
            // TODO: handle exception
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    @DeleteMapping(path = "/tutorials", produces = "application/json")
    public ResponseEntity<HttpStatus> deleteAll(){
        try {
            tutorialRepository.deleteAll();
            return new ResponseEntity<HttpStatus>(HttpStatus.OK);
        } catch (Exception e) {
            // TODO: handle exception
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}
```



# Travaux-pratiques

Développement d'une API REST.

On souhaite créer une application qui permet de gérer le catalogue des livres appartenant à des catégories différentes.

Chaque Livre est défini par :

- Son titre de type String
- Son auteur de type String
- Son prix de type float
- Sa disponibilité de type boolean

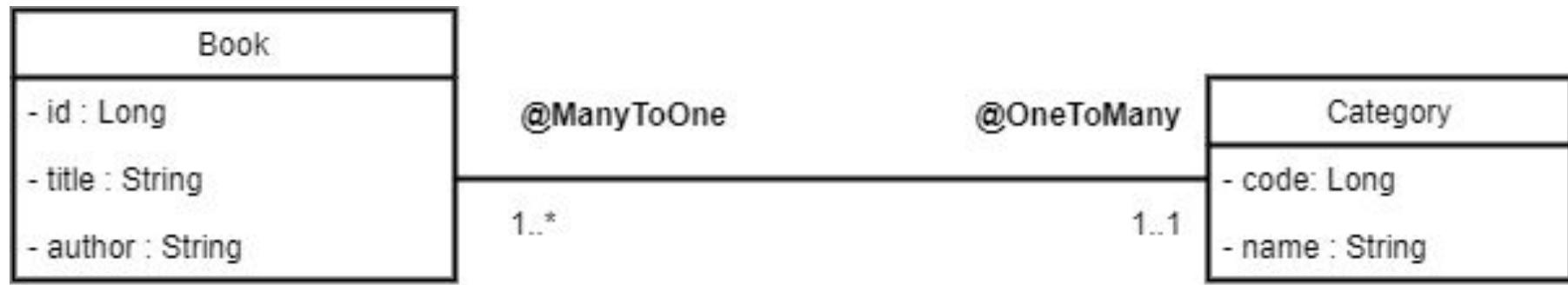
Une catégorie est définie par :

- Son code de type Long (Auto Increment)
- Son nom de type String

L'application doit permettre :

- D'ajouter une nouvelle catégorie
- Ajouter un livre appartenant à une catégorie
- Consulter toutes les catégories
- Consulter les livres dont le nom contient un mot clé
- Consulter les livres d'une catégorie
- Consulter un livre
- Mettre à jour un livre
- Supprimer une catégorie

## Diagramme de classe : Modèle objet



- CATEGORY (ID\_CAT, NOM\_CATEGORY)  
- BOOK (ID, TITLE, AUTHOR, PRIX, DISPO, #ID\_CAT)

## Entités JPA

```
@Entity
@Table(name = "book")
public class Book implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "title")
    private String title;
    @Column(name = "author")
    private String author;
    @Column(name = "price")
    private float price;
    @ManyToOne
    @JoinColumn(name="ID_CATEGORY")
    private Category cat;
```

```
@Entity
@Table(name = "category")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long code;
    @Column(name = "name")
    private String name;
    @OneToMany(mappedBy = "cat", fetch = FetchType.LAZY)
    private Collection<Book> books;
```

## Base de données

Tables (2)	
▼	book
▼	category
Columns (5)	
id	
author	
price	
title	
Columns (2)	
code	
name	

# La boîte à outils.



JSON



HTTP



Insomnia

## Qu'est-ce que OpenAPI ?

---

La Spécification **OpenAPI** (anciennement Spécification Swagger) est un format de description d'API pour les API REST. Un fichier OpenAPI vous permet de décrire l'intégralité de votre API, notamment :

- Points de terminaison disponibles (**/users**) et opérations sur chaque point de terminaison (**GET /users, POST /users**)
- Paramètres de fonctionnement Entrée et sortie pour chaque opération
- Méthodes d'authentification
- Coordonnées, licence, conditions d'utilisation et autres informations.

Les spécifications de l'API peuvent être écrites en **YAML** ou **JSON**. Le format est facile à apprendre et lisible aussi bien par les humains que par les machines. <sup>74</sup>

## Qu'est-ce que Swagger ?

---

Swagger est un ensemble d'outils open source construits autour de la spécification **OpenAPI** qui peuvent vous aider à concevoir, créer, documenter et utiliser des API REST.

<https://editor.swagger.io/>

## Conception d'APIs ReST avec OpenAPI et Swagger.

---

La spécification définit plusieurs étiquettes qui vont permettre entre autres de :

- définir les informations générales sur vos **API** : description, termes d'utilisation, licence, contact, etc. ;
- fournir la liste des **services** qui seront offerts, avec pour chacun, comment les appeler et la structure de la réponse qui est retournée ;
- définir le chemin pour consommer votre **API** ;
- etc.

# Structure d'un document YAML OpenAPI.

---

```
1. openapi: 3.0.0
2. info:
3.   title: Sample API
4.   description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/) or HTML.
5.   version: 0.1.9
6.
7. servers:
8.   - url: http://api.example.com/v1
9.     description: Optional server description, e.g. Main (production) server
10.    - url: http://staging-api.example.com
11.      description: Optional server description, e.g. Internal staging server for testing
12.
13. paths:
14.   /users:
15.     get:
16.       summary: Returns a list of users.
17.       description: Optional extended description in CommonMark or HTML.
18.       responses:
19.         '200': # status code
20.           description: A JSON array of user names
21.           content:
22.             application/json:
23.               schema:
24.                 type: array
25.                 items:
26.                   type: string
```



Environnement de test &  
outils.

# JSON Generator

JSON GENERATOR

## Generate

Reset

 Help Ukraine!

[Try the next version](#)

Help

 Feedback

# Rappels sur la sécurité



# **Les grands principes de la sécurité informatique.**

---

Les grands principes de la sécurité informatique sont essentiels pour protéger les systèmes d'information contre diverses menaces.

- Confidentialité
- Intégrité
- Disponibilité
- Authentification
- Autorisation
- Non-répudiation
- Audit et surveillance

## Les différentes injections.

---

**XSS (Cross-Site Scripting)** : Le principe est d'injecter des données arbitraires dans une page web. Si les données arrivent telles quelles dans la page transmise au navigateur sans avoir été vérifiées alors il existe une faille. On peut s'en servir pour exécuter un code malveillant en langage de script dans le navigateur web qui consulte cette page.

**Retour d'expérience - Cas de la SNCF** : L'attaquant est injecté dans le header X-APP du Javascript sur l'appel à /api/account (un des endpoint de l'application **SNCF Connect**). Il récupère ensuite un mail de création de compte comportant un lien. Quand on clique sur ce lien, le javascript malveillant injecté par l'attaquant est exécuté dans le navigateur de la victime. Ce javascript pourrait permettre de récupérer le **mot de passe** de la personne.

# Les différentes injections.

---

## Les injections SQL :

Une **injection SQL** est une technique d'attaque informatique utilisée pour manipuler ou détruire une base de données à travers une application web.

Elle se produit lorsque un attaquant parvient à insérer ou "injecter" une requête **SQL** malveillante dans une entrée de l'application qui est ensuite exécutée par le système de gestion de base de données (SGBD).

Ce type d'attaque peut permettre à un attaquant de lire des données sensibles, de les modifier, de les supprimer, ou d'exécuter des opérations administratives sur la base de données, comme éléver ses privilèges.

# Pour se protéger contre les injections SQL.

---

## Valider et Sanitiser les Entrées Utilisateur :

La **validation** consiste à vérifier si les données entrées par l'utilisateur respectent un certain format ou des critères prédéfinis (par exemple, un numéro de téléphone ou une adresse e-mail valide).

La **sanitation**, quant à elle, consiste à nettoyer les données entrantes en retirant ou en remplaçant les caractères qui pourraient être utilisés pour une injection **SQL**.

## Utiliser des ORM (Object-Relational Mapping)

Des **frameworks ORM** comme **Hibernate** encapsulent les requêtes **SQL** et utilisent souvent les **Prepared Statements**, ce qui réduit le risque d'injections SQL. Ils offrent également une abstraction supplémentaire qui aide à sécuriser les requêtes.

## Pour se protéger contre les injections SQL.

---

### Appliquer le Principe du Moindre Privilège

Limitez les permissions de l'utilisateur de la base de données utilisé par l'application au strict nécessaire. Par exemple, si votre application n'a pas besoin de modifier les données, vous pouvez limiter l'utilisateur à des opérations de lecture seule.

### Utiliser des Outils et des Bibliothèques de Sécurité

Des outils comme **OWASP ESAPI** (Enterprise Security API) fournissent des méthodes pour sécuriser les entrées et les sorties des applications, offrant une couche supplémentaire de protection contre les injections **SQL** et d'autres vulnérabilités.

## Présentation de l'OWASP TOP 10

---

**L'OWASP Top 10** est un document de référence conçu pour sensibiliser les développeurs et les professionnels de la sécurité des applications Web aux risques de sécurité les plus critiques. Il est reconnu mondialement et est considéré comme une première étape importante vers un codage plus sécurisé.

La liste de l'OWASP Top 10 est mise à jour régulièrement pour refléter les nouvelles tendances et menaces dans le domaine de la sécurité des applications Web.

Elle comprend des catégories telles que le contrôle d'accès défectueux, les échecs cryptographiques, et les injections, entre autres.

<https://owasp.org/Top10/fr/>

## Pentesting

---

Le **pentesting**, ou **test d'intrusion**, est une pratique de cybersécurité où des experts en sécurité simulent des attaques informatiques sur des systèmes, réseaux ou applications afin d'identifier et de corriger les vulnérabilités de sécurité.

L'objectif est de découvrir les faiblesses avant qu'elles ne soient exploitées par des acteurs malveillants.

Le **pentesting** peut couvrir divers aspects, y compris les tests des systèmes physiques, des réseaux sans fil, des applications web, et plus encore. Il joue un rôle essentiel dans la maintenance de la sécurité informatique dans les organisations.

## Introduction à RESTler-Fuzzer.

---

**Restler-Fuzzer** est un outil développé par **Microsoft** pour le test d'intrusion automatique de services web **RESTful**. Il est conçu pour détecter rapidement et automatiquement des failles de sécurité dans des **API REST**. En utilisant des techniques de fuzzing, **Restler-Fuzzer** génère et envoie des requêtes potentiellement malveillantes à une API pour identifier des vulnérabilités telles que des injections SQL, des débordements de tampon ou des problèmes de gestion d'authentification.

C'est un outil utile pour les développeurs et les professionnels de la sécurité afin de renforcer la sécurité des **API REST** avant leur déploiement.

<https://github.com/microsoft/restler-fuzzer>

# **Authentification & autorisation**

## **JWT (Json Web Token)**



## CORS (Cross-Origin Resource Sharing)

---

- Il s'agit d'un mécanisme basé sur un en-tête **HTTP** qui permet à un serveur d'indiquer toute origine (*domaine, schéma ou port*) autre que la sienne à partir de laquelle un navigateur doit autoriser le chargement de ressources.
- CORS est un mécanisme qui permet d'accéder aux ressources restreintes d'une page Web à partir d'un autre domaine en dehors du domaine à partir duquel la première ressource a été servie.

## CORS (Cross-Origin Resource Sharing)

---

Client

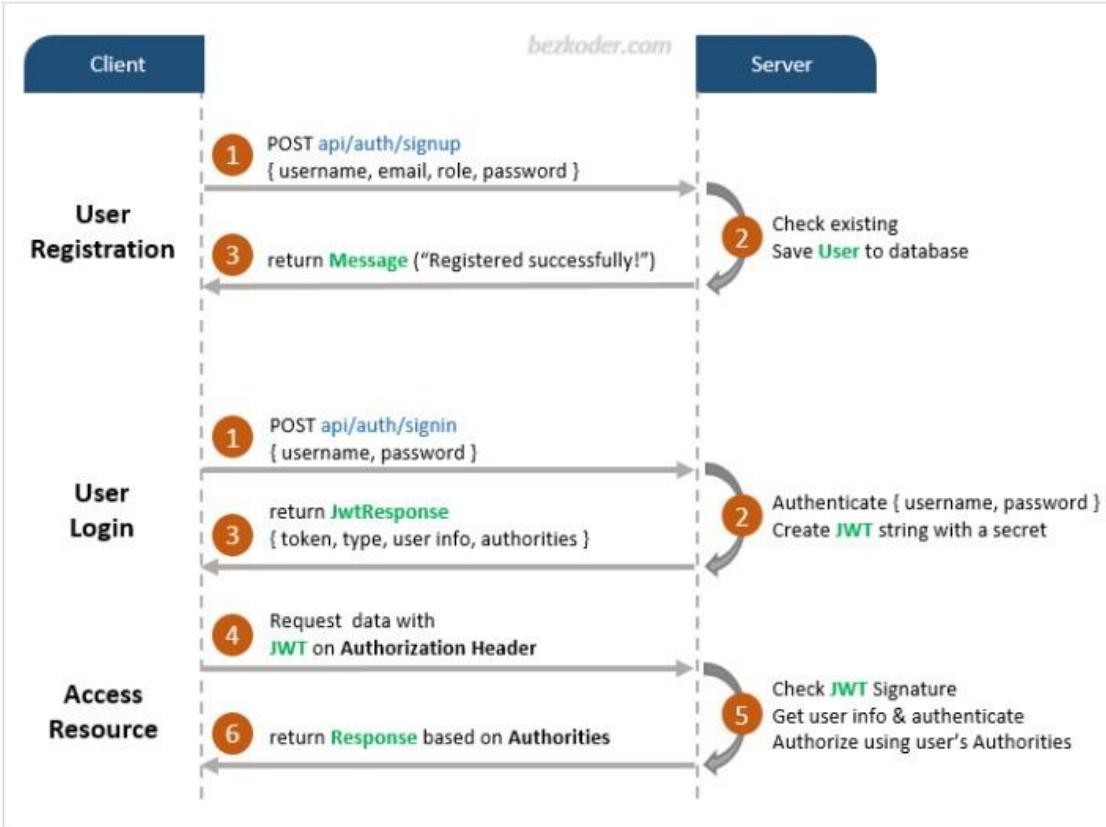
Server

```
GET /resources/public-data/ HTTP/1.1
Origin: https://foo.example
```

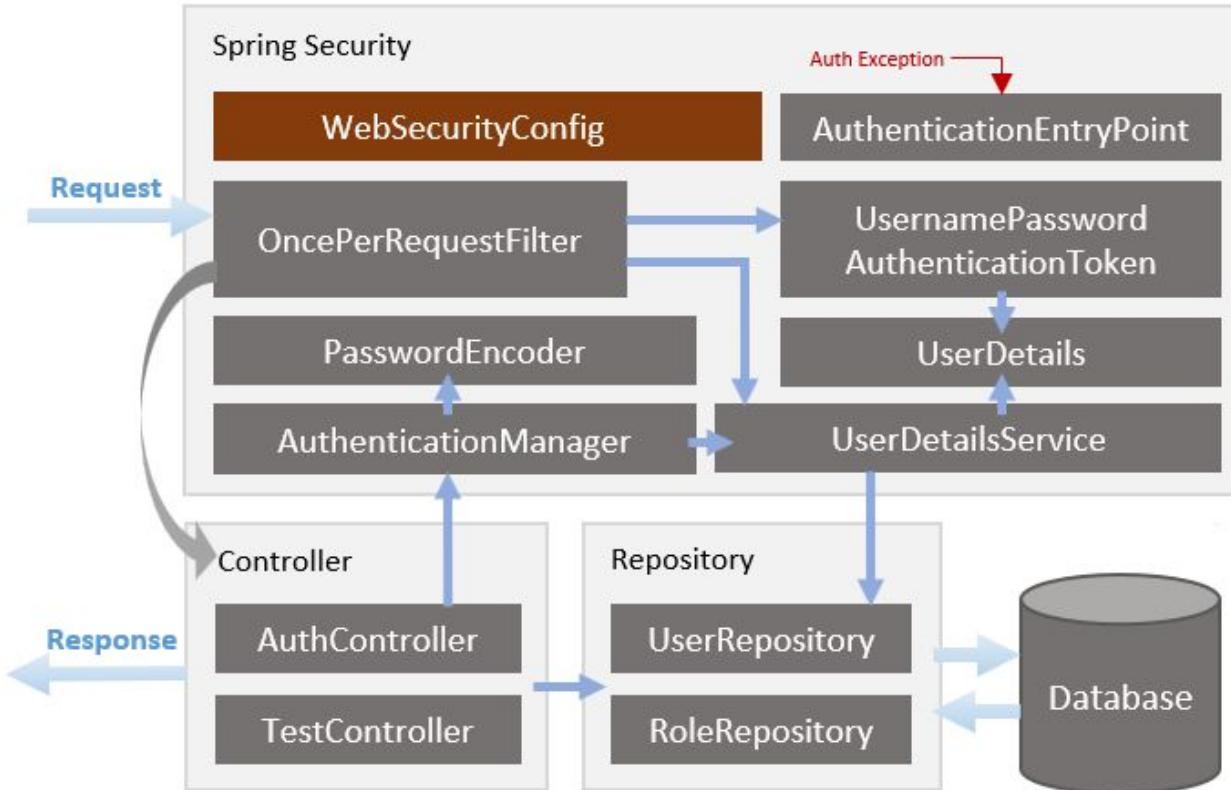
```
HTTP/1.1 200 OK
```

```
Access-Control-Allow-Origin: *
```

# Sign up and login avec JWT authentification



# Authentification basée sur Spring-Boot, Spring security & JWT.



# Dépendances nécessaires.

---

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>

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

# Package models.

Classe énumérée **ERole** pour l'énumération des rôles ou priviléges dans l'application (*ROLE\_USER* et *ROLE\_ADMINISTRATOR*).

```
package com.entreprise.tutorialApplication.models;

public enum ERole {
    ROLE_USER,
    ROLE_ADMINISTRATOR
}
```

Classe **Role**, pour la gestion des rôles. Les noms reconnus aux rôles sont de type **ERoles**.

```
package com.entreprise.tutorialApplication.models;
```

```
import jakarta.persistence.Column;
```

```
@Entity
@Table(name = "roles")
public class Role {
```

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

```
    @Column(length = 20)
    private ERole name;
```

```
    public Role() {}
```

```
    public Role(ERole name) {
        this.name = name;
    }
```

```
    public Integer getId() {
        return id;
    }
```

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

```
    public ERole getName() {
        return name;
    }
```

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

# Package models - Classe User.

```
package com.entreprise.tutorialApplication.models;

import java.util.HashSet;

@Entity
@Table(name = "Users", uniqueConstraints = {
    @UniqueConstraint(columnNames = "username"),
    @UniqueConstraint(columnNames = "email")
})
public class User {

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

    @Column(name = "username")
    @Size(max = 50)
    private String username;

    @Column(name = "email")
    @Size(max = 120)
    @Email
    private String email;

    @Column(name = "password")
    @Size(max = 20)
    private String password;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id") )
    private Set<Role> roles = new HashSet<Role>();
}
```

Classe **User** de gestion des utilisateurs. Chaque utilisateur disposant des attributs **username** - **email** - **password** et **liste des rôles**.

## Package models - Classe User (suite).

---

```
//Getters & Setters

public User() {}

public User(String username, String email, String password) {
    this.username = username;
    this.email = email;
    this.password = password;
}

public Long getId() {
    return id;
}

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

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getPassword() {
    return password;
}
```

# Package Repository : UserRepository et RoleRepository.

```
package com.entreprise.tutorialApplication.repository;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.entreprise.tutorialApplication.models.Tutorial;

@Repository
public interface TutorialRepository extends JpaRepository<Tutorial, Long> {
    List<Tutorial> findByPublished(boolean published);
    List<Tutorial> findByTitle(String title);
}
```

**UserRepository** Responsable de l'accès aux données utilisateur en BD.

```
import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.entreprise.tutorialApplication.models.User;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
    Boolean existsByUsername(String username);
    Boolean existsByEmail(String email);
```

**RoleRepository** Responsable de l'accès aux données sur les rôles en BD.

# Package security.service - UserDetailsImpl.

---

```
package com.entreprise.tutorialApplication.security.service;

import java.util.Collection;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import com.entreprise.tutorialApplication.models.User;
import com.fasterxml.jackson.annotation.JsonIgnore;

public class UserDetailsImpl implements UserDetails {

    private static final long serialVersionUID = 1L;

    private Long id;

    private String username;

    private String email;

    @JsonIgnore
    private String password;

    private Collection<? extends GrantedAuthority> authorities;

    public UserDetailsImpl(Long id, String username, String email,
        String password, Collection<? extends GrantedAuthority> authorities ) {
        this.id = id;
        this.username = username;
        this.email = email;
        this.password = password;
        this.authorities = authorities;
    }
}
```

**UserDetails** contient les informations nécessaires (*telles que : nom d'utilisateur, mot de passe, autorités*) pour créer un objet d'authentification.

# Package security.service - UserDetailsImpl (Suite).

---

```
public static UserDetailsImpl build(User user) {
    List<GrantedAuthority> authorities = user.getRoles().stream().map(role -> new SimpleGrantedAuthority(role.getName().name()))
        .collect(Collectors.toList());

    return new UserDetailsImpl(
        user.getId(),
        user.getUsername(),
        user.getEmail(),
        user.getPassword(),
        authorities);
}

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    // TODO Auto-generated method stub
    return null;
}

public Long getID() {
    return id;
}

public void setID(Long id) {
    this.id = id;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}
```

## Package security.service - UserDetailsImpl (Suite).

---

```
public boolean equals(Object o) {  
    if (this == o) {  
        return true;  
    }  
    if (o == null || getClass() != o.getClass() ) {  
        return false;  
    }  
    UserDetailsImpl user = (UserDetailsImpl) o;  
    return Objects.equals(id, user.id);  
}
```

# Package security.service - UserDetailsServiceImpl.

---

```
package com.entreprise.tutorialApplication.security.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.entreprise.tutorialApplication.models.User;
import com.entreprise.tutorialApplication.repository.UserRepository;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        // TODO Auto-generated method stub
        User user = userRepository.findByUsername(username)
            .orElseThrow( () -> new UsernameNotFoundException("User not found with username" + username));

        return UserDetailsImpl.build(user);
    }
}
```

L'interface **UserDetailsService** dispose d'une méthode pour charger l'utilisateur par nom d'utilisateur et renvoie un objet **UserDetails** que **Spring Security** peut utiliser pour l'authentification et la validation.

# Package security.jwt - AuthenticationEntryPoint

---

```
import java.io.IOException;

@Component
public class AuthEntryPointJwt implements AuthenticationEntryPoint {

    private static final Logger logger = LoggerFactory.getLogger(AuthEntryPointJwt.class);

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)
        throws IOException, ServletException {
        logger.error("Unauthorized error: {}", authException.getMessage());

        response.setContentType(MediaType.APPLICATION_JSON_VALUE);
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

        final Map<String, Object> body = new HashMap<>();
        body.put("status", HttpServletResponse.SC_UNAUTHORIZED);
        body.put("error", "Unauthorized");
        body.put("message", authException.getMessage());
        body.put("path", request.getServletPath());

        final ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(response.getOutputStream(), body);
    }
}
```

## AuthenticationEntryPoint

déetectera  
d'authentification.

l'erreur

# Package security.jwt - AuthTokenFilter.

```
package com.entreprise.tutorialApplication.security.jwt;

import org.slf4j.Logger;

public class AuthTokenFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtils jwtUtils;

    private static final Logger logger = LoggerFactory.getLogger(AuthTokenFilter.class);

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        // TODO Auto-generated method stub
        try {
            String jwt = parseJwt(request);

        } catch (Exception e) {
            // TODO: handle exception
        }
    }

    private String parseJwt(HttpServletRequest request) {
        // TODO Auto-generated method stub
        String headerAuth = request.getHeader("Authorization");
        if(StringUtils.hasText(headerAuth) && headerAuth.startsWith("Beearer")) {
            return headerAuth.substring(7);
        }

        return null;
    }
}
```

**OncePerRequestFilter** effectue une seule exécution pour chaque requête adressée à notre **API**.

Il fournit une méthode **doFilterInternal()** que nous implémenterons en analysant et en validant **JWT**, en chargeant les détails de l'utilisateur (à l'aide de **UserDetailsService**), en vérifiant l'autorisation (à l'aide de **UsernamePasswordAuthenticationToken**).

# Package security.jwt - JwtUtils.

---

```
package com.entreprise.tutorialApplication.security.jwt;

import java.util.Date;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.Authentication;
import org.springframework.stereotype.Component;

import com.entreprise.tutorialApplication.security.service.UserDetailsImpl;

import io.jsonwebtoken.ExpiredJwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.MalformedJwtException;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.UnsupportedJwtException;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;

import java.security.Key;

@Component
public class JwtUtils {

    private static final Logger logger = LoggerFactory.getLogger(JwtUtils.class);

    @Value("${orsys.app.jwtSecret}")
    private String jwtSecret;

    @Value("${orsys.app.jwtExpirationMs}")
    private int jwtExpirationMs;

    public String generateJwtToken(Authentication authentication) {
        UserDetailsImpl userPrincipal = (UserDetailsImpl) authentication;

        return Jwts.builder()
            .setSubject((userPrincipal.getUsername()))
            .setIssuedAt(new Date())
            .setExpiration(new Date((new Date()).getTime() + jwtExpirationMs))
            .signWith(Key(), SignatureAlgorithm.HS256)
            .compact();
    }

    private Key Key() {
        // TODO Auto-generated method stub
        return Keys.hmacShaKeyFor(Decoders.BASE64.decode(jwtSecret));
    }
}
```

## Package security.jwt - JwtUtils.

---

```
public String getUserNameFromJwtToken(String token) {
    return Jwts.parserBuilder().setSigningKey(Key()).build()
        .parseClaimsJws(token).getBody().getSubject();
}

public boolean validateJwtToken(String authToken) {
    try {
        Jwts.parserBuilder().setSigningKey(Key()).build().parse(authToken);
        return true;
    } catch (MalformedJwtException e) {
        Logger.error("Invalid JWT token: {}", e.getMessage());
    } catch (ExpiredJwtException e) {
        Logger.error("JWT token is expired: {}", e.getMessage());
    } catch (UnsupportedJwtException e) {
        Logger.error("JWT token is unsupported: {}", e.getMessage());
    } catch (IllegalArgumentException e) {
        Logger.error("JWT claims string is empty: {}", e.getMessage());
    }

    return false;
}
```

# Package configuration - WebSecurityConfig.

---

```
package com.entreprise.tutorialApplication.configuration;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import com.entreprise.tutorialApplication.security.jwt.AuthEntryPointJwt;
import com.entreprise.tutorialApplication.security.jwt.AuthTokenFilter;
import com.entreprise.tutorialApplication.security.service.UserDetailsServiceImpl;
```

# Package configuration - WebSecurityConfig.

---

```
@Configuration
@EnableMethodSecurity
public class WebSecurityConfig { // extends WebSecurityConfigurerAdapter {
    @Autowired
    UserDetailsServiceImpl userDetailsService;

    @Autowired
    private AuthEntryPointJwt unauthorizedHandler;

    @Bean
    public AuthTokenFilter authenticationJwtTokenFilter() {
        return new AuthTokenFilter();
    }

    @Bean
    public DaoAuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();

        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());

        return authProvider;
    }

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

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

# Package configuration - WebSecurityConfig.

---

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf(csrf -> csrf.disable())
        .exceptionHandling(exception -> exception.authenticationEntryPoint(unauthorizedHandler))
        .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth ->
            auth.requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/api/test/**").permitAll()
                .anyRequest().authenticated()
        );
    http.authenticationProvider(authenticationProvider());
    http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
}
```



# Travaux-pratiques

Challenge sur une API non sécurisée.

# Les Tests d'API.



# Les 10 domaines de test d'une application.

---

Les tests d'API (Application Programming Interface) sont une partie essentielle du développement de logiciels modernes.

Ils visent à s'assurer que les interfaces entre différentes applications logicielles fonctionnent correctement.

**Tests de validation** : Vérifient si l'API se comporte comme prévu et répond correctement aux requêtes.

**Tests fonctionnels** : Évaluent les fonctionnalités spécifiques et les opérations que l'API est censée effectuer.

**Tests de charge** : Mesurent la capacité de l'API à gérer un grand nombre de requêtes simultanées sans perte de performance.

# Les 10 domaines de test d'une application.

---

**Tests de performance** : Évaluent la rapidité et l'efficacité avec lesquelles l'API traite les requêtes.

**Tests de sécurité** : Vérifient la robustesse de l'API face à diverses menaces de sécurité, telles que les injections SQL, le Cross-Site Scripting (XSS), et autres vulnérabilités.

**Tests d'intégration** : Assurent que l'API fonctionne correctement lorsqu'elle est intégrée avec d'autres composants ou systèmes.

**Tests de compatibilité** : Vérifient que l'API fonctionne sur différents systèmes, plateformes, et appareils.

**Tests d'interface utilisateur** : Dans le cas où l'API est liée à une interface utilisateur, ces tests vérifient que l'interaction entre l'UI et l'API est fluide et conforme aux attentes.

**Tests de contrat** : Évaluent la conformité de l'API avec sa documentation et ses spécifications, garantissant que les contrats entre les différentes parties de l'API sont respectés.

**Tests de fiabilité** : Vérifient la stabilité et la fiabilité de l'API sur de longues périodes, en s'assurant qu'elle peut gérer des requêtes de manière constante et fiable.

# Avantages et limites des tests d'API one shot.

---

Les tests "one shot" sont une méthode de test où un test particulier est exécuté une seule fois, plutôt que dans un cadre de tests répétitifs ou automatisés.

## Avantages :

**Efficacité pour les Scénarios Rares** : Utile pour tester des cas uniques qui ne nécessitent pas un cadre de test répété.

**Économie de Ressources** : Peut économiser du temps et des ressources si le scénario de test ne se présentera probablement pas à nouveau.

**Rapidité** : Permet une exécution rapide sans nécessiter une planification à long terme ou des ressources de maintenance continue.

**Flexibilité** : Adapté aux situations où les conditions de test ne peuvent pas être facilement dupliquées ou automatisées.

# Avantages et limites des tests d'API one shot.

---

## Inconvénients

**Manque de Répétabilité** : Ne permet pas de vérifier la fiabilité ou la stabilité sur plusieurs exécutions.

**Risque d'Erreur Humaine** : Plus susceptible aux erreurs, car il manque souvent de la rigueur et de la standardisation des tests automatisés.

**Limité en Couverture de Test** : Ne permet pas une évaluation exhaustive de toutes les fonctionnalités ou scénarios possibles.

**Problèmes de Documentation et de Suivi** : Peut manquer de documentation détaillée, rendant difficile la reproduction ou l'examen des tests pour un dépannage futur.

**Non Idéal pour les Environnements en Évolution** : Moins adapté pour les applications ou systèmes qui subissent des changements fréquents.

## **Les tests de durcissement.**

---

Les tests de durcissement, souvent appelés "**hardening tests**" en anglais, sont un type de tests informatiques axés sur le renforcement de la sécurité des systèmes, des applications ou des réseaux.

Leur objectif principal est de minimiser les vulnérabilités et de renforcer la résistance aux attaques ou aux intrusions malveillantes.

**Objectif** : Renforcer la sécurité en identifiant et en corrigeant les vulnérabilités.

**Cible** : Peut être appliqué à des logiciels, des systèmes d'exploitation, des réseaux ou des infrastructures cloud.

**Processus** : Implique généralement l'analyse, le test, et la mise en œuvre de mesures de sécurité supplémentaires.



# Travaux-pratiques

Tests d'une API avec Postman

# API Management



## Qu'est-ce qu'une solution d'API Management ?

---

Une solution d'**API Management**, est un ensemble d'outils et de services utilisés pour créer, gérer, sécuriser, et analyser les API (*Application Programming Interfaces*).

Ces solutions sont conçues pour aider les organisations à gérer efficacement leurs **interfaces de programmation**, surtout lorsque le nombre d'API augmente dans le cadre de la transformation numérique et de l'intégration de divers systèmes et services.

## Fonctionnalités Clés d'une Solution d'API Management.

---

**Portail pour Développeurs** : Offre une interface où les développeurs peuvent s'inscrire, découvrir, et utiliser les **API** disponibles.

**Sécurité et Authentification** : Intègre des mécanismes de sécurité comme **OAuth**, **JWT**, pour contrôler l'accès et protéger les **API**.

**Gestion du Trafic** : Permet de gérer le trafic des **API**, incluant le **throttling** (limitation) et la mise en cache pour optimiser les performances.

**Analyse et Reporting** : Fournit des outils pour surveiller l'utilisation des **API**, analyser les performances, et identifier les problèmes.

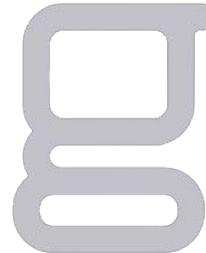
**Gestion du Cycle de Vie des API** : Supporte le développement et la maintenance des **API** à travers leur cycle de vie complet.

**Transformation et Orchestration** : Permet la transformation de requêtes et réponses, ainsi que l'orchestration des appels **API**.

## Gravitee : APIm opensource moderne et efficace .

---

Gravitee est une plateforme open source qui offre des fonctionnalités d'**API Management**. Elle permet aux organisations de gérer efficacement leurs APIs en fournissant un ensemble d'outils et de fonctionnalités.



Prérequis pour la mise en place de l'APIm **Gravitee** avec Docker

- **Docker**
- **Un éditeur de code**

# Installation Gravitee avec Docker.

---

## Procédure d'installation

Première connexion :

**login** : admin

**MDP** : admin



# Travaux-pratiques

Utiliser Gravitee pour créer une API

## Travaux -pratiques.

---

Reprendre l'API tutoriel développée précédemment (**simple CRUD application**) et déployer sur la plateforme **Gravitee**.

Tester les endpoints de l'API déployée sur **Postman**.