

Notes Formation

Comprendre l'architecture de microservices

Architecture en couches : du monolithe au microservices

Contraintes d'architecture des microservices

Gestion de l'authentification centralisée dans une architecture microservices

Intérêt d'une passerelle d'API

Gestion centralisée des traces

Qu'est ce qu'une architecture logicielle ?

Assemblage de plusieurs composants logiciels et matériels

Exemples de composants : appli web, web service, bdd, microservice, bibliothèque, appli mobile, serveur web, passerelle d'api,....

Quels styles d'architecture ?

- appels et retours : décomposition fonctionnelle
- en couches
- en flot de données
-

Architecture en couches : du monolythe au microservices

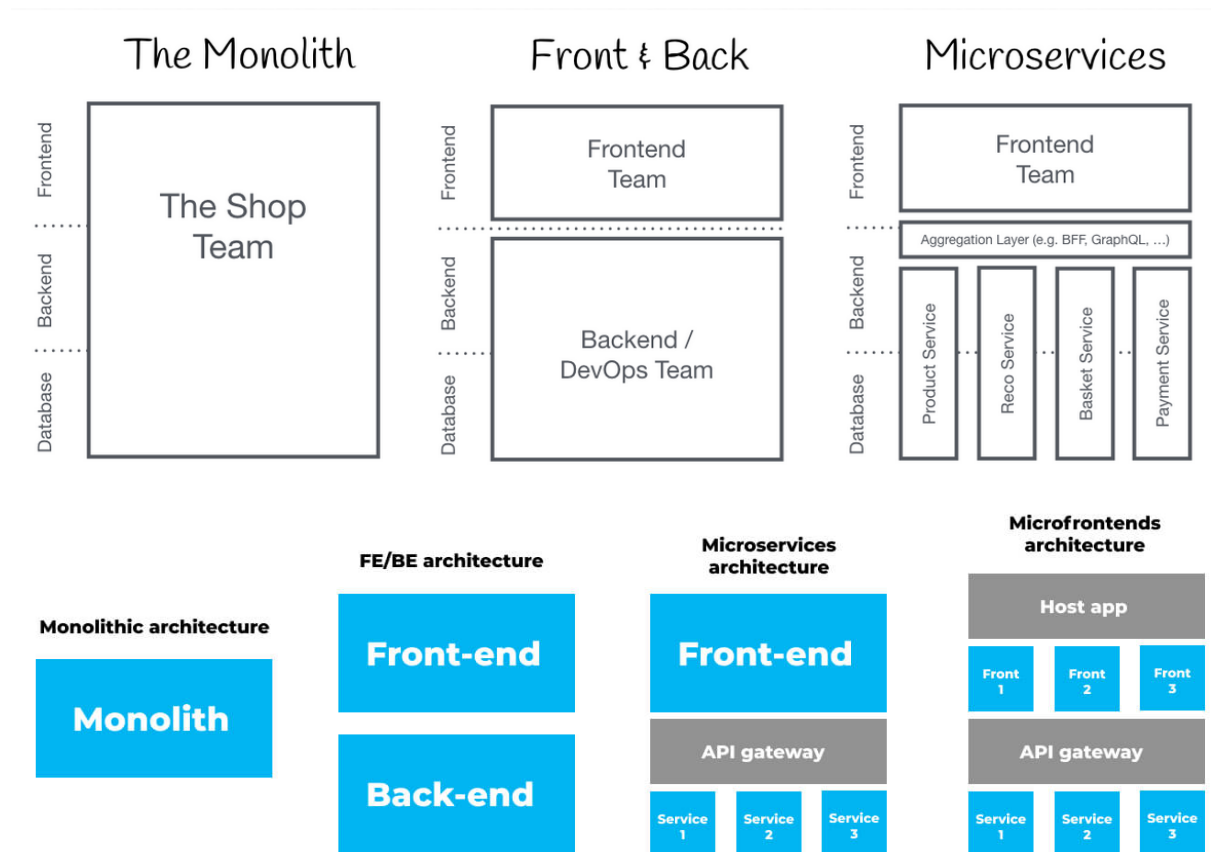
style architectural permettant d'organiser les couches d'une application : de la présentation à la persistance

Chaque couche a des "responsabilités distinctes"

Présentation : frontend Angular

API Web
(Service
Accès aux données)

Persistance
(bdd MariaDB)



Monolithe : un seul bloc regroupant les différentes couches
Av. simplicité, performance

Front/Back : découpage de la couche présentation puis métier

Microservices = services indépendants les uns des autres

Principe : découper les composants en de petits services autonomes qui peuvent être déployés et industrialisés séparément

Développement de microservices avec Spring Boot

Galaxie Spring : présentation, apports

Spring = ensemble de frameworks permettant de faciliter le développement

<https://spring.io/projects>

Spring Framework (Core, IoC) : brique de base qui apporte un “conteneur léger d’objets” qui instancie des objets au démarrage de l’application et les mets à la disposition de l’application.

Inversion de contrôle et injection de dépendances :

On ne contrôle plus la création des objets. On paramètre leur création.

@Component, @Service, @Repository, @Controller, @RestController, @Bean sur une méthode => annotation pour demandeur qu'un objet soit créé au démarrage de l'application
Par défaut, ces annotations créent un seul objet (singleton)
On peut définir des portées (@Scope) différentes : prototype, session, request,...

L'annotation @Autowired permet d'injecter un bean instancié au démarrage de l'application

```
@Service
public class ProductService {

    service;
}
```

```
@RestController
public class ProductController{
    @Autowired ProductService
}
```

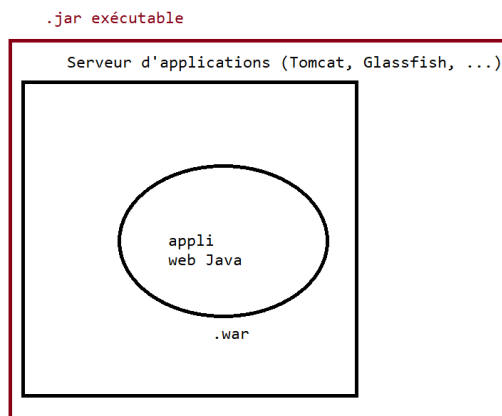
Spring Web : brique permettant de créer des applications/services web

Spring Data JPA : brique permettant de gérer la persistance des données avec SGBDR

Spring Boot : principe, fonctionnalités, pré-requis

framework permettant de faciliter la configuration et le déploiement d'une application Spring au travers de starters (templates de projet) préconfigurés

Il permet également la création d'un FAT JAR embarquant l'environnement d'exécution de l'application



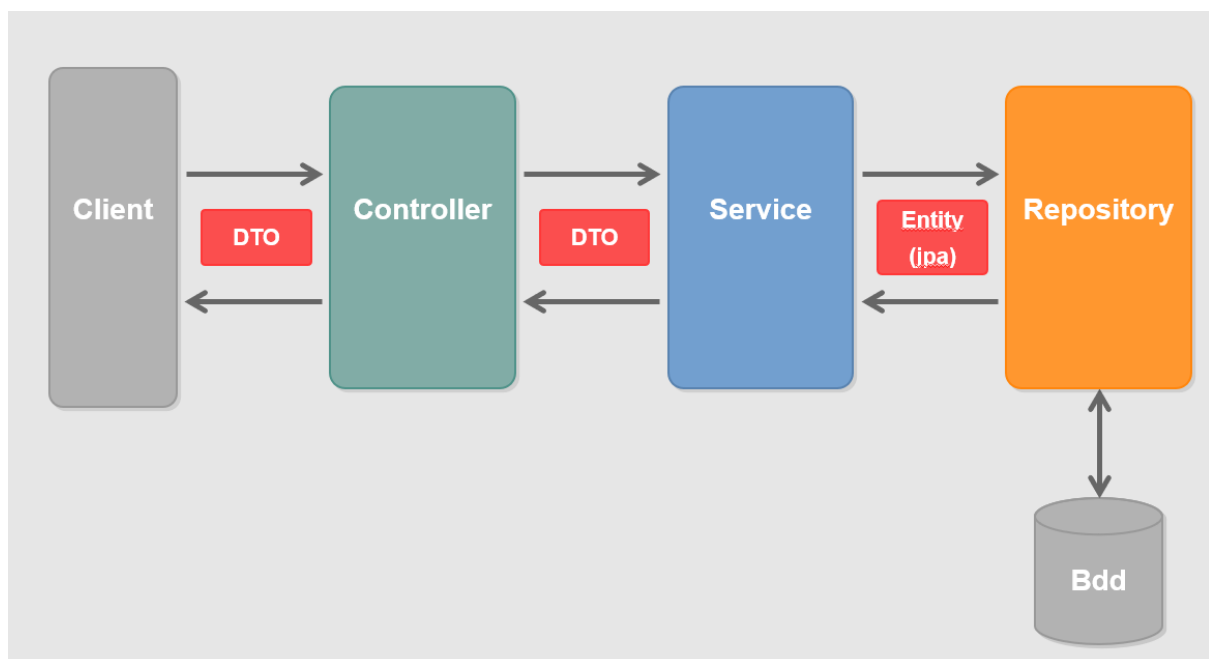
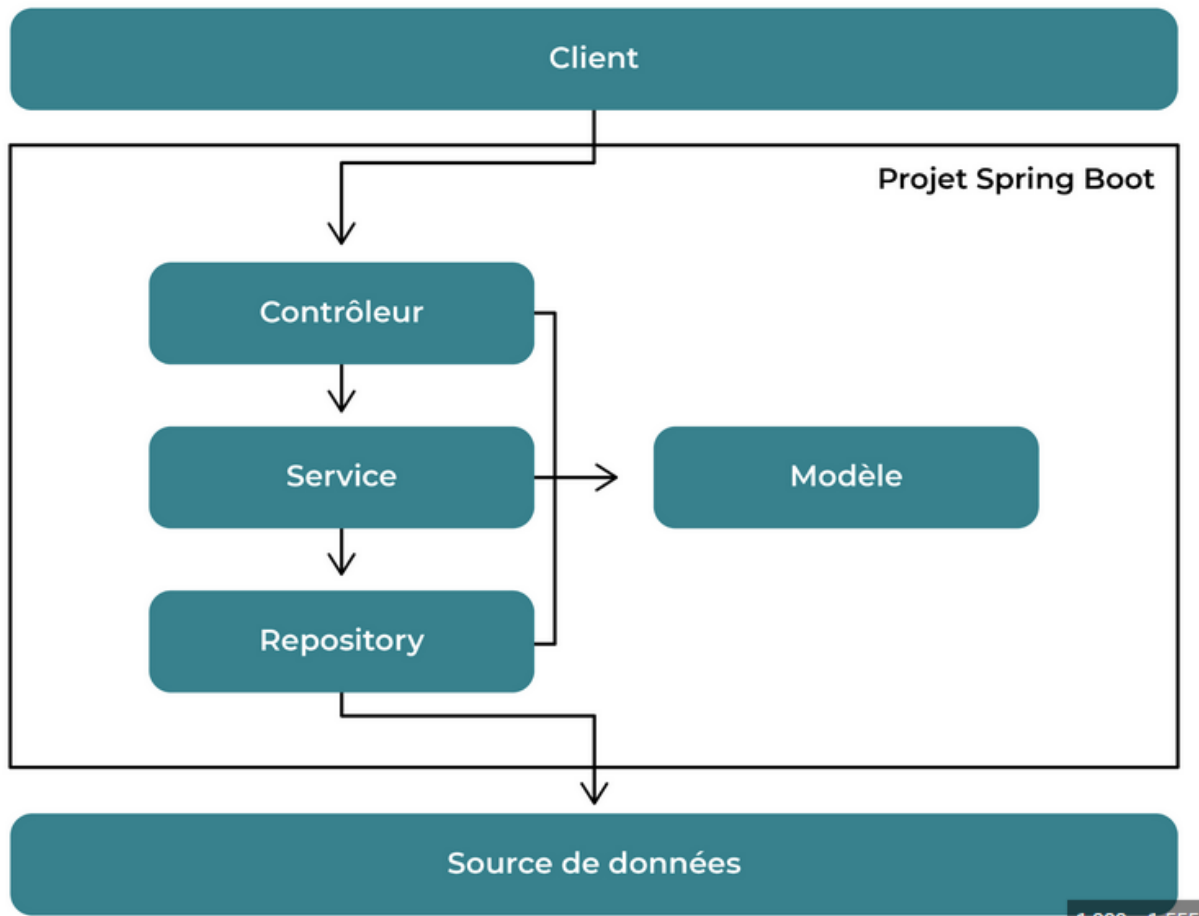
Création d'un projet : starters, gestion des dépendances, packaging

Le projet va utiliser plusieurs éléments :

- starter : modèle de projet avec un outil de build (Maven, Gradle)
<https://start.spring.io/>

Maven = outil de build Java permettant de gérer un projet (compilation, lancement de test, packaging, déploiement + gestion des dépendances)

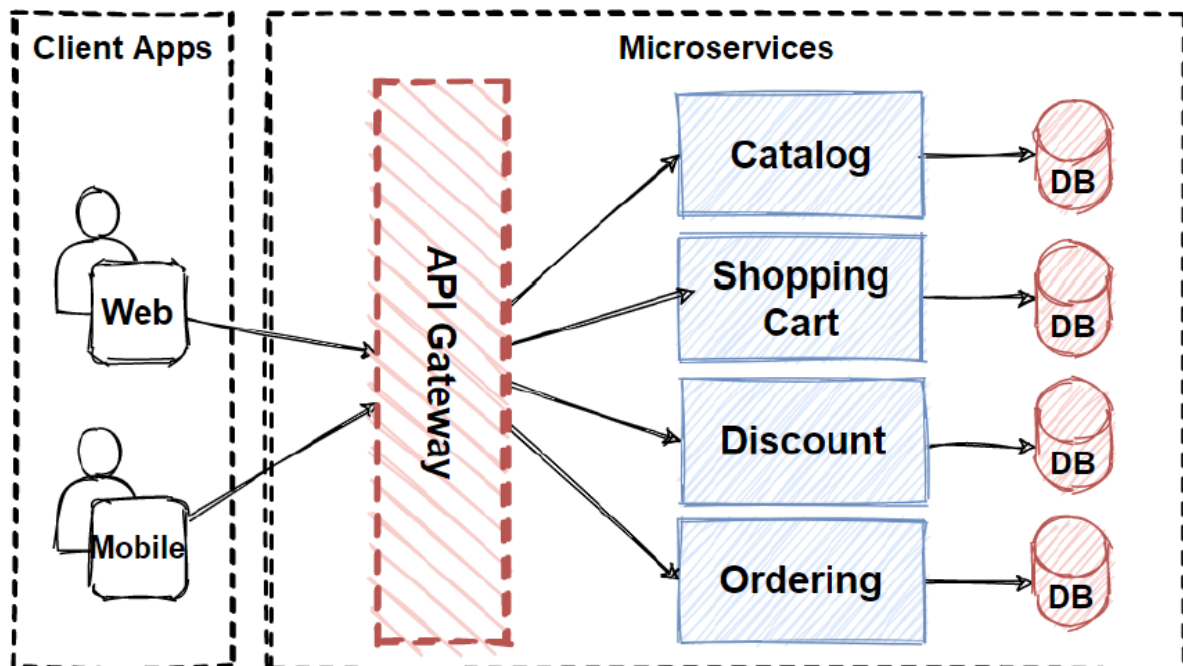
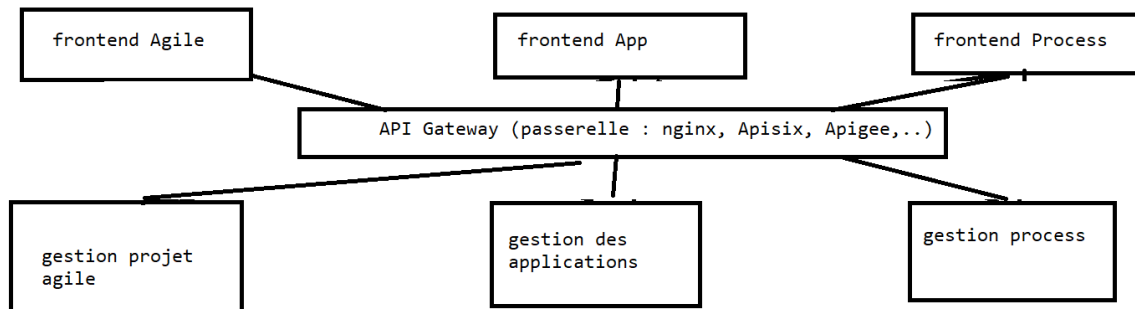
pom.xml : fichier de configuration de Maven



Architecture en couches orientée services web

Passerelle d'api : intermédiaire entre applications clientes et les différents microservices
Elle permet :

- le routage (rediriger les différentes req)
- l'authentification
- la sécurité : restriction d'usage



```

@Service
public class ProductService {
    @Lazy @Autowired CategoryService s;
}
  
```

```

@Service
public class CategoryService{
    @Autowired ProductService service;
}
  
```

si dépendance circulaire (A utilise B et B utilise A), on peut retarder le chargement d'une dépendance en utilisant l'annotation `@Lazy`

Web Service = Application web sans IHM qui expose un ensemble de méthodes

Il existe 2 types de services web :

- SOAP (Simple Object Access Protocol) : protocole de communication basé sur du XML
peut être véhiculé par HTTP, TCP, SMTP
Av. : synchrone/asynchrone, stateless/stateful, contrat formel
Inc. : lourdeur, à chaque changement du service, on doit prévenir l'ensemble des clients
- REST (Representational state transfer) : style d'architecture logicielle (pour utiliser des services web)
Il se base sur le protocole HTTP
Multiples formats de messages : JSON, XML, Texte, YML, binaire

Le client envoie une requête http qualifiée :
url de la ressource + méthode HTTP

<https://developer.mozilla.org/fr/docs/Web/HTTP/Methods>

Av. : plus léger, stateless
Inc. : synchrone, stateless (sans état uniquement)

Codes de retour HTTP : https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

Documentation d'une API REST : <https://springdoc.org/>

Open API Specification (Swagger) est la spécification de documentation d'une API
Elle décrit l'api rest au travers d'un fichier (JSON ou YML) qu'on peut visualiser/tester grâce à Swagger Editor

Il existe une bibliothèque : SpringDoc Open API UI

si spring boot < 3

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-ui</artifactId>

<version>1.6.14</version>

</dependency>

Si spring >=3 :

`<dependency>`

`<groupId>org.springdoc</groupId>`

`<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>`

`<version>2.1.0</version>`

`</dependency>`

Pour accéder au json :
/v3/api-docs

Pour l'éditeur : /swagger-ui.html

Si on souhaite modifier les chemins, il faut ajouter dans application.properties :

/api-docs endpoint custom path

springdoc.api-docs.path=/api-docs

swagger-ui custom path

springdoc.swagger-ui.path=/swagger-ui.html

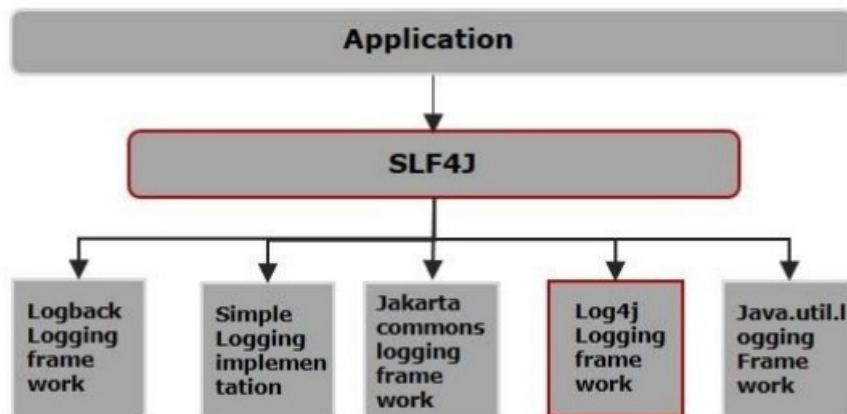
Gestion des logs en Java :

“Ecriture de traces”

Le jdk fournit une api permettant d'écrire les logs : java.util.logging

On peut utiliser des bibliothèques externes comme : Log4j, Logback, Commons-logging, etc...

SLF4j est une façade pour pouvoir écrire sur plusieurs systèmes utilisant des bibliothèques différentes :



Spring Boot => framework de logs utilisé => **Logback (considéré comme le successeur de log4j)**

Vocabulaire :

- **Logger** : Objet permettant d'écrire les logs

```
private static Logger myLogger =
LoggerFactory.getLogger(TestC.class);//root
private static Logger myLogger = LoggerFactory.getLogger("myLogger2");
```

Dans un logger, on peut avoir plusieurs appenders (où écrire)

- **Appender** : support de logs (ConsoleAppender, RollingFileAppender, ..)
<https://logback.qos.ch/manual/appenders.html>
- **Layout** : type de support (texte par défaut, on peut faire du XmlLayout)
- **Pattern** : format du message %date %message
- **Level** : Niveau du message (info, warning,)

Mise en place :

- Logback est déjà chargé par le starter de Spring Boot
=> aucune dépendance à ajouter
- configuration des loggers à mettre soit dans .properties/.yaml
ou dans un fichier logback.xml

Un contrôleur REST est une classe annotée @RestController

Si on souhaite factoriser une configuration du contrôleur au niveau de la classe :

@RequestMapping

Le contrôleur étant un bean géré par le conteneur Spring, on peut injecter des dépendances avec `@Autowired` Type objetPresentDansLeConteneur;

Le contrôleur pourra contenir plusieurs méthodes annotées :
`@RequestMapping` et on précise la méthode (GET, POST,)
ou `@GetMapping`, `@PostMapping`, ...

Une méthode peut être mappée avec plusieurs URLs : prop value = { `"/test/m1"`, `"/test/m1/{page}"` }

Le type Mime de retour/d'entrée peut être renseigné dans `"produces"` / `"consumes"`

Toute méthode peut avoir des paramètres :

- dans l'url : `//{nomParam}`, le mapping s'effectue avec `@PathVariable`
- dans l'url avec des paramètres nommés (`?p1=23`), le mapping s'effectue avec `@RequestParam`

- dans le corps de la requête : `@RequestBody`

Le retour devrait être soit un type primitif (déconseillé), soit un objet, soit un `ResponseEntity<TypeObjet>`.

L'intérêt de `ResponseEntity` est de préciser des informations complémentaires : code de retour HTTP, entêtes (headers) ou autre

On peut retourner un `ResponseEntity<Resource>` si on veut envoyer un flux binaire (download)

La gestion des erreurs peut s'effectuer à n'importe quel endroit du traitement métier en capturant et traitant l'exception (try/catch) ou en remontant l'exception (throw/throws) et en définissant un `ExceptionHandler` global par type d'exception capturée.

Le Cross-origin Resource Sharing (CORS) = un mécanisme qui permet à des ressources restreintes d'être récupérée par un autre domaine extérieur.

Par défaut votre webservice n'est accessible que par le même domaine.

On doit configurer notre stratégie de CORS

(client)
entête Access-Control-Allow-Origin

(service)
configuration du CORS

La configuration du CORS côté serveur peut être réalisée, soit au niveau du contrôleur, soit au niveau d'une méthode du contrôleur :

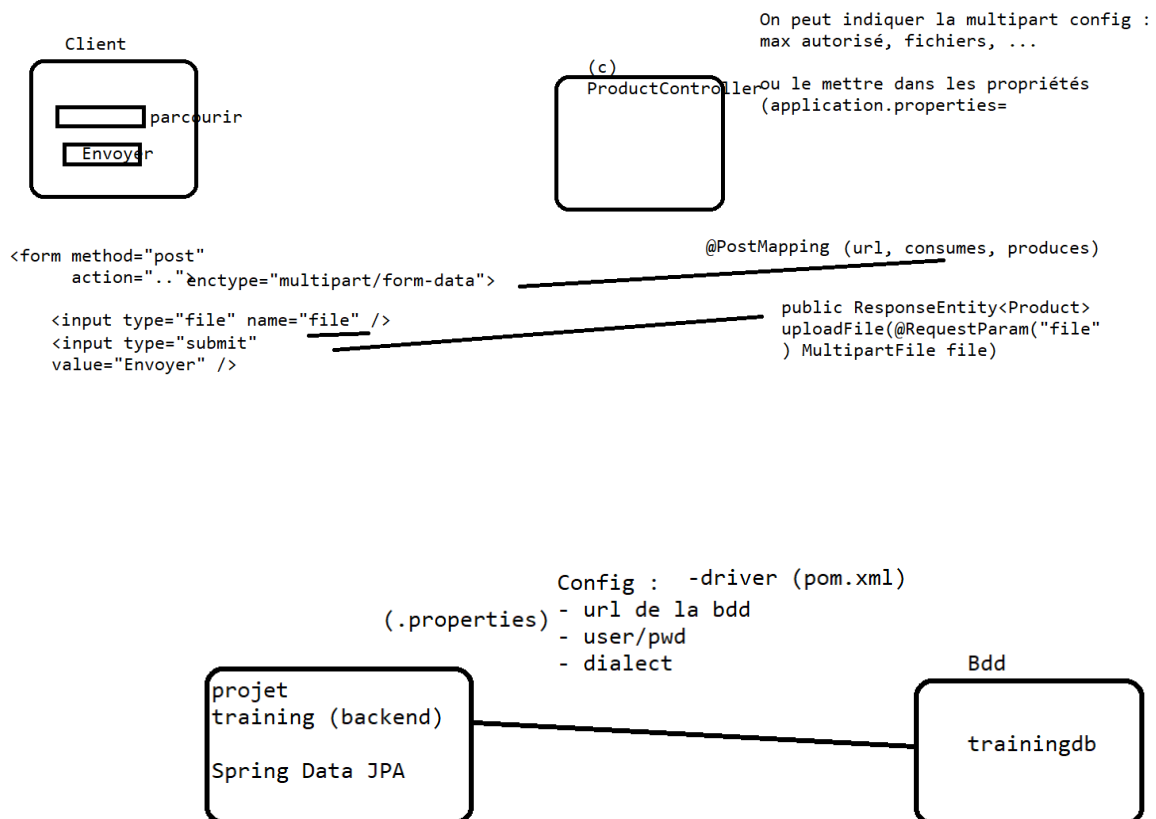
`@CrossOrigin(origins = {"domaine1"})`

<https://spring.io/guides/gs/rest-service-cors/>

En plus du paramétrage “fin”, on peut faire une configuration plus globale au niveau du “WebMvcConfigurer” (bean qui a été configuré automatiquement par Spring au démarrage.

<https://spring.io/guides/gs/rest-service-cors/#global-cors-configuration>

UPLOAD D’UN FICHIER :



d'interfaces pour faire du mapping relationnel-objet)

Spring Data JPA est une implémentation de JPA qui embarque le moteur "Hibernate"

Spring Data JPA = surcouche d'Hibernate (implements) JPA

Apports :

- annotations de mappings entités-tables
- un mécanisme pour réaliser des requêtes JP-QL ou SQL natif
- un ensemble de repositories génériques prêts à l'emploi

Mapping des entités

@Entity : mapping de la classe avec une table qui porte le même nom

@Table : nom de table, schéma

Schéma = regroupement de plusieurs tables/views/objets

Si on souhaite ignorer un attribut => @Transient

* simple : @Id

- ## La génération d'une clé @GeneratedValue

On peut factoriser des colonnes dans une classe mère annotée @MappedSuperclass

(G)

```

1
Bureau
100
80
0

```

select * from
product where id=1
with uplock

update product
set prix=80 where
id=1 AND version=0

Bdd

```

trainingdb
product
1 Bureau 100
@Version int
version
0 1

```

StaleObjectException
ObjectOptimisticLockingFa
ilureException
gestion de la concurrence :

- ne rien faire (la dernière req
exécutée appliquera les modifs)
- on verrouille la ligne "Lock" annotation
(verrous pessimiste)
- verrous optimiste qui permet de
versionner la ligne

(S)

```

1
Bureau
100
90
0

```

update product
set prix=90 where
id=1 AND version=0

1) Relation 1 à n

Une catégorie peut contenir plusieurs produits
Un produit appartient à une seule et unique catégorie

table product

id	description	price	category_id
1	bureau	100	23

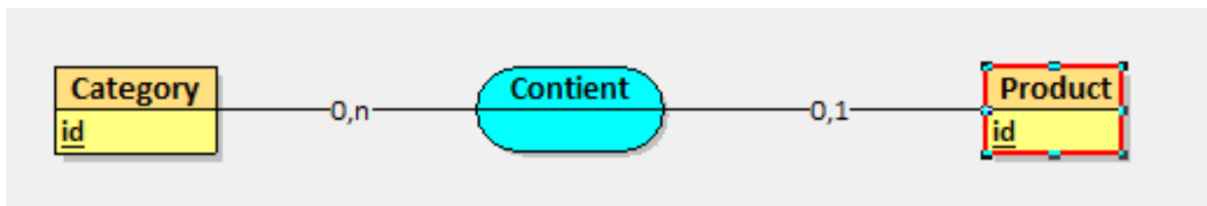
table category

id	name
23	Mobilier

En bdd : on aura une clé étrangère (category_id) dans la table product

@ManyToOne : many products to one category

Modèle Entité-Association :



Modèle physique de données :

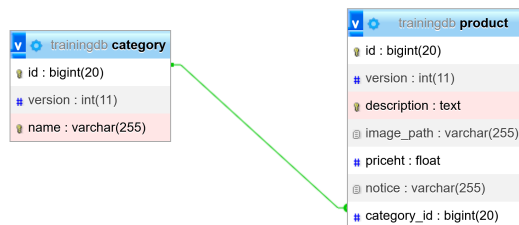
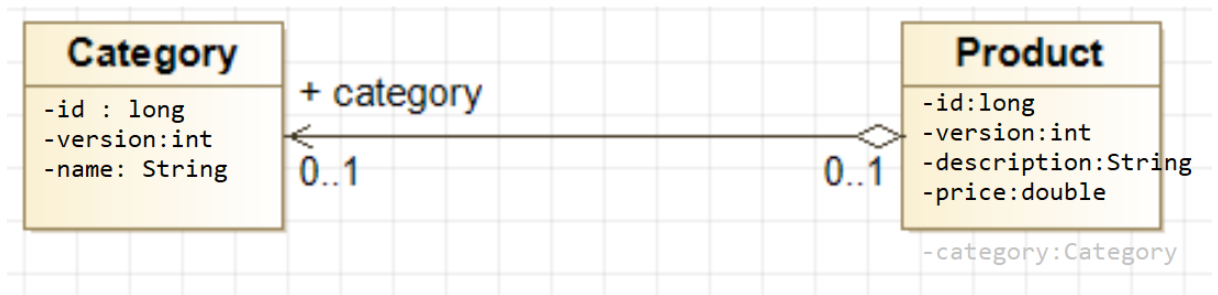


Diagramme de classe :



Il existe 2 types de chargement des associations :

LAZY : chargement tardif

par défaut sur le OneToMany, ManyToMany

EAGER : chargement immédiat

par défaut dans OneToOne, ManyToOne

EAGER :

produit 1 (1 bureau , 100, ... category (23, Mobilier))

LAZY :

produit 1 (1 bureau , 100, ... category:null)

ManyToMany 'relation plusieurs à plusieurs'

Modèle entité-association :

Fournisseur —0,n----- (possède) -----0,n----- Produit

Modèle physique de données :

1 table de jointure possédant des clés étrangères pointant vers la clé primaire de chacune des tables

fournisseur
id name

fournisseur_produit
fournisseur_id, produit_id

produit
id, description



Diagramme de classe :

- Supplier : long id, int version, String name, Set<Product> products
- Product :, Set<Supplier> suppliers

Mapping JPA :

@ManyToMany sur chacune des entités avec un mappedBy du côté de products

NB. : si on oublie de mettre le mappedBy, on aura 2 tables de jointures(erreur)

Relation 1 à 1 (OneToOne)

Player <====0,1==== est encadré <====0,1==== Manager

3 représentations possibles en Bdd :

Sol1 : une seule table

player (id, name, managerName)

@Embeddable © Manager

© Player : @Embedded private Manager manager;

Sol2 : association par clé primaire

2 tables séparées

player

24 pl1

manager

24 M1

@OneToOne @MapsId que d'un côté

Solution 3 : association par clé étrangère avec une contrainte d'unicité

player	managerId (U)	manager
1 PL1	24	24 M1
2 PL2	24	

@OneToOne @OneToOne
+ contrainte d'unicité sur la colonne

<https://docs.oracle.com/javaee/6/api/javax/persistence/OneToOne.html>

Mapping des collections simples : @ElementCollection

```
private List<String> col;
```

Mapping de l'héritage : @Inheritance (3 stratégies possibles)

Repositories

Services

DTO

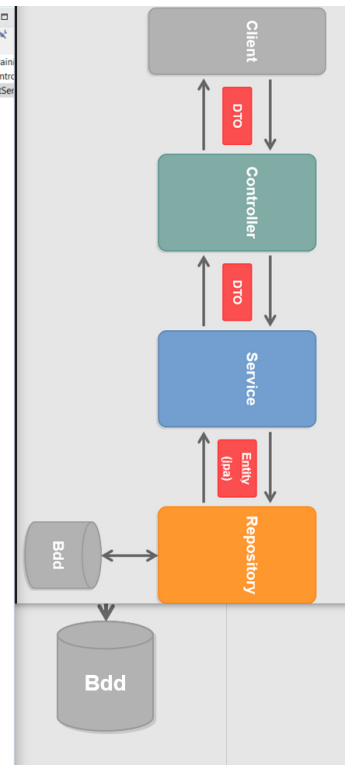
Organisation des couches :

```
ProductController.java
7
8 @RestController
9 public class ProductController {
10     @Autowired
11     private IProductService productService;
12 }

IProductService.java
4
5 import fr.dawan.training.dto.ProductDto;
6
7 public interface IProductService {
8     List<ProductDto> getAllBy(int page, int size, String search) throws Exception;
9 }

ProductServiceImpl.java
12 @Service
13 @Transactional
14 public class ProductServiceImpl implements IProductService {
15     @Autowired
16     private ProductRepository productRepository;
17 }

ProductRepository.java
1 package fr.dawan.training.repositories;
2
3 import java.util.List;
4
5 @Repository
6 public interface ProductRepository extends JpaRepository<Product, Long>{
7
8     //FROM Product p WHERE p.description LIKE %:description%
9     Page<Product> findAllByDescriptionContaining(String description, Pageable pageable);
10
11     long countByDescriptionContaining(String description);
12
13     //Requête avec JP-QL
14     @Query(value = "FROM Product p WHERE p.category.id = :id")
15     List<Product> findAllByCategoryId(@Param("id") long id);
16 }
```



Intercepteur de la couche ORM

Envoi de mails

Spring Security

Connexion / Mdp oublié

Event

Async

Scheduled

Appel de web services















AUTRES DÉFINITIONS

Recherche des vulnérabilités sur les librairies : (MITRE / CVE)
<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=>

Modèle OSI

LE MODÈLE OSI

LE MODÈLE OSI PEUT ÊTRE CONSIDÉRÉ COMME UN LANGAGE UNIVERSEL POUR LES RÉSEAUX INFORMATIQUES. IL EST BASÉ SUR LE CONCEPT CONSISTANT À DIVISER UN SYSTÈME DE COMMUNICATION EN SEPT COUCHES ABSTRAITES, EMPILÉES LES UNES SUR LES AUTRES.

7		COUCHE APPLICATION	Point de contact avec les services réseaux	 DONNÉES	TELNET, FTP, HTTP, SMTP, ETC.
6		COUCHE PRÉSENTATION	Préparation des données pour la présentation (formatage, chiffrement, encodage etc.)	 DONNÉES	HTML, DOC, MP3, JPEG, ETC.
5		COUCHE SESSION	Organisation de la session de communication (points de contrôle, etc.)	 DONNÉES	SIP, RTP, ETC.
4		COUCHE TRANSPORT	Coordination du transfert des segments (numéro de port, contrôle réception, etc.)	 SEGMENTS	TCP, UDP, SSL, TLS, ETC.
3		COUCHE RÉSEAU	Routage des paquets entre les noeuds d'un réseau	 PAQUETS	IP, ARP, ETC.
2		COUCHE LIAISON	Assure le transfert des trames de noeud à noeud	 TRAMES	ETHERNET, PPP, ETC.
1		COUCHE PHYSIQUE	Transmission des bits	 BITS	MULTIPLEXING, MODULATION, ETC.

RÉALISÉ PAR



ExceptionHandler
LogController

Interface définit un contrat à respecter
pseudo classe abstraite, non instanciable directement

contient des signatures de méthodes, des méthodes statiques, des méthodes avec un corps par défaut.

```
public interface Pliable {  
    void plier();  
    void deployer();  
    default void pivoter(){  
        //traitement  
    }  
}
```

Une classe peut implémenter une ou plusieurs interfaces

```
public class Chaise implements Pliable {  
    //...  
    public void plier() {  
    }  
    public void deployer() {  
    }  
}
```

```
Chaise ch = new Chaise();
```

```
Pliable p = new Chaise();
```

```
p = new Table();
```

Pliable p2 = new Pliable(); // ON NE PEUT PAS INSTANCIER UNE INTERFACE

On peut créer une classe anonyme :

```
Pliable p3 = new Pliable(){  
    public void plier(){  
    }  
  
    public void deployer(){  
    }  
}
```

Base64 : mécanisme de codage de l'information en utilisant 64 caractères

On pourra ainsi convertir une chaîne/fichier <> base64 et inversement

dans java.util, une classe Base64 est disponible, offrant 2 méthodes :
getEncoder().encode(...) pour convertir vers Base64
getDecoder().decode(...) pour convertir depuis le base64 vers byte[]

 //logo.png est le fichier physique

en base64 :

Wamp server regroupe plusieurs services :

- 1 serveur web Apache HTTP Server
- 1 serveur MySQL
 - 1 serveur MariaDb

Sur le serveur web, est déployé une application Php (PhpMyAdmin) qui permet d'administrer les services de MySQL/MariaDB

Généricité : concept permettant de réutiliser un traitement pour différents types de données

<un générique>, on peut avoir plusieurs génériques

- Classe générique :

```
public class Calcul<T>{  
    T a;  
    T b;  
  
    public void permuter(){  
        T c = a;  
        a = b;  
        b = c;  
    }  
}
```

Calcul<Integer> cl = new Calcul<>();

```
cl.a = 23;  
cl.b = 58;  
cl.permuter();
```

```
Calcul<Product> cl2 = new Calcul<>();  
cl2.a = new Product(...);  
cl.b = new Product(...);  
cl.permuter();
```

```
@Service
```

```
@Transactional
```

```
public class ProductServiceImpl implements IProductService {
```

```
    @Autowired
```

```
    private ProductRepository productRepository;
```

```
    @Autowired
```

```
    private CategoryRepository categoryRepository;
```

```
    @Override
```

```
    public List<ProductDto> getAllBy(int page, int size, String  
description) throws Exception {
```

```
        List<ProductDto> result = new ArrayList<>();
```

```
        List<Product> products = productRepository
```

```
.findAllByDescriptionContaining(description, PageRequest.of(page, size))
```

```
.getContent();
```

```
    for(Product p : products) {  
        result.add(DtoTools.convert(p, ProductDto.class));  
    }
```

```
    return result;
```

```
}
```

```
@Override
```

```
public LongDto countBy(String description) throws Exception {
```

```
    long nb =  
productRepository.countByDescriptionContaining(description);
```

```
    LongDto result = new LongDto();
```

```
    result.setResult(nb);
```

```
    return result;
```

```
}
```

```
@Override
```

```
public ProductDto getById(long id) throws Exception {
```

```
    Optional<Product> opt = productRepository.findById(id);
```

```
    if(opt.isPresent())
```

```
        return DtoTools.convert(opt.get(), ProductDto.class);
```

```
        return null;
    }

    @Override
    public ProductDto saveOrUpdate(ProductDto pDto) throws
    Exception {

        Product prod = DtoTools.convert(pDto, Product.class);

        //category association because we have only the categoryId

        Optional<Category> optC =
        categoryRepository.findById(pDto.getCategoryId());

        if(optC.isPresent())

            prod.setCategory(optC.get());

        prod = productRepository.saveAndFlush(prod);

        return DtoTools.convert(prod, ProductDto.class);
    }
}
```

