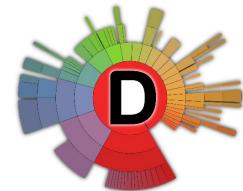


# Web- ESIR 2

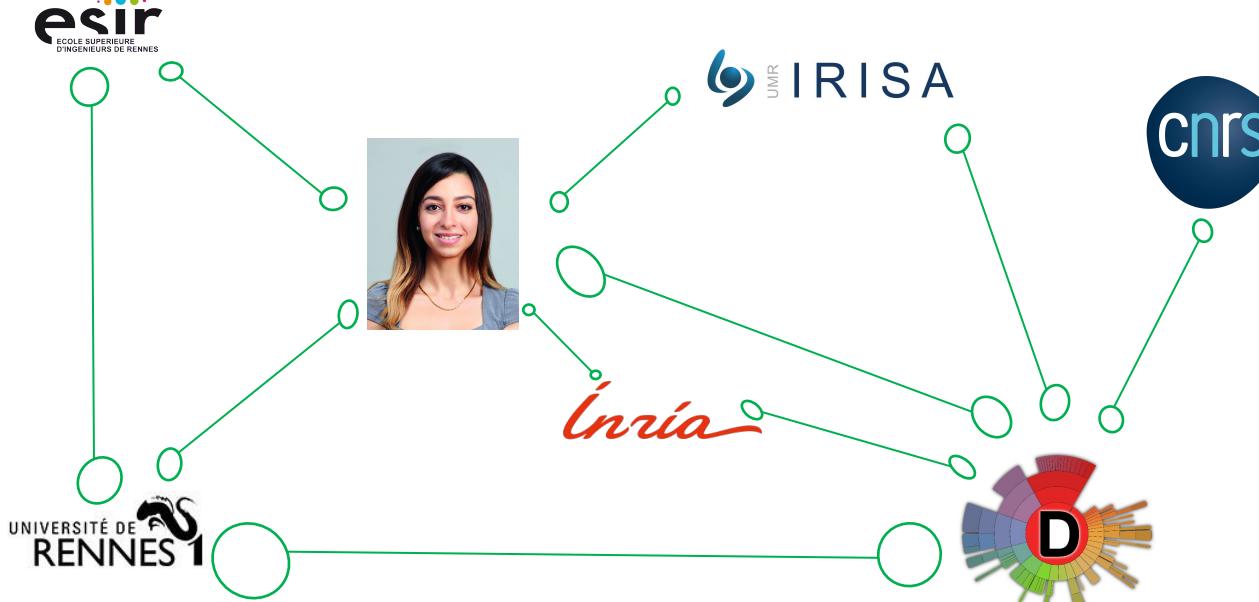
Cours développement Web - partie Backend

octobre 2022



# Qui suis-je ?

- **Maître de Conférences @ Université de Rennes 1**
  - DiverSE team (University of Rennes, IRISA, Inria)



🌐 <https://stephaniechallita.github.io/>

🌐 <https://www.diverse-team.fr/>

# Organisation générale

- 5 CMs sur le Backend
- 4 CMs sur le Frontend
- 14 TPs -> Projet

# Organisation - partie Backend

- 5 cours (10h)
- 7 séances projet (14h)

- Lien vers les slides (pdf) :  
<https://stephaniechallita.github.io/web/WebServer-ESIR.pdf>
- Lien vers les détails du module :  
<https://stephaniechallita.github.io/web/>
- Lien vers le dépôt du guide du projet :  
<https://github.com/stephaniechallita/WebServer>

# Organisation - partie Frontend

- 4 cours (8h)
- 7 séances projet (14h)

- Lien vers les slides :

<https://stephaniechallita.github.io/web/>

- Lien vers le dépôt du guide du projet :

[https://gitlab.istic.univ-rennes1.fr/hfeuilla/jxc\\_fradministrationfront](https://gitlab.istic.univ-rennes1.fr/hfeuilla/jxc_fradministrationfront)

# TP & Évaluation

- Projet guidé en binôme : soutenance de présentation et de compréhension (2h)

## Partie Backend

- Premiers pas avec NestJS
- Contrôleurs et première API
- Modules et logique métier
- TypeORM, Repository et données
- OpenAPI
- Tester son backend NestJS
- Sécurité
- Développement

# TP & Évaluation

- Projet guidé en binôme : soutenance de présentation et de compréhension (2h)

## Partie Frontend

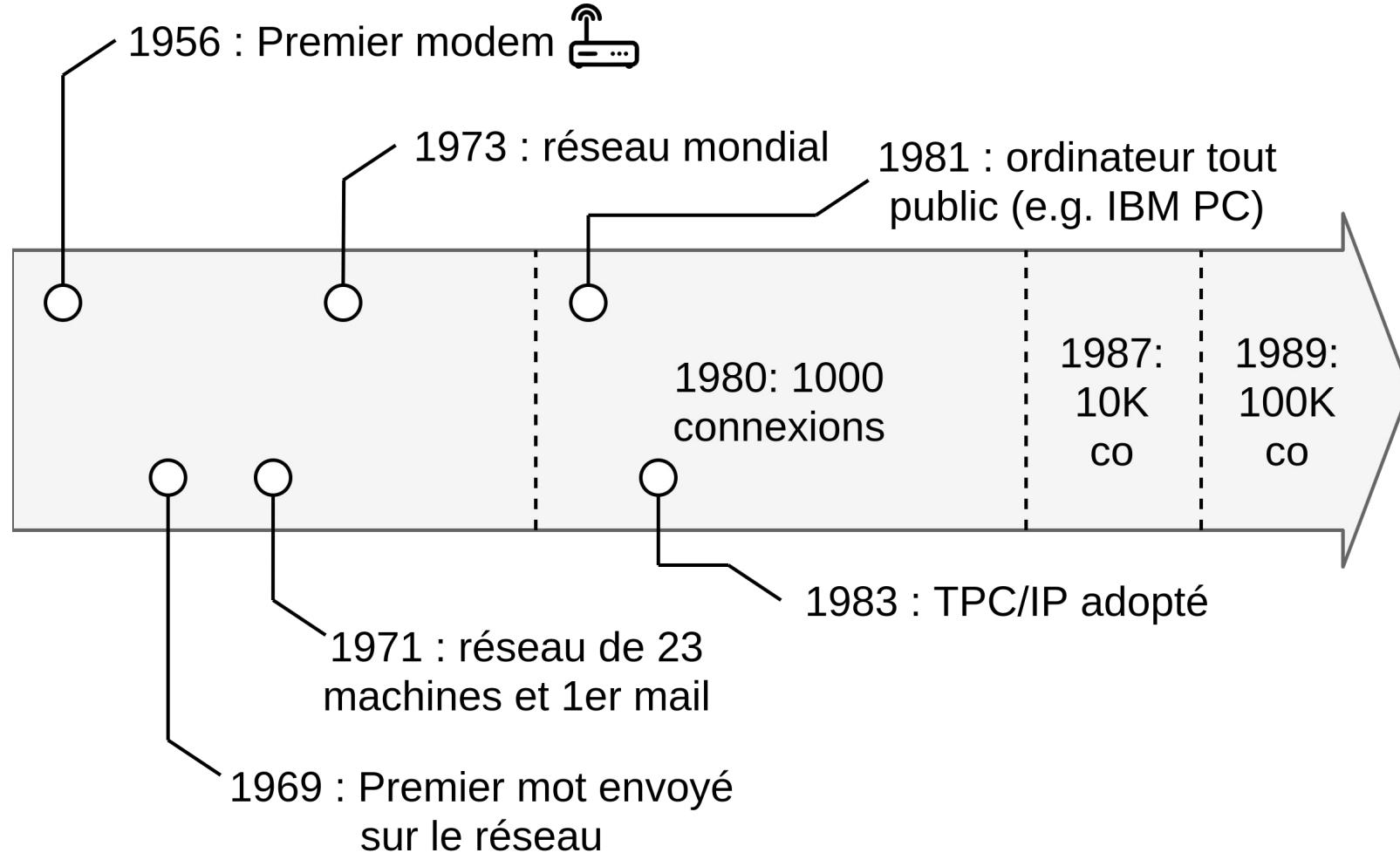
- S'authentifier en tant qu'utilisateur
- Gérer un utilisateur :  
    création, mise à jour et suppression
- Gérer une association :  
    création, mise à jour et suppression
- Lister les utilisateurs et les associations
- Rechercher un utilisateur ou une association

# Table des matières

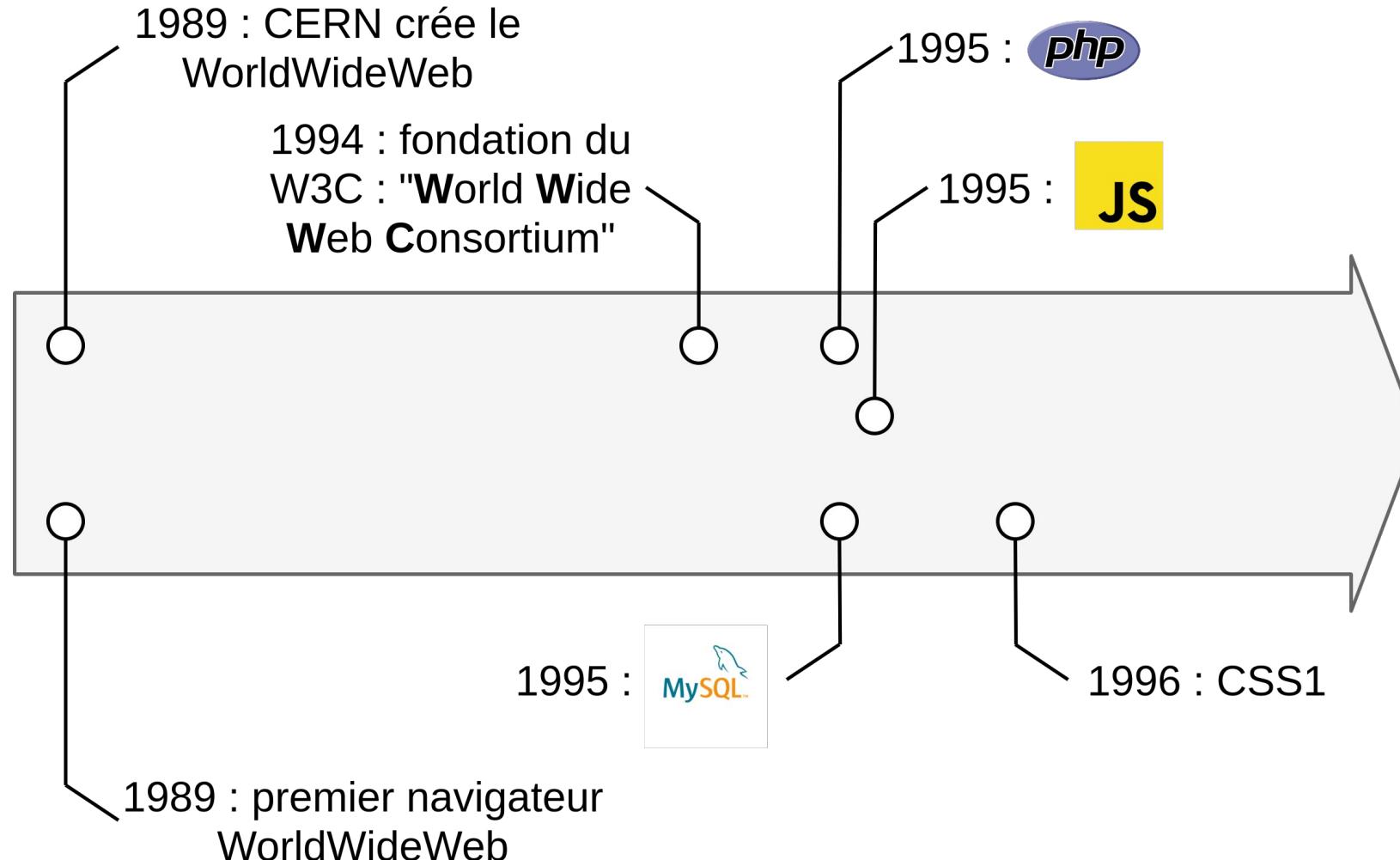
- Généralités
- Coup d'oeil sur le frontend (côté client)
- Le backend en détails (côté server)
- Un peu de technique avec NestJS
- De la base de données aux objets (ORM)
- API REST, OpenAPI & bonnes pratiques
- Sécurité : Autorisation & Authentification

# Généralités

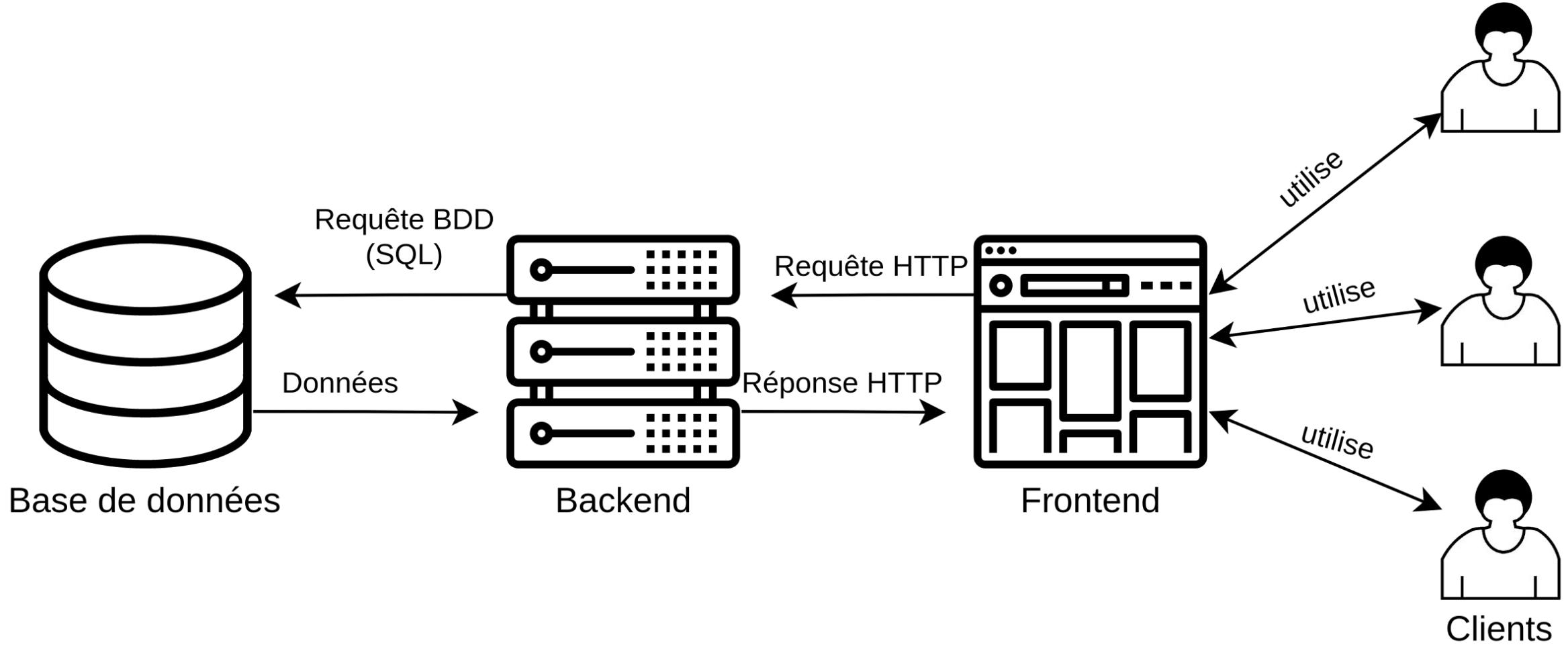
# Généralités : histoire du web



# Généralités : histoire de la programmation web



# Généralités : rappel d'architecture



# Généralités : technologies modernes

- Frontend (Web) : Le plus souvent en JS, ou un framework construit au-dessus. Les plus connus sont : React, Angular et Vue.js.
- Frontend (Mobile) : Applications natives (Kotlin pour Android ou Swift pour iOS) ou cross-plateformes : ReactNative ou Flutter
- Backend : NodeJS (Express, Koa), TypeScript (NestJS), Java (Spring Boot, Vert.x), Python (Django, Flask)
- Base de données : MySQL, PostgreSQL, MariaDB, MongoDB
- Cloud-native et microservization

# Motivations : Pourquoi ce cours est important ?

- Le web est aujourd’hui omniprésent
- Besoins : en 2017 10000 développeurs<sup>1</sup>
- Innovations
- Diversité

1: <https://www.cefim.eu/blog/formation/10-raisons-de-devenir-developpeur-web/>

# Objectif du cours

- Fondements des backend web : architectures & composants principaux
- Gestion des données
- Panorama des autres aspects du développement Web
- Introduction à TypeScript

# Ce que ce cours n'est pas !

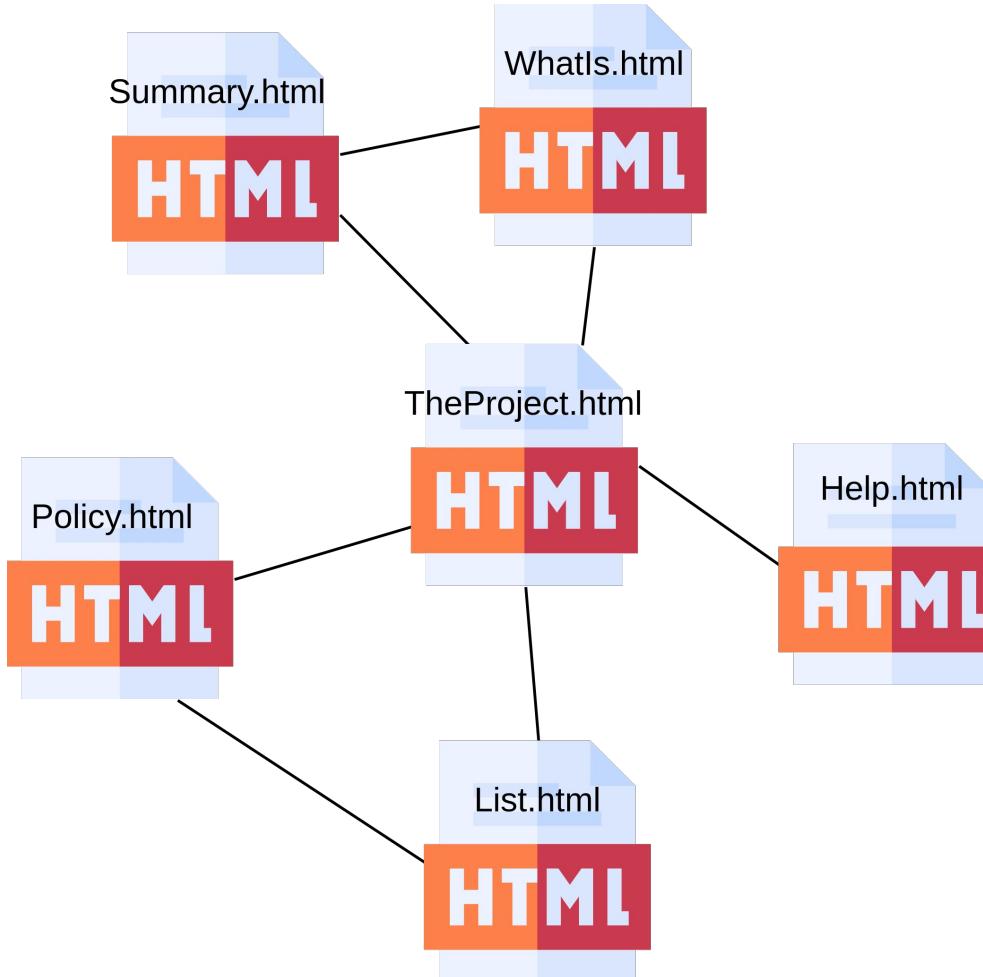
- Un cours fullstack développeur
- Un cours de TypeScript

# Coup d'oeil sur le Frontend

# Coup d'oeil sur le frontend

- HTML & CSS, la base
- Du dynamisme avec JavaScript
- Single Page Application vs Multiple Page Application

# HTML



```
1 <HEADER>
2 <TITLE>The World Wide Web project</TITLE>
3 <NEXTID N="55">
4 </HEADER>
5 <BODY>
6 <H1>World Wide Web</H1>The WorldWideWeb (W3) is a wide-area<A
7 NAME=0 HREF="WhatIs.html">
8 hypermedia</A> information retrieval
9 initiative aiming to give universal
10 access to a large universe of documents.<P>
11 Everything there is online about
12 W3 is linked directly or indirectly
13 to this document, including an <A
14 NAME=24 HREF="Summary.html">executive
15 summary</A> of the project, <A
16 NAME=29 HREF="Administration/Mailing/Overview.html">Mailing lists</A>
17 , <A
18 NAME=30 HREF="Policy.html">Policy</A>, November's <A
19 NAME=34 HREF="News/9211.html">W3 news</A>,
20 <A
21 NAME=41 HREF="FAQ/List.html">Frequently Asked Questions</A> .
```

# HTML : premier site web mis en ligne

## World Wide Web

The WorldWideWeb (W3) is a wide-area [hypermedia](#) information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an [executive summary](#) of the project, [Mailing lists](#) , [Policy](#) , November's [W3 news](#) , [Frequently Asked Questions](#) .

### What's out there?

Pointers to the world's online information, [subjects](#) , [W3 servers](#), etc.

### Help

on the browser you are using

### Software Products

A list of W3 project components and their current state. (e.g. [Line Mode](#) ,[X11 Viola](#) , [NeXTStep](#) , [Servers](#) , [Tools](#) , [Mail robot](#) , [Library](#) )

### Technical

Details of protocols, formats, program internals etc

### Bibliography

Paper documentation on W3 and references.

### People

A list of some people involved in the project.

### History

A summary of the history of the project.

### How can I help ?

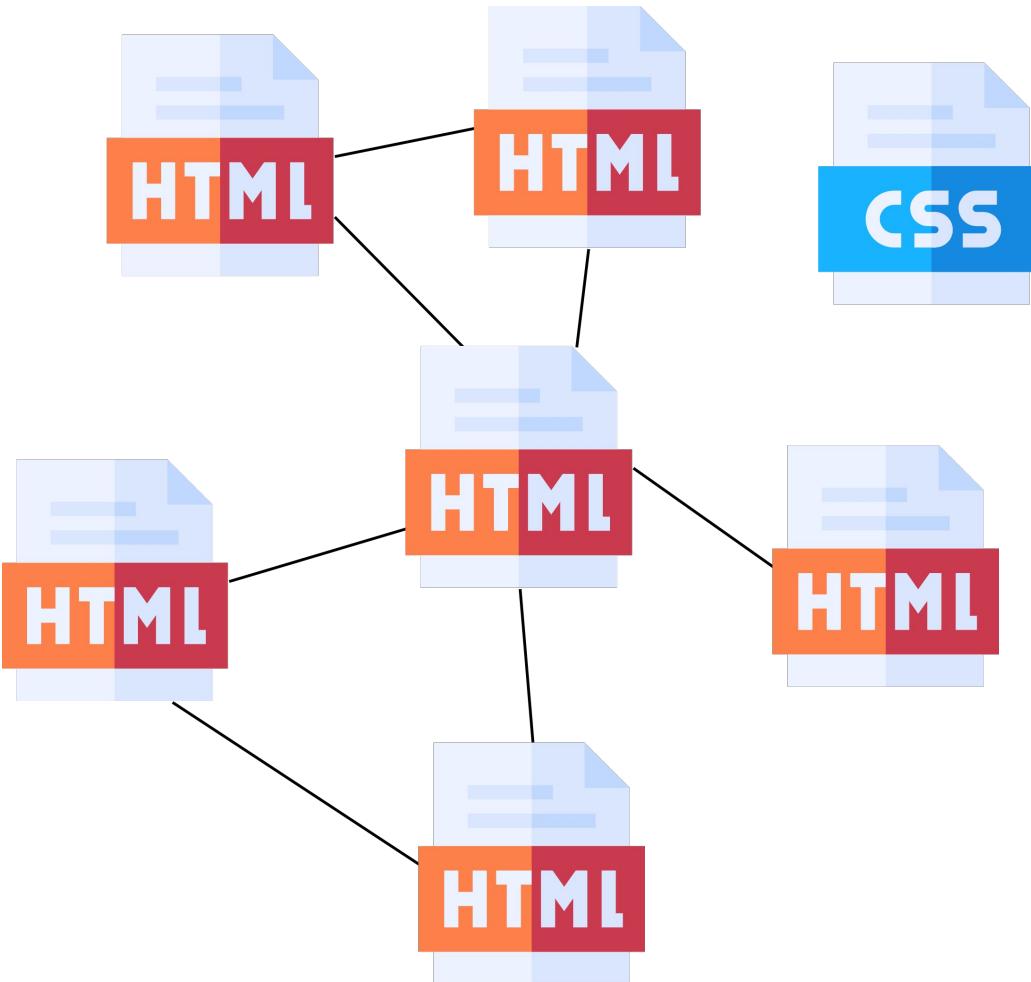
If you would like to support the web..

### Getting code

Getting the code by [anonymous FTP](#) , etc.

source : <https://www.w3.org/History/19921103-hypertext/hypertext/WWW/TheProject.html>

# HTML & CSS



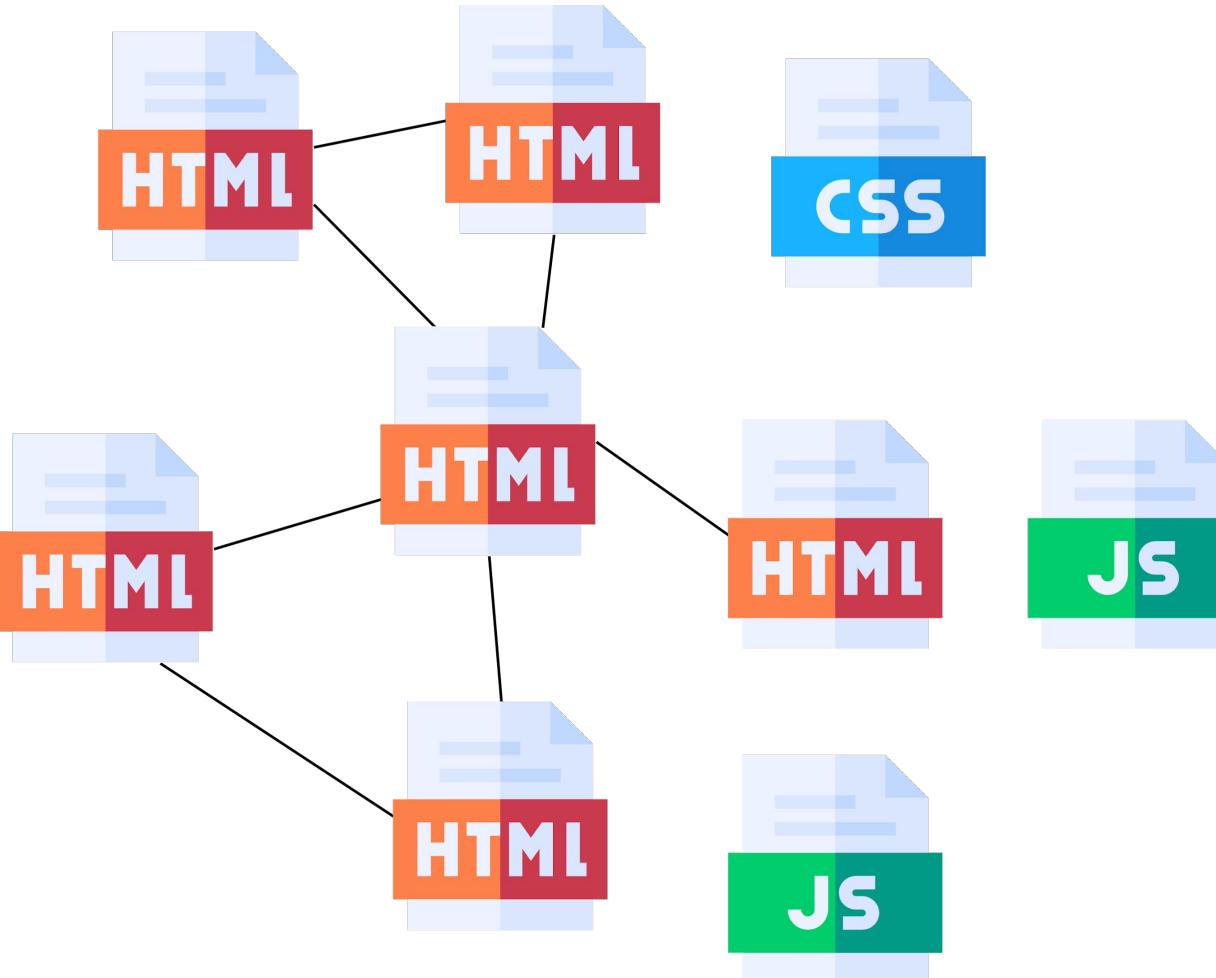
```
h1 {  
    font-family: courier, courier-new, serif;  
    font-size: 20pt;  
    color: blue;  
    border-bottom: 2px solid blue;  
}  
p {  
    font-family: arial, verdana, sans-serif;  
    font-size: 12pt;  
    color: #6B6BD7;  
}  
.red_txt {  
    color: red;  
}
```

# HTML & CSS



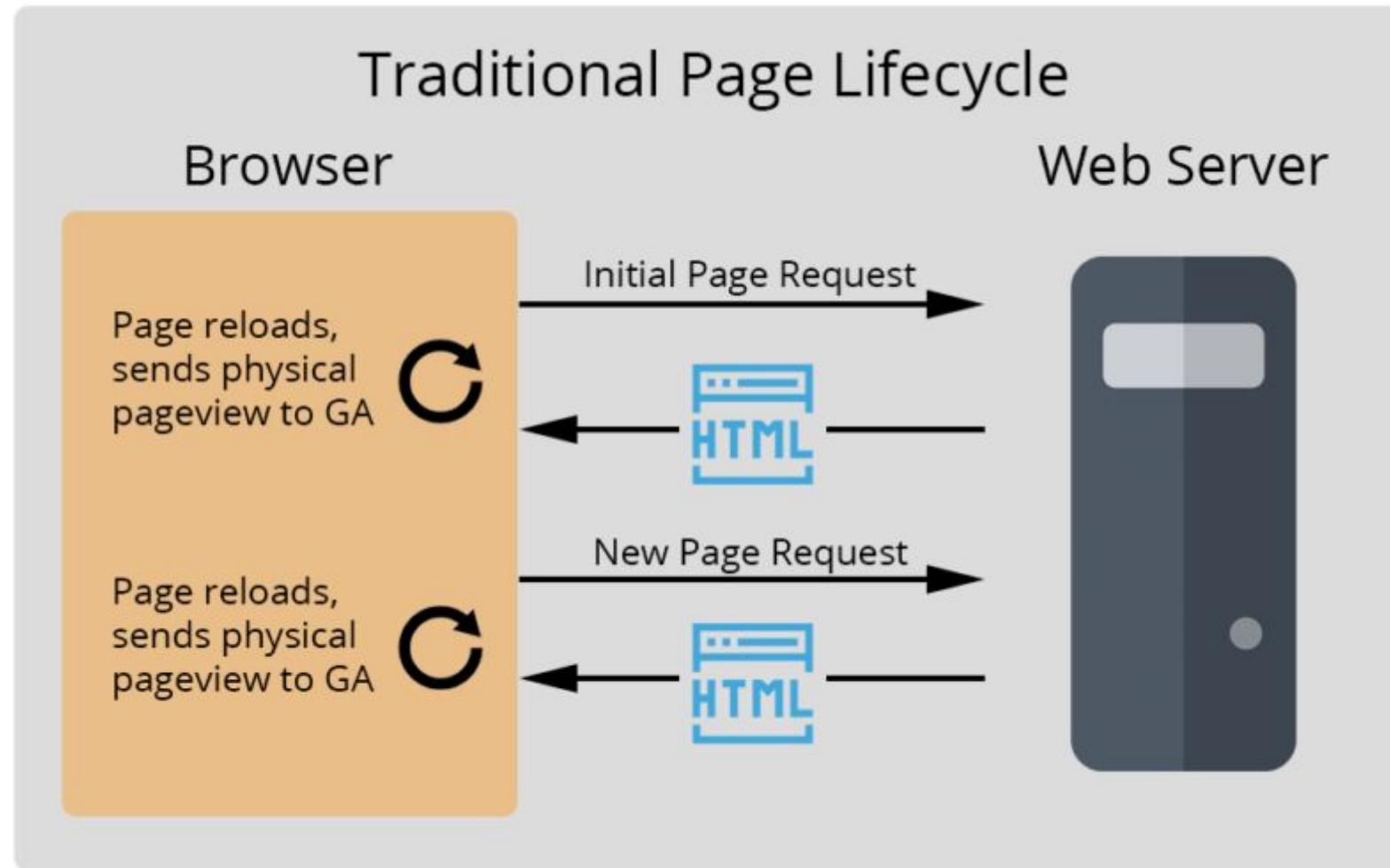
source: <http://krakoukas.com/mieux-avant/wayback-machine/>

# JavaScript

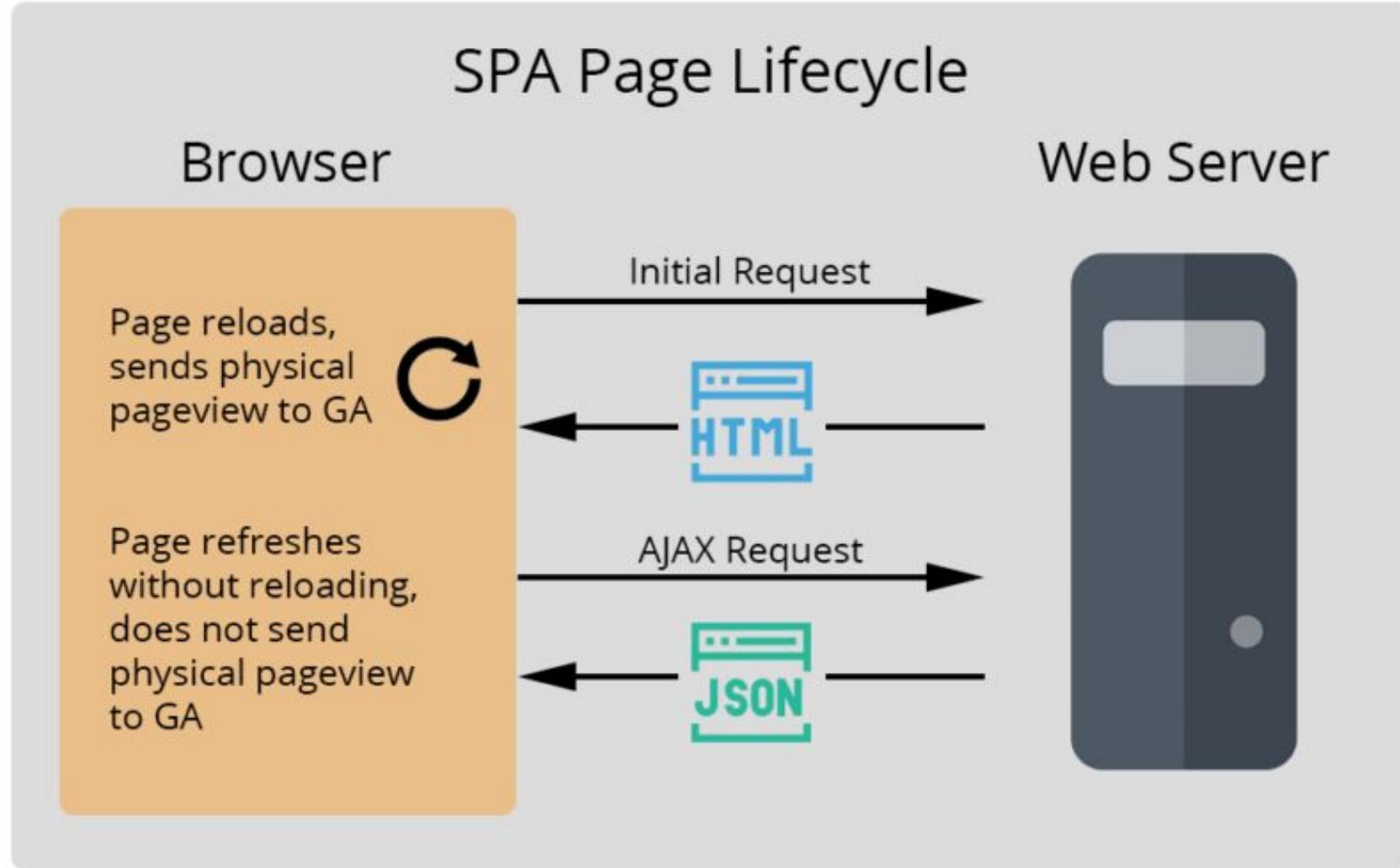


```
$function() {  
  
    function saveState() {  
        if (this.options.saveState == true) {  
            if (this.state == "normal") {  
                $.ajax({  
                    url : "bla"  
                });  
            } else {  
                $.ajax({  
                    url : "not bla"  
                });  
            }  
        }  
    }  
}(jQuery);
```

# Multi Pages Applications

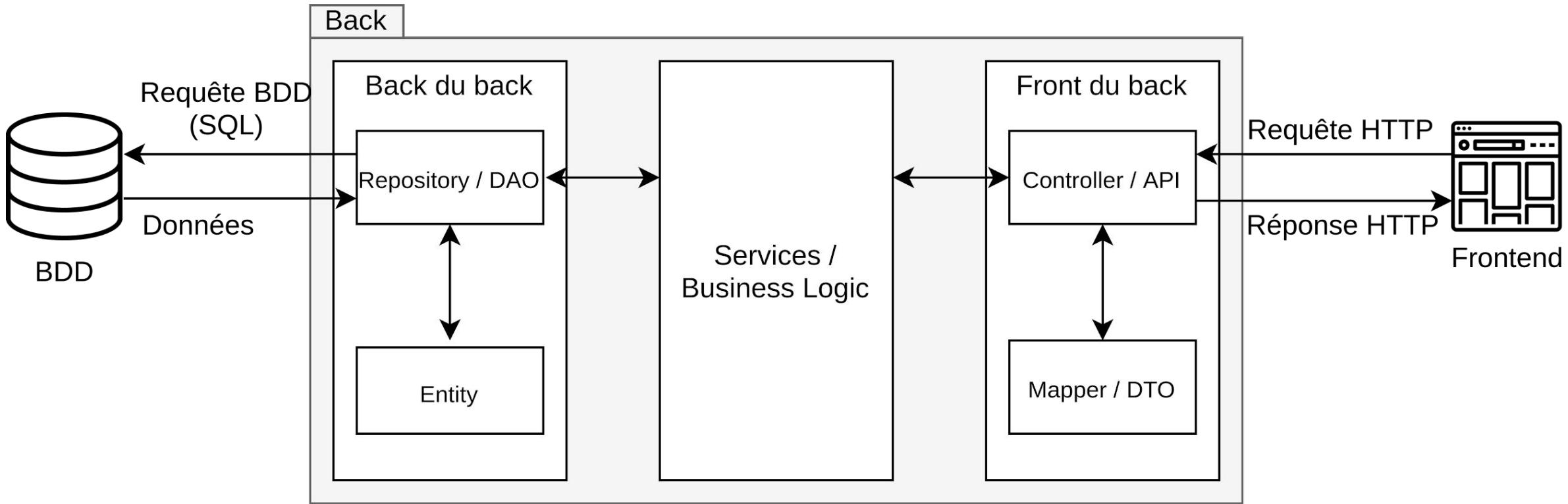


# Single Page Application (SPA)

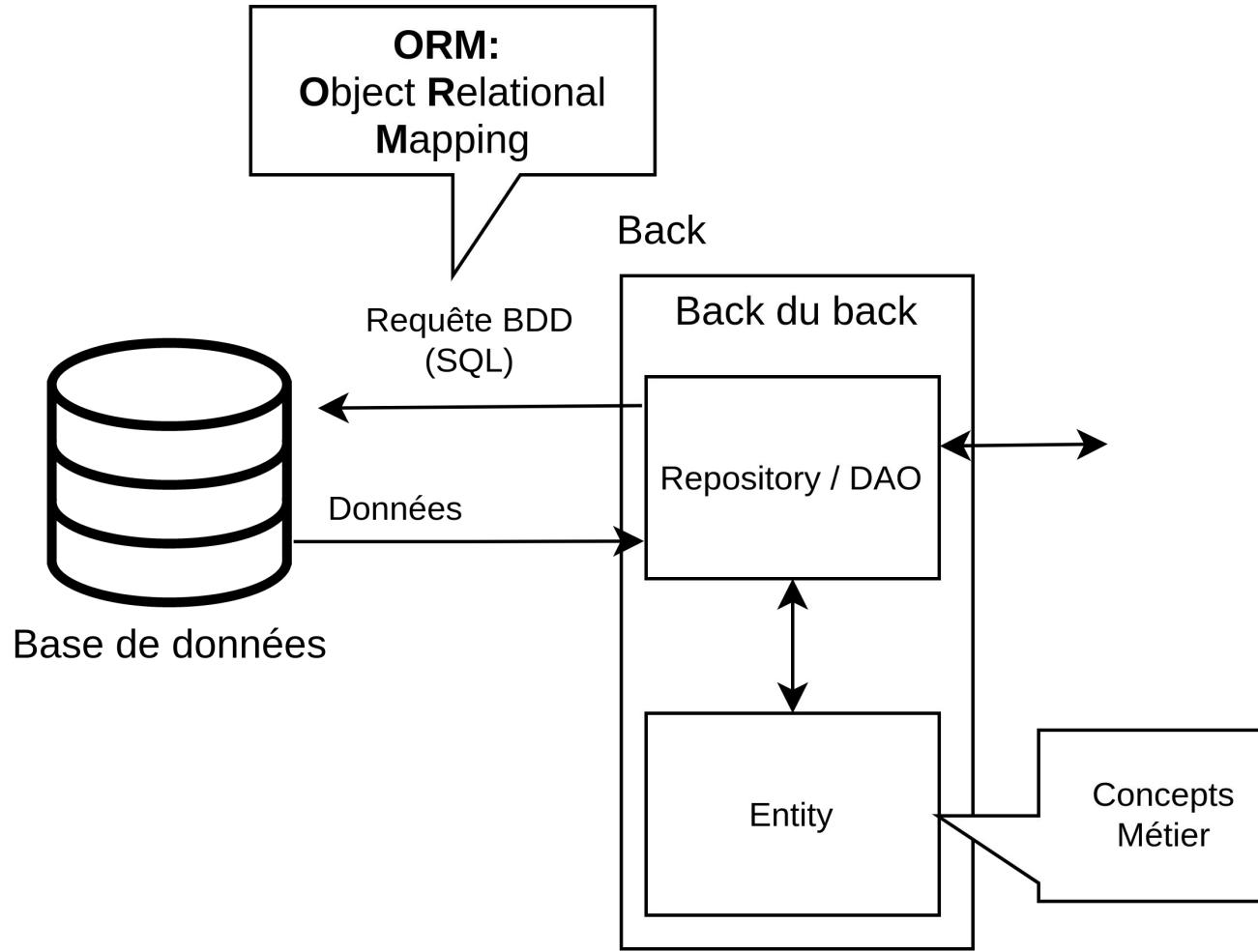


# Le backend en détail

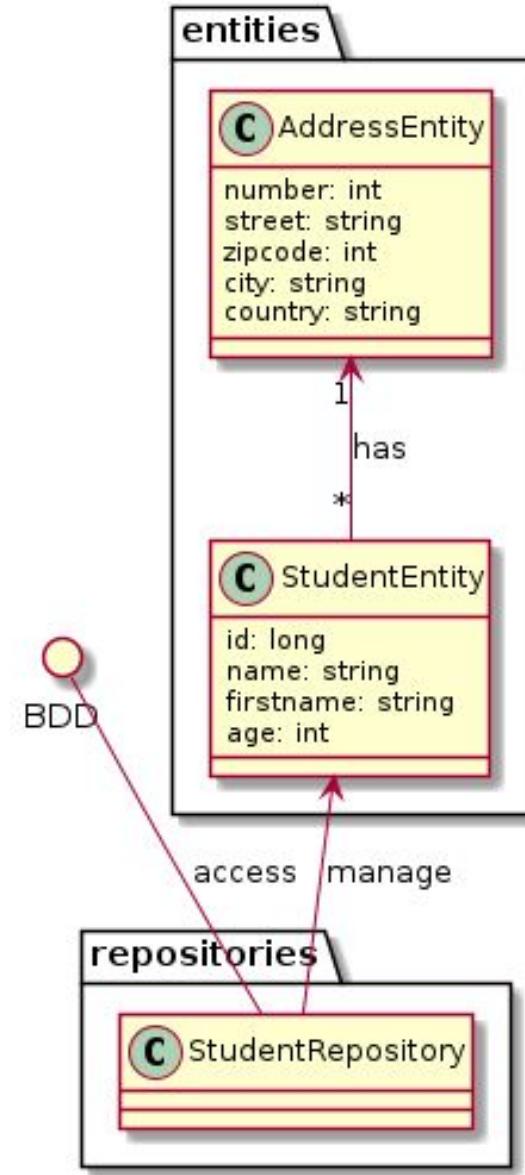
# Aperçu du backend



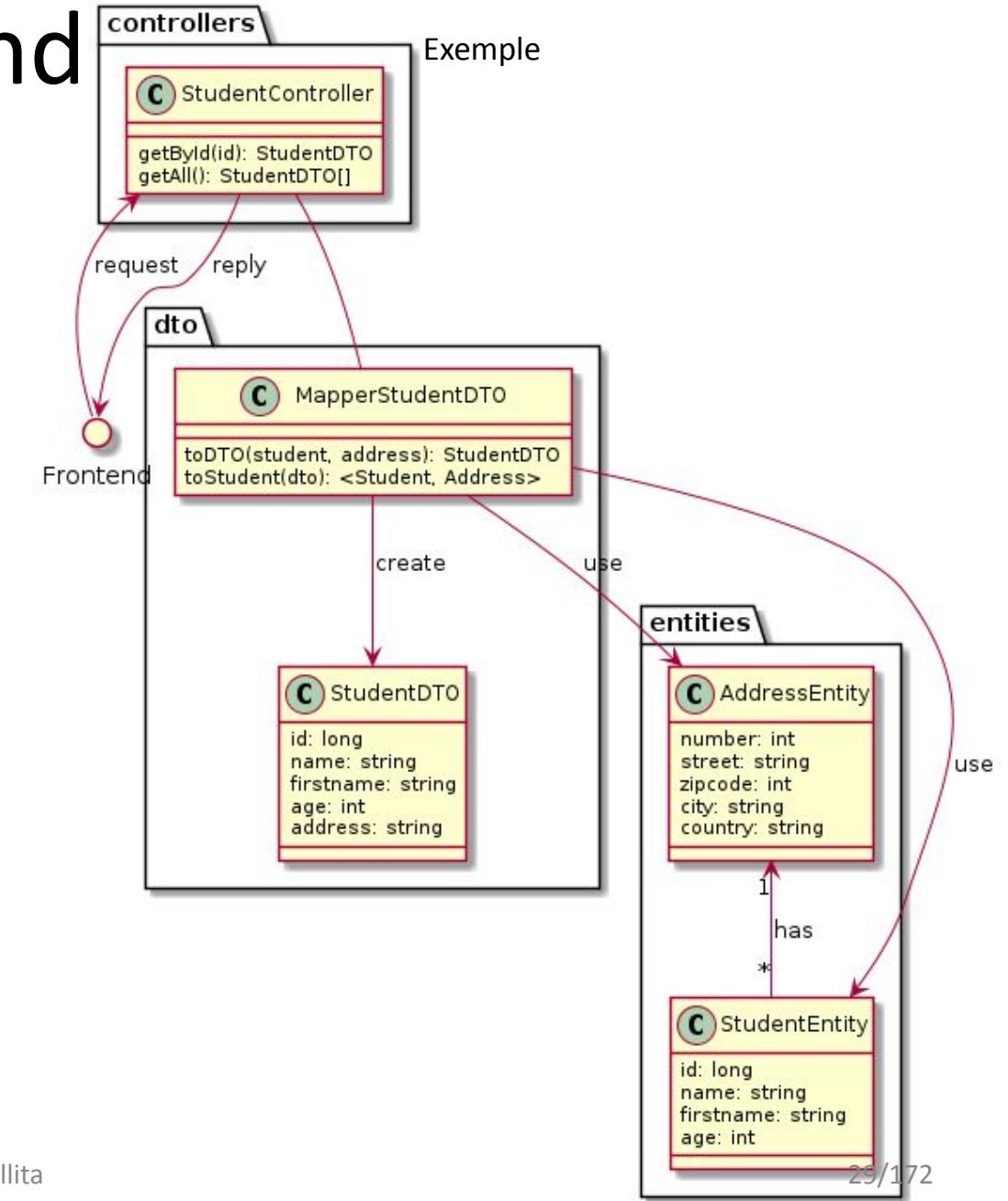
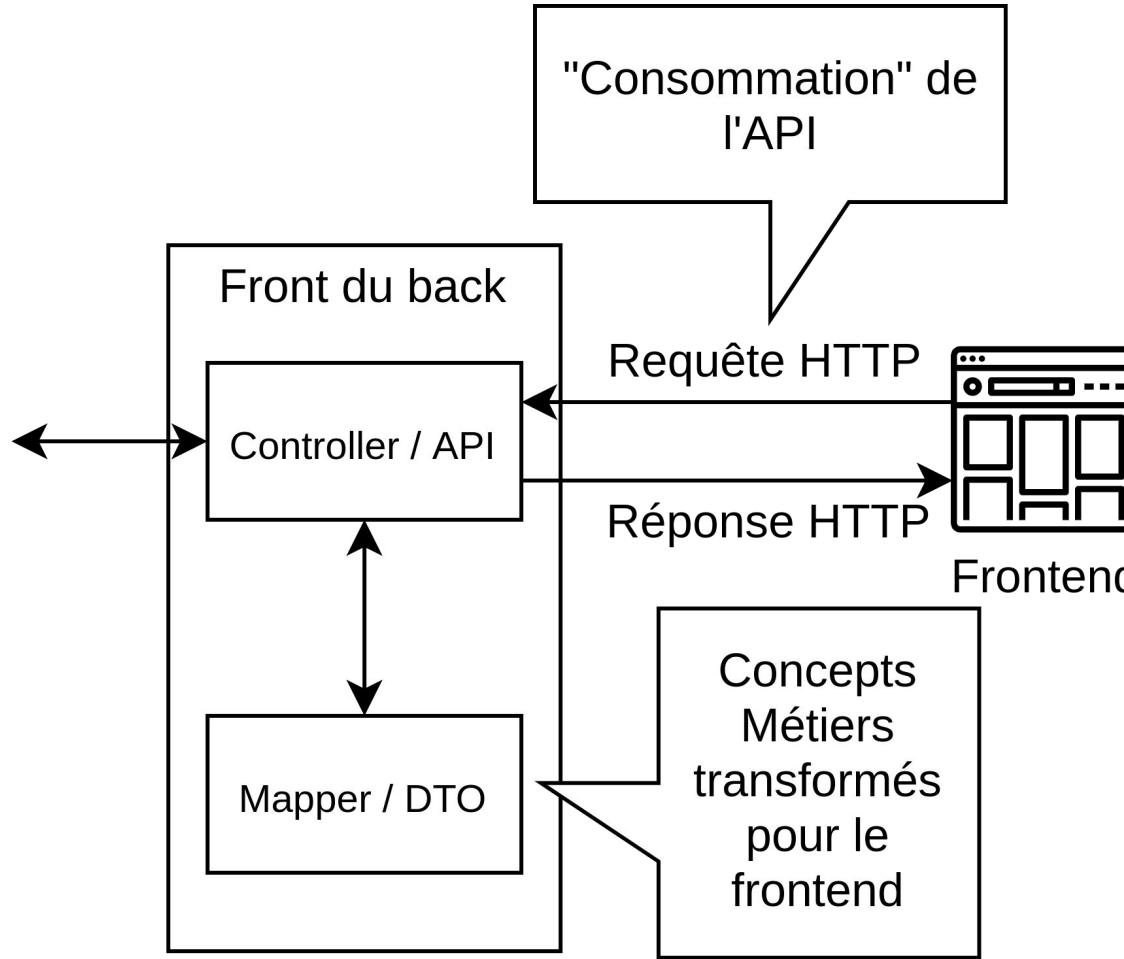
# Aperçu du back du backend



Exemple

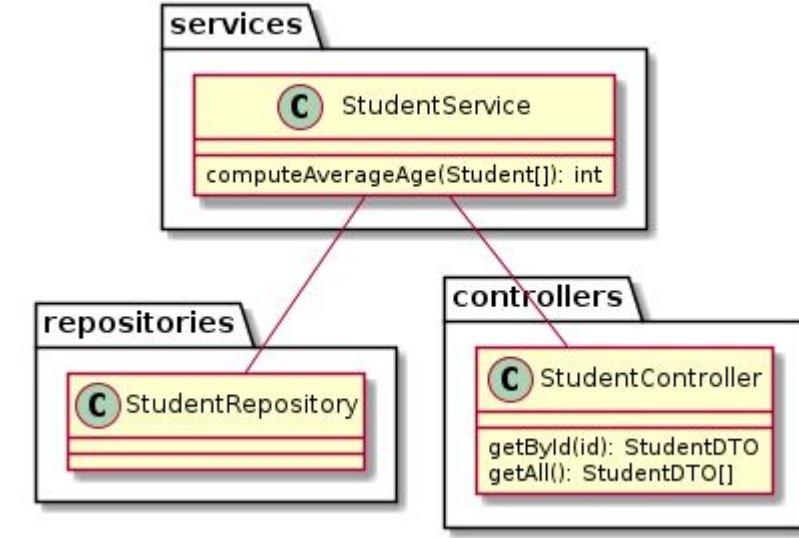
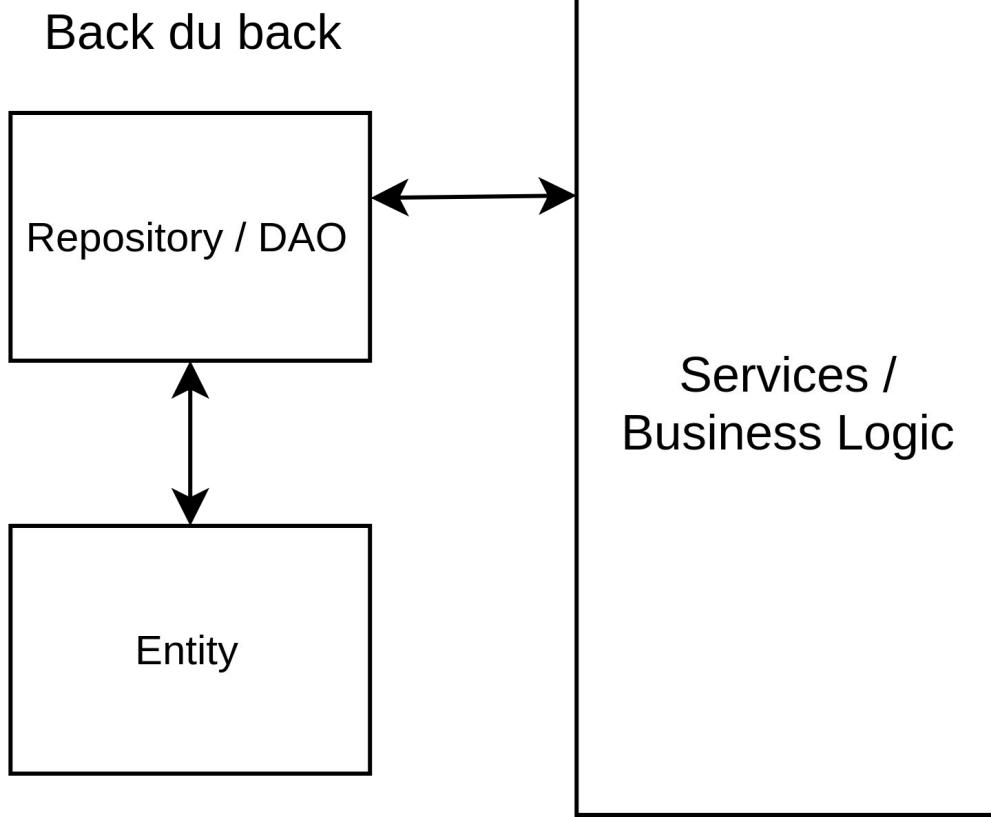


# Aperçu du front du backend



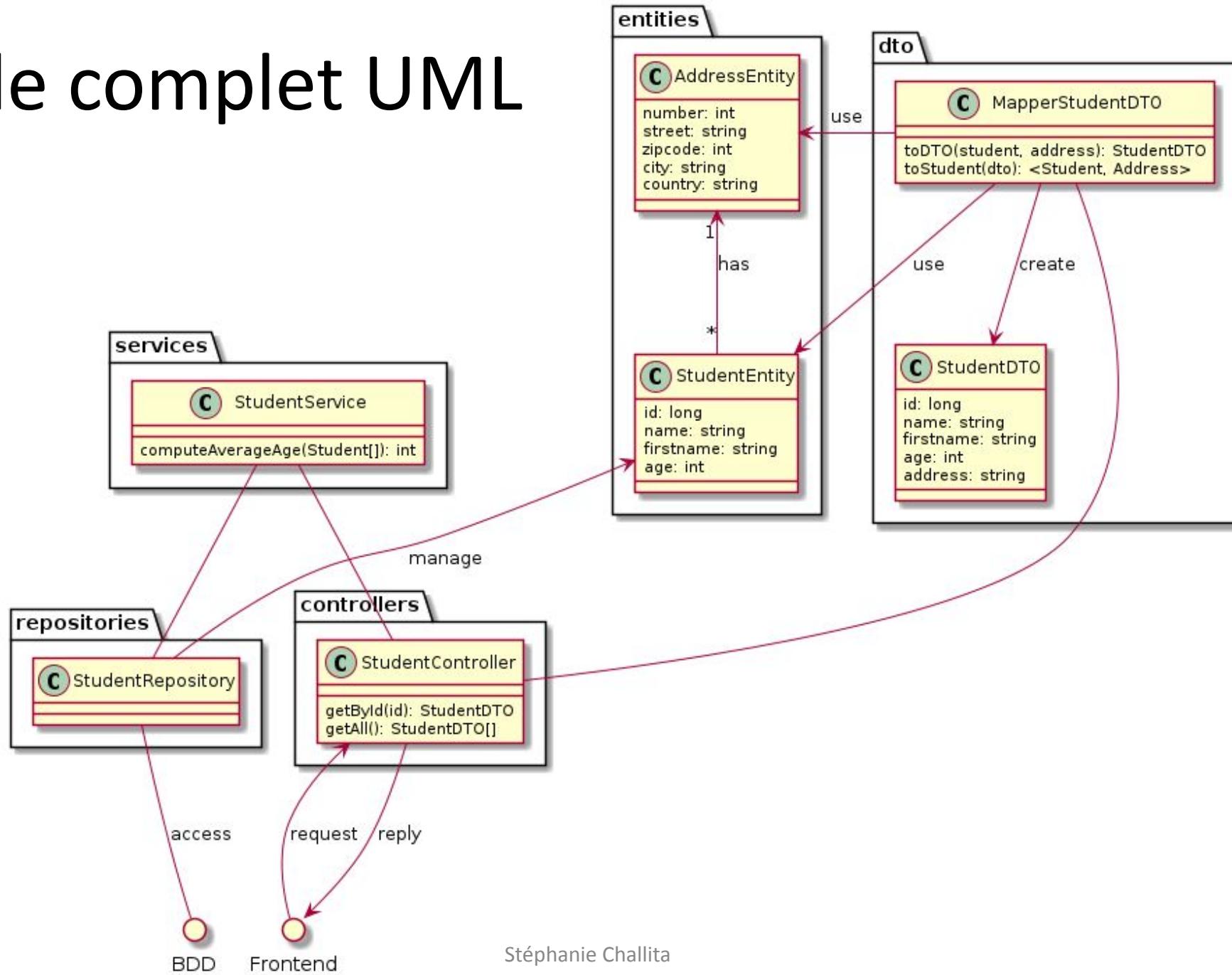
# Aperçu des services du backend

Exemple



La couche service fait le pont entre les contrôleurs et les données. Elle implémente aussi toute la logique métier pour traiter les données.

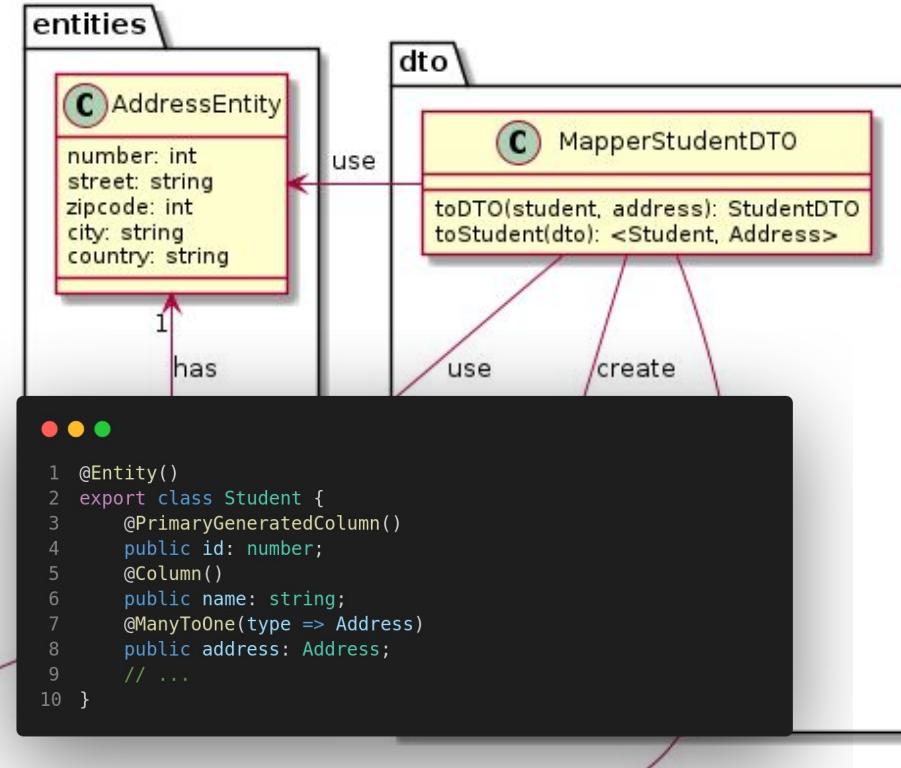
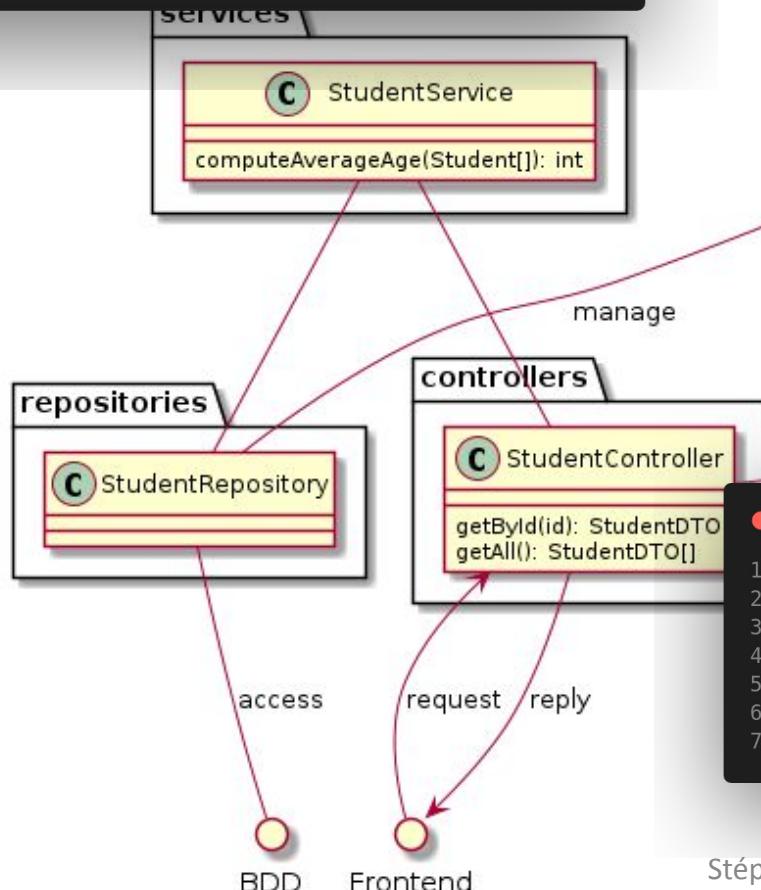
# Exemple complet UML



# Code snippets



```
1 @Injectable()
2 export class StudentsService {
3     constructor(@InjectRepository(Student)
4                 private repository: Repository<Student>) {}
5     public computeAvgAge(students: Student[]): number {}
6 }
```



```
1 @Entity()
2 export class Student {
3     @PrimaryGeneratedColumn()
4     public id: number;
5     @Column()
6     public name: string;
7     @ManyToOne(type => Address)
8     public address: Address;
9     // ...
10 }
```

**Code snippets**

```
1 @Controller('students')
2 export class StudentsController {
3     @Get(':id')
4     public getById(id: number): StudentDTO { /*...*/ }
5     @Post()
6     public createStudent(data: any): StudentDTO { /*...*/ }
7 }
```

# Controller : rôles

- Définition des endpoints
- Traitement des requêtes
- Délégation de la logique à la couche services
- Envoi des réponses avec les données ou des erreurs

# Controller : exemple

```
1 @Controller('students')
2 export class StudentsController {
3
4     constructor(
5         private service: StudentsService
6     ) {}
7
8     @Get(':id')
9     public getById(@Param() param): User {
10         const user = this.service.getById(param.id);
11         if (user === undefined) {
12             throw new HttpException(
13                 `Student ${param.id} not found!`,
14                 HttpStatus.NOT_FOUND,
15             );
16         } else {
17             return user;
18         }
19     }
20 }
```

- GET /students/1
  - 200 {  
“id”: 1,  
“name”: “John”  
}
  - 404 NOT FOUND  
Si l’étudiant.id == 1  
n’existe pas

# Services

- Implémente la logique métier
- Fait le pont entre les controllers et les repositories

# Services : exemple

```
1  @Injectable()  
2  export class StudentsService {  
3      constructor(  
4          @InjectRepository(User)  
5          private repository: Repository<User>  
6      ) {}  
7  
8      async getById(idToFind: long): Promise<User> {  
9          return await this.repository  
10             .findOne({  
11                 id: Equal(idToFind)  
12             } );  
13     }  
14 }  
15 }
```

- Utilise le repository pour interroger la BDD

# Repositories et Entities

- Les repositories communiquent avec la BDD
- Les Entities sont les objets que l'on souhaite persister en base
- Plus de détails dans la suite... sur la partie ORM

# Un peu de technique, TS, NestJS

# Variables et Constantes

```
1 // déclaration d'une constante de type number
2 const myVar: number = 3;
3 // déclaration d'une variable de type string
4 let secondVar: string;
5 // affectation de valeur
6 secondVar = 'ThisIsAString';
7 // string template
8 const templateString: string = `template ${secondVar}`;
9 // inférence du type à partir de la valeur d'initialisation
10 let booleanVar = false;
11 // erreur
12 booleanVar = 'toto';
```

# Arrays

```
1 // tableau
2 let list: number[] = [1, 2, 3];
3 // idem
4 let array: Array<number> = list;
5 // déclaration d'un tableau vide
6 const constList = [];
7 // ajout d'un élément
8 constList.push(1);
9 // affiche le premier element
10 console.log(constList[0]);
11 // retire 1 élément à partir de l'indice 0
12 constList.splice(0, 1);
```

# Classes

```
1  export class MyClass {  
2      // attributs  
3      public attribute1: number;  
4      private attribute2: string;  
5      // constructeur  
6      constructor(  
7          // paramètres  
8          attribute1: number,  
9          attribute2: string  
10     ) {  
11         // affectations  
12         this.attribute1 = attribute1;  
13         this.attribute2 = attribute2;  
14     }  
15     // ...  
16 }
```

# Classes

```
1 export class MyClass {  
2     public attribute1: number;  
3     private attribute2: string;  
4     constructor(  
5         attribute1: number,  
6         attribute2: string  
7     ) {  
8         this.attribute1 = attribute1;  
9         this.attribute2 = attribute2;  
10    }  
11    // ...  
12 }
```

```
1 export class MyClass {  
2     constructor(  
3         public attribute1: number,  
4         private attribute2: string  
5     ) { }  
6     // ...  
7 }
```

# Method

```
1  export class MyClass {  
2      constructor(  
3          public attribute1: number,  
4          private attribute2: string  
5      ) {}  
6      public myMethod(parameter: number): string {  
7          console.log(parameter + this.attribute1);  
8          if (this.attribute1 === parameter) {  
9              return this.attribute2 + "";  
10         } else {  
11             return ""  
12         }  
13     }  
14 }
```

# Asynchronisme

```
1 async asyncMethod(): Promise<number> {
2     // les access a la db sont
3     // souvent asynchrones
4     return this.db.getRndId();
5 }
```

Mot-clé **async**  
pour déclarer  
une méthode  
asynchrone  
Renvoie  
toujours une  
Promesse  
Access db  
souvent en  
asynchrone

# Asynchronisme : then

```
1 method(): number {  
2     let value;  
3     this.asyncMethod()  
4         .then(asyncValue => value = asyncValue)  
5         .catch(error => console.log('error'));  
6     return value;  
7 }
```

- **then()** : que faire avec la valeur lorsque la promesse est résolue ?
- **catch()** : que faire en cas d'erreur ?

# Asynchronisme : await

```
1  async method2(): Promise<number> {  
2      let value = await this.asyncMethod();  
3      return value;  
4  }
```

# Object Relational Mapping (ORM)

# De la base de données aux objets (ORM)

- Introduction à la problématique et au contenu
- De la table à la classe
- Traduction des associations
- Objet dépendant
- Traduction de l'héritage
- Navigation entre les objets
- Coup d'oeil sur TypeORM

# Problématique de l'objet aux bases données

Définition de la structure de la base de données



Modélisation objets

But : mapping automatique, voire génération complète du schéma de la base de données à partir de la modélisation de données

# Contexte

- Back du backend
- Faire le lien entre la couche service et la base de données
- On prend le cas d'une base de données relationnelle

# Pourquoi persister les objets dans une BDD?

## ■ Persistence ?

- Sauvegarde des données en cas de crash
- Cohérence des données partagées

## ■ BDD ?

- Performance
- Stockage de masse

# Pourquoi une BDD relationnelle ?

- Position dominante
- Bas coût
- Facile à mettre en place et efficace pour les recherches complexes
- Grande adaptation, notamment avec les vues
- Spécification de contraintes d'intégrité naturelles
- Théorie solide et norme
- Grande présence de compétences sur le marché

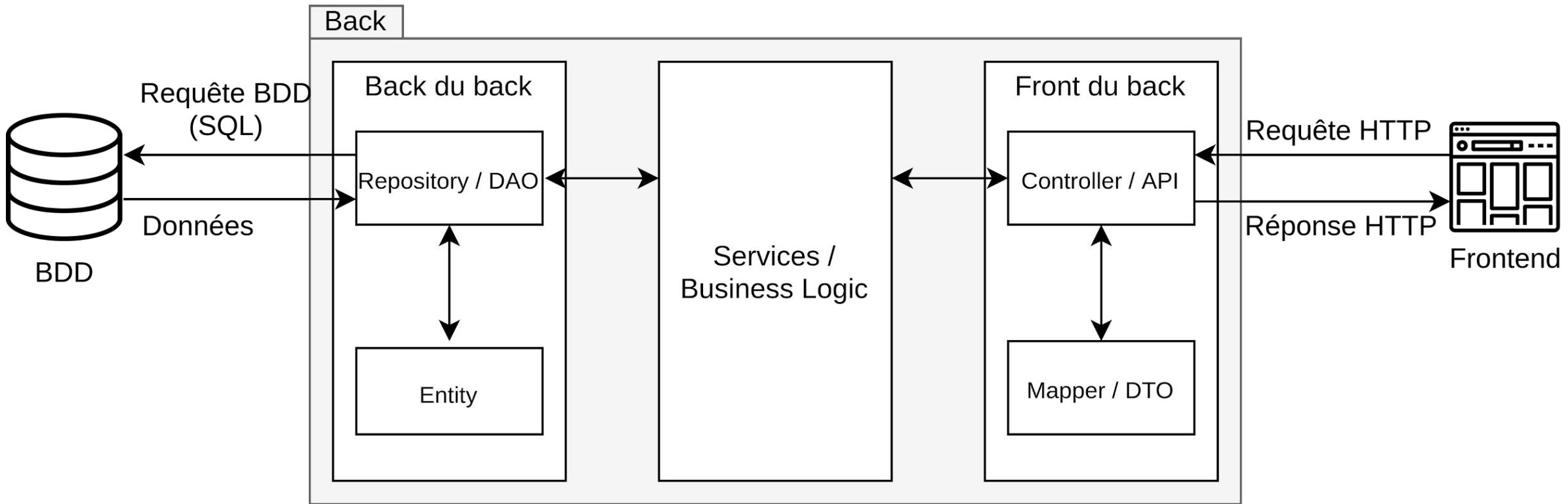
# POO donc pourquoi pas un SGBD objet ?

- BDD relationnelles ont une position dominante
- SGBD objet passe moins bien à l'échelle que les SGBD relationnels
- Moins de souplesse
- Pas ou peu de normalisation :
  - Beaucoup de solutions propriétaires
- **Peu de compétence sur le marché (du fait des autres raisons)**

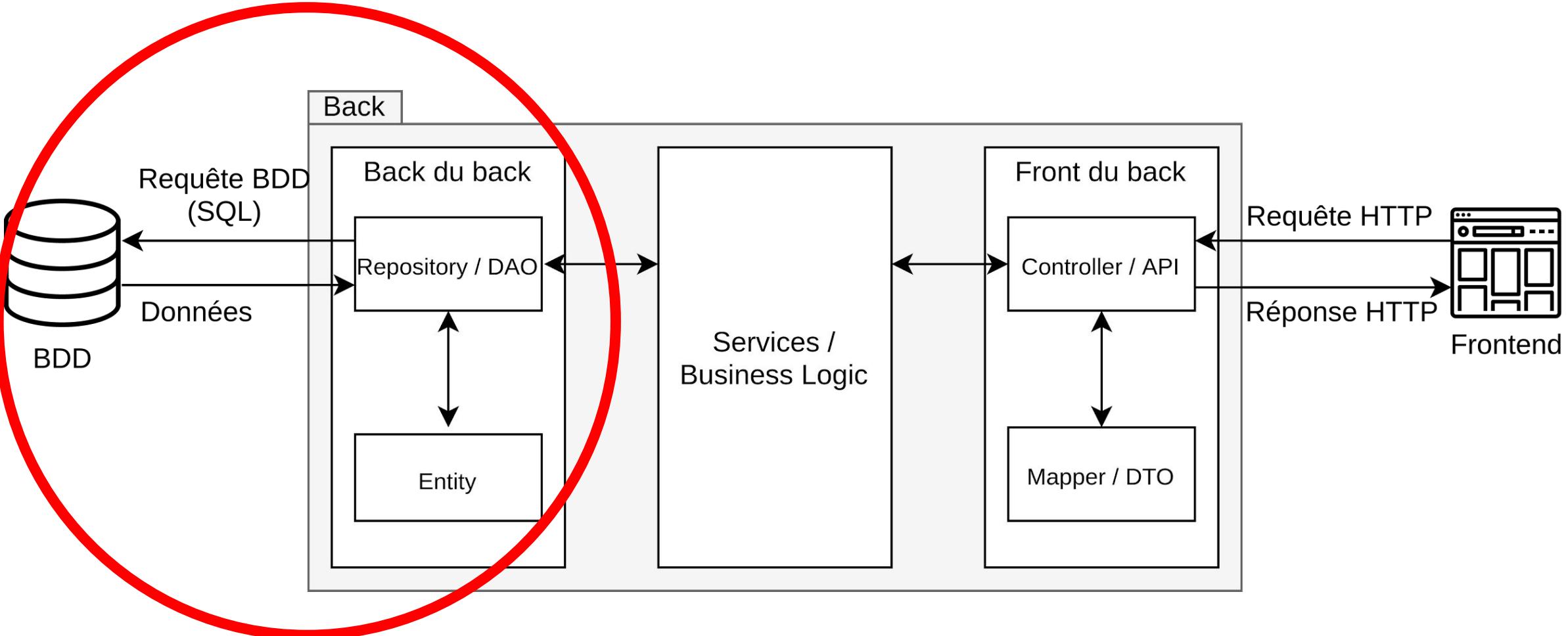
# Objet -> BDD, quelles sont les difficultés ?

- Deux paradigmes différents : relationnel vs objet
  - Concepts différents
  - Relationnel moins riche :
    - Pas d'héritage, de références, de collections, etc.

# Rappel : Où se situe cette problématique ?



# Rappel : Où se situe cette problématique ?



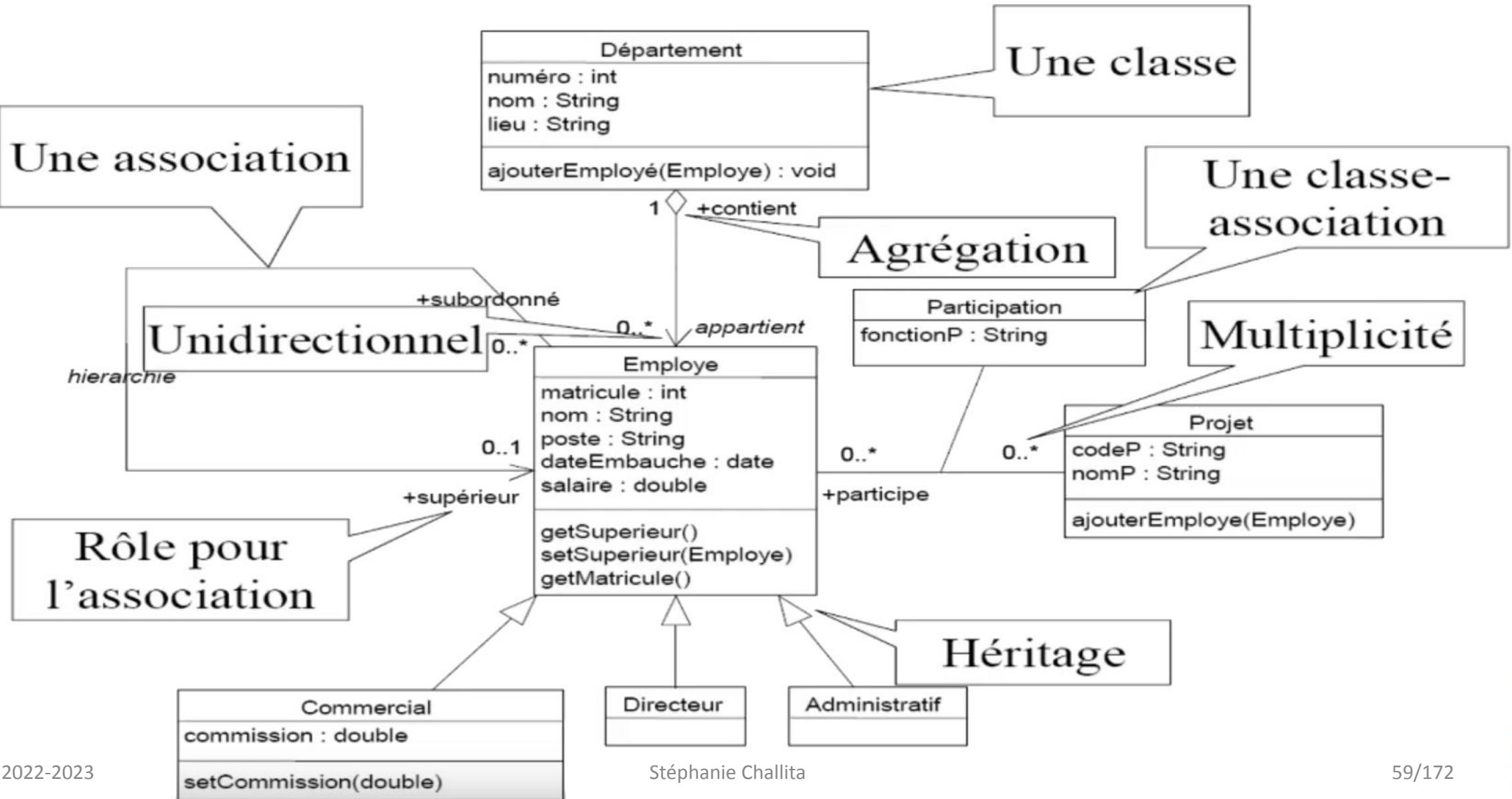
# Contenu de cette partie du cours

- Quelles sont les problématiques pour exprimer cette correspondance ?
- TypeORM : une implémentation ORM pour TypeScript

# Contexte

- Au moment de la conception objet, la BDD peut exister ou être inexistante
- Dans ce cours: on va (souvent) partir du principe qu'il n'y a pas de BDD
- Les langages d'ORM peuvent s'adapter à une BDD existante assez facilement

# Rappel UML



# Problèmes : R <-> Object

- Identité des objets
  - Traductions des associations
  - Traductions de l'héritage
  - Navigation entre les objets
- 
- But : Enregistrer les objets dans une BDD R
  - Structure des objets complexes ≈ graphe (arbre)
  - Racine = Objet, Fils = attributs
  - Besoin d'aplatir le graphe pour l'insérer dans la BDD R

# De la classe à la table

- Dans les cas les plus simples : 1 class == 1 table
- Chaque instance de class est une ligne dans la table
- Exemple : Class Département == Table Département



```
● ● ●  
1 {  
2     numéro: 2,  
3     nom: "Département Informatique",  
4     lieu: "Campus Beaulieu",  
5 }
```

DÉPARTEMENT(numéro, nom, lieu)

numéro	nom	lieu
2	Département Informatique	Campus Beaulieu
...	...	...

# En SQL



```
1 CREATE TABLE Département {  
2     numéro SMALLINT  
3     CONSTRAINT pk_dept PRIMARY KEY,  
4     nom VARCHAR(30),  
5     lieu VARCHAR(30)  
6 }
```

# Identification des objets

- Problème: un objet est identifiable par son emplacement en mémoire
- 2 objets **distincts** peuvent avoir des valeurs strictement identiques
- Problème: dans une table, seules les valeurs des colonnes peuvent identifier une ligne
- Si 2 lignes ont les mêmes valeurs, il est impossible de les différencier
- Dans un schéma relationnel on a besoin d'une clé primaire pour toute table

# Propriété de “clé primaire” pour les objets

- Pour effectuer la correspondance objet - table :  
    → Besoin d'ajouter un identificateur à un objet
- Cet identificateur fera office de clé primaire dans la BDD

# Éviter les identificateurs significatifs

- Idée : utiliser une valeur métier pour l'identification
- Préférer les identificateurs artificiels
- Surtout quand l'identificateur est un composite

# Problème de duplications

- On veut éviter toute duplication :  
1 objet == 1 ligne et 1 ligne == 1 objet
- Risque de perte de données, ou de mauvaise synchronisation

# Exemple de duplications

- Un objet P1, de type produit, créé lors de la récupération d'une facture de la BDD
- Depuis une autre facture, on peut retrouver le même produit
- Une erreur serait de créer un second objet P2, qui représente le même produit, en mémoire

# Éviter les duplications

- Mise en place d'un cache, qui garde en mémoire tous les objets, et conserve leur identité en base (clé primaire)
- Lors de l'interrogation de la BDD, le cache intervient
- Soit le cache fournit l'objet, soit il interroge la BDD

# Objets embarqués

- Le modèle objet peut avoir une granularité plus fine que le modèle relationnel
- On peut alors avoir certaines classes qui n'ont pas de tables dédiées, mais sont insérées dans une table d'une autre classe
- Ces objets sont appelés “embarqués” et ne nécessitent pas d'identificateur (clé primaire)

# Objets embarqués : exemple

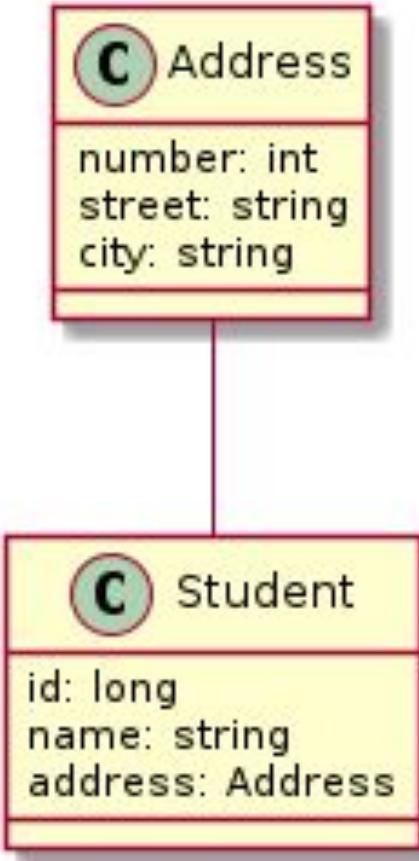


Table Student

<b>id</b>	<b>name</b>	<b>number</b>	<b>street</b>	<b>city</b>
1	John	23	Esir Street	Rennes

# Exemples de code

## TypeScript (TypeORM)

```
1  @Entity()  
2  export class Student {  
3      @PrimaryGeneratedColumn()  
4      public id: number;  
5      @Column()  
6      public name: string;  
7      @Column(() => Address)  
8      public address: Address;  
9      // ...  
10 }
```

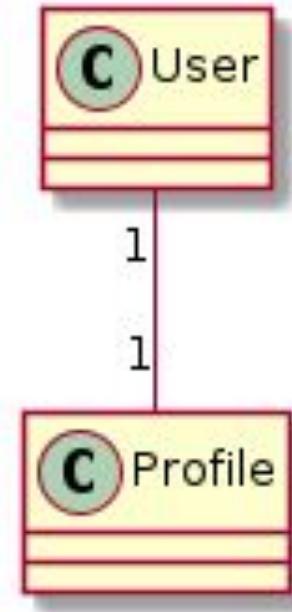
## Java (Hibernate)

```
@Entity  
public class Student {  
    @Id  
    @GeneratedValue  
    private int id;  
    private String name;  
    @Embedded  
    private Address address;  
}
```

# Traduction des associations

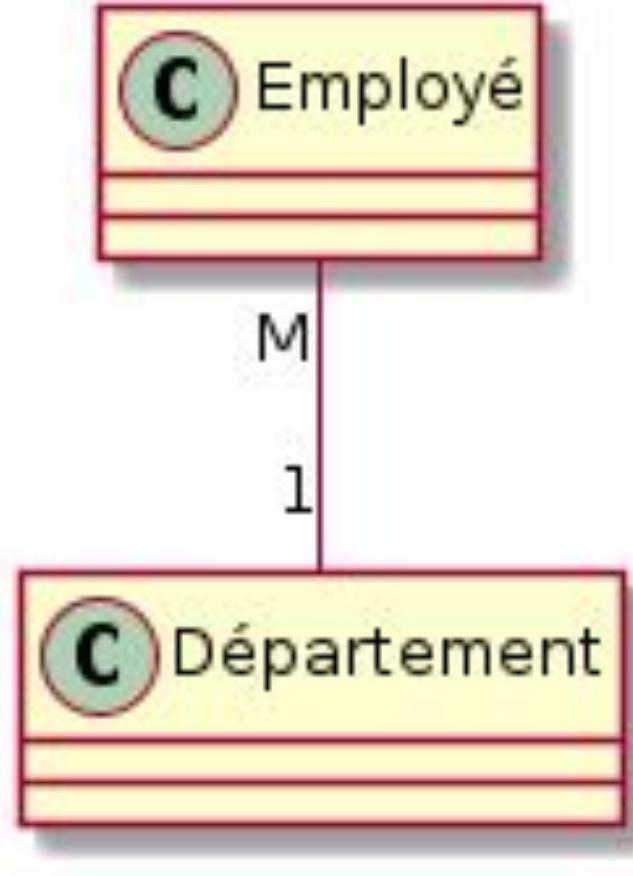
- 3 associations à distinguer :
  2. Un attribut d'instance représentant “l'autre” objet (1:1, M:1)
  3. Un attribut d'instance de type collection représentant tous les autres objets (1:M, M:N)
  4. Une classe association (M:N)

# Exemple d'association 1 (1:1)



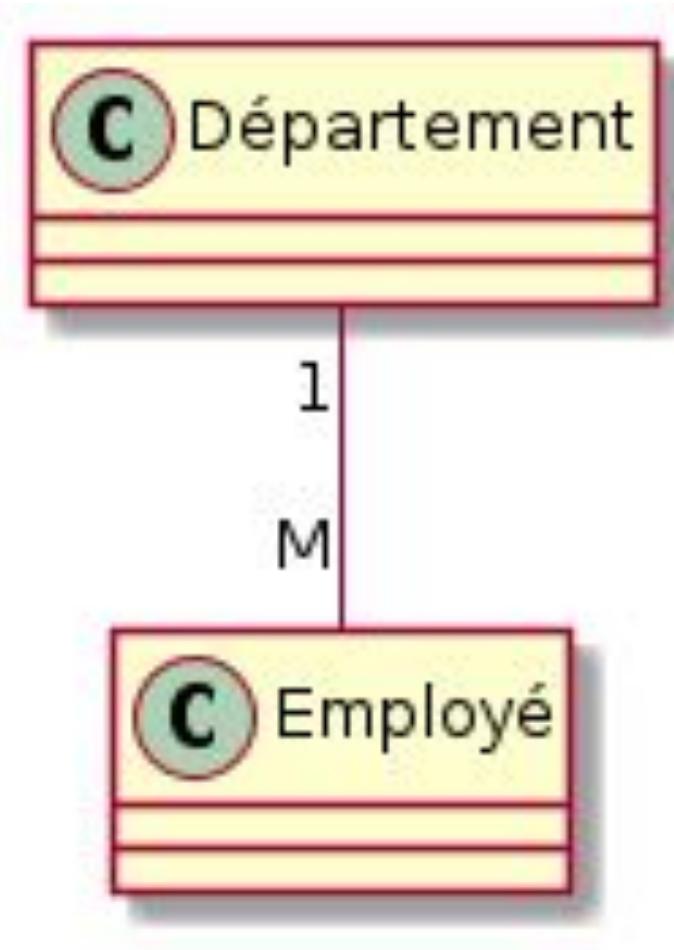
```
1 export class User {  
2     profile: Profile  
3 }  
4 export class Profile {  
5     user: User  
6 }
```

# Exemple d'association 2 (M:1)



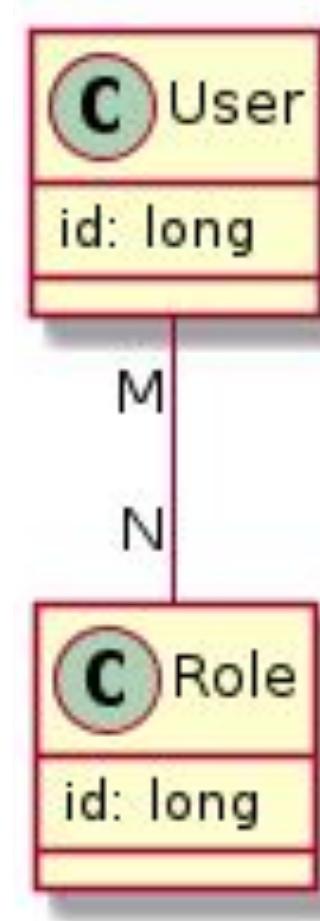
```
1 export class Département {  
2     employés: Employé[];  
3 }  
4 export class Employé {  
5     département: Département;  
6 }
```

# Exemple d'association 2 (1:M)



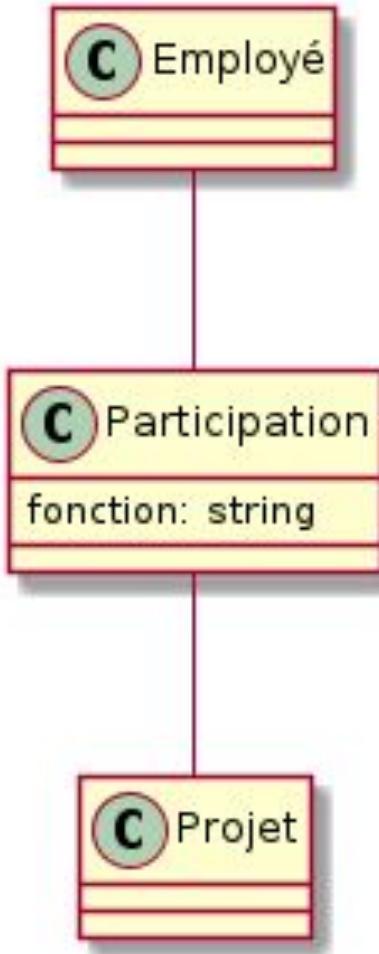
```
1 export class Département {  
2     employés: Employé[];  
3 }  
4 export class Employé {  
5     département: Département;  
6 }
```

# Exemple d'association 3 (M:N)



```
export class User {  
    roles: Role[];  
}  
export class Role {  
    users: User[];  
}
```

# Exemple d'association 3 (M:N) avec propriétés



```
export class Employé {
    public participations: Participation[];
}

export class Participation {
    public employé: Employé;
    public projet: Projet;
    public fonction: string;
}

export class Projet {
    public participations: Participation[];
}
```

# Et dans le monde relationnel ?

- Une ou plusieurs clé(s) étrangère(s) (foreign keys)
- Table d'associations

# Exemple d'association 1 (1:1)

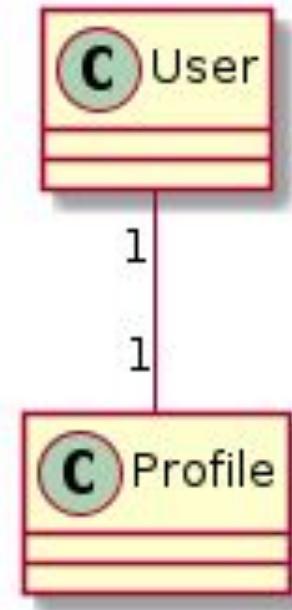


Table User

<b>id</b>	<b>idProfile</b>	...
1	3	...
2	42	...

Table Profile

<b>id</b>	...
3	...
42	...

# Exemple d'association 2 (1:M)

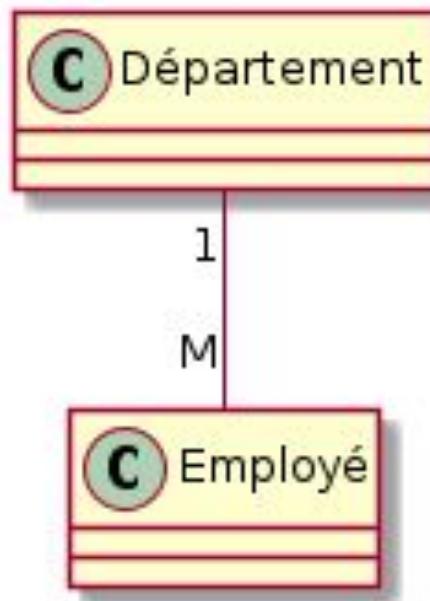


Table Département

<b>id</b>	<b>Nom</b>	...
1	INFO	...
2	ADMIN	...

Table Employé

<b>id</b>	<b>idDept</b>	...
3	1	...
42	1	...
23	2	...

# Exemple d'association 2 (M:N)

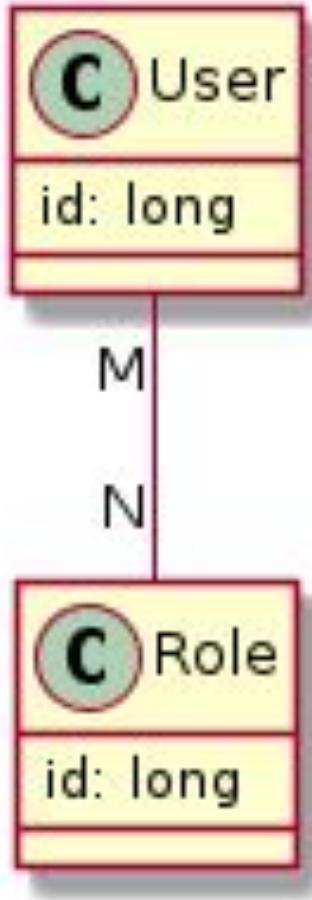


Table User

<b>id</b>	...
1	...
2	...

Table User - Rôle

<b>idUser</b>	<b>idRôle</b>
1	1
2	1
2	2

Table Rôle

<b>id</b>	...
1	...
2	...

# Exemple d'association 3 (M:N) avec propriétés

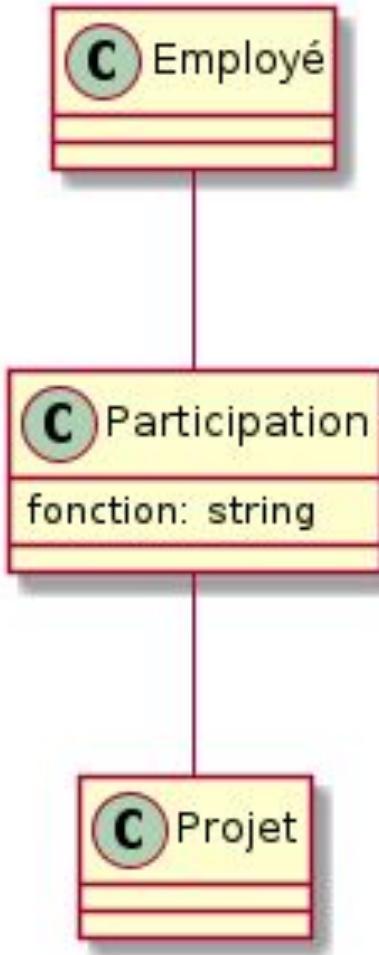


Table Employé

<b>id</b>	...
1	...
2	...

Table Participation

<b>idE</b>	<b>idP</b>	<b>fonction</b>
1	1	PO
2	1	Dév
2	2	Scrum Master

Table Projet

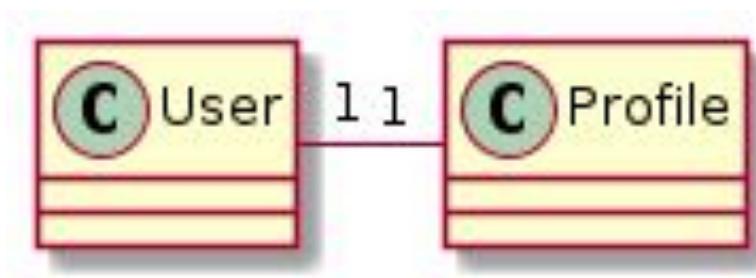
<b>id</b>	...
1	...
2	...

# La navigation dans les objets

- En objet, une association peut-être bidirectionnelle ou unidirectionnelle
- Exemple :

```
1 export class Département {  
2  
3 }  
4 export class Employé {  
5     département: Département;  
6 }
```

# Différents types de bidirectionnalités

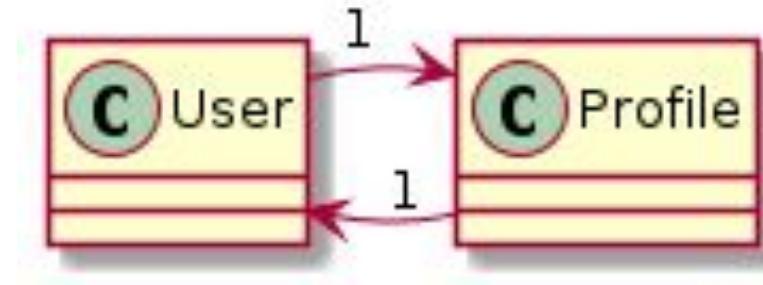


User

<b>id</b>	<b>idProfile</b>	...
1	3	...
2	42	...

Profile

<b>id</b>	...
3	...
42	...



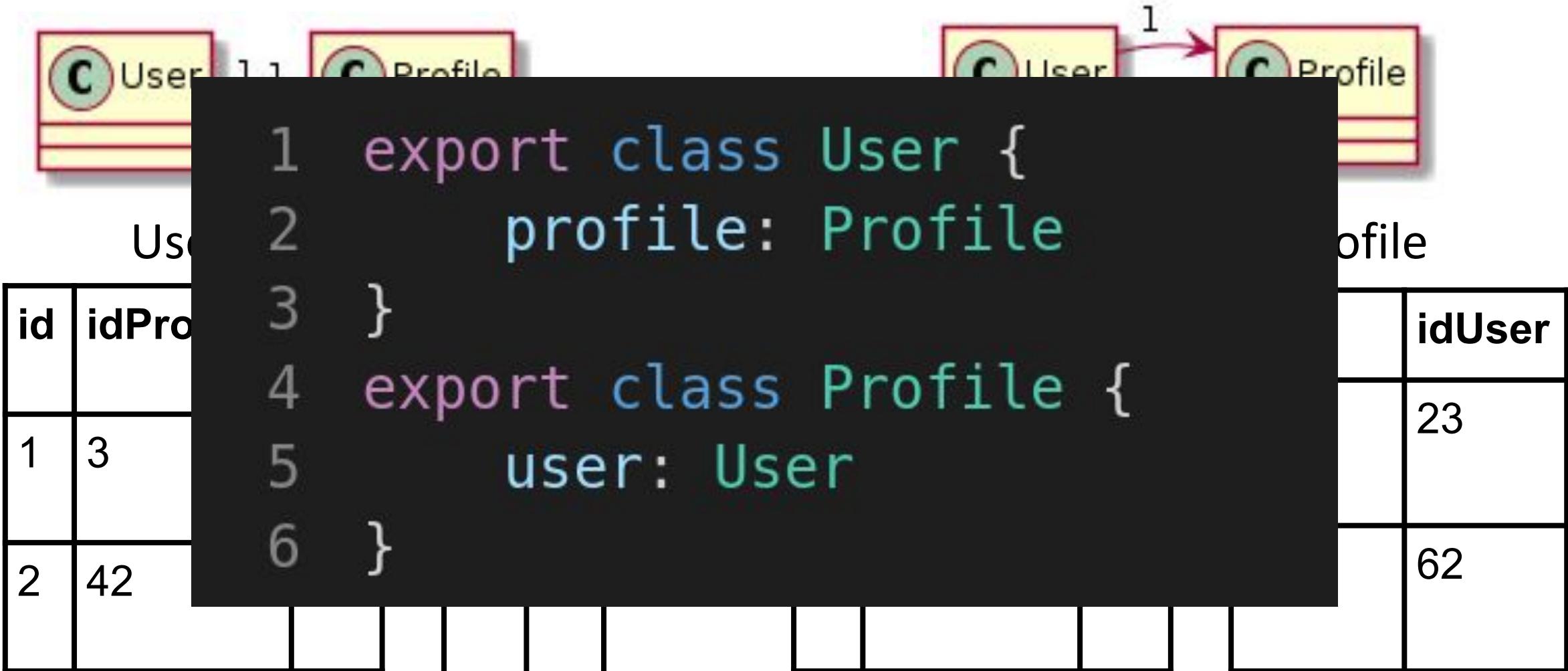
User

<b>id</b>	<b>idProfile</b>	...
1	3	...
2	42	...

Profile

<b>id</b>	<b>idUser</b>
3	23
42	62

# Différents types de bidirectionnalités



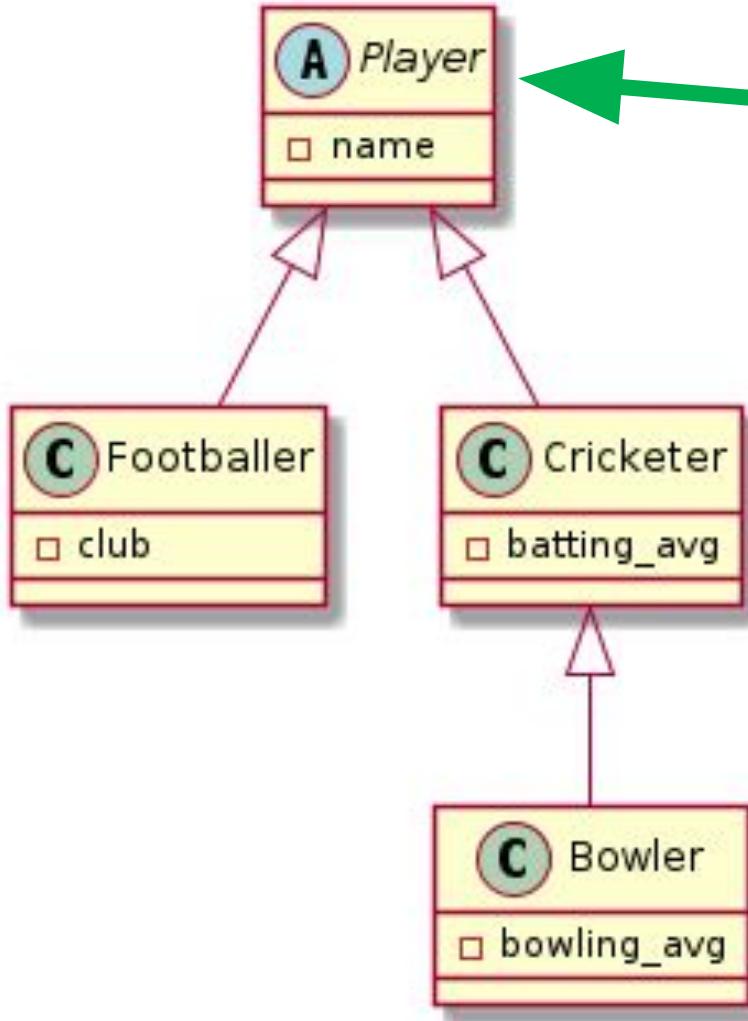
# Objet dépendant

- Le cycle de vie d'un objet dépendant dépend du cycle de vie d'un objet propriétaire
- Que faire lorsque je supprime l'objet propriétaire ? Suppression en cascade ?

# Objet dépendant : exemple

- Une facture est composée de lignes  
Suppression de facture → suppression des lignes
- Une ligne référence une produit  
Suppression de la ligne → on garde le produit intact
- Cas typique où il faut donner ces informations au framework

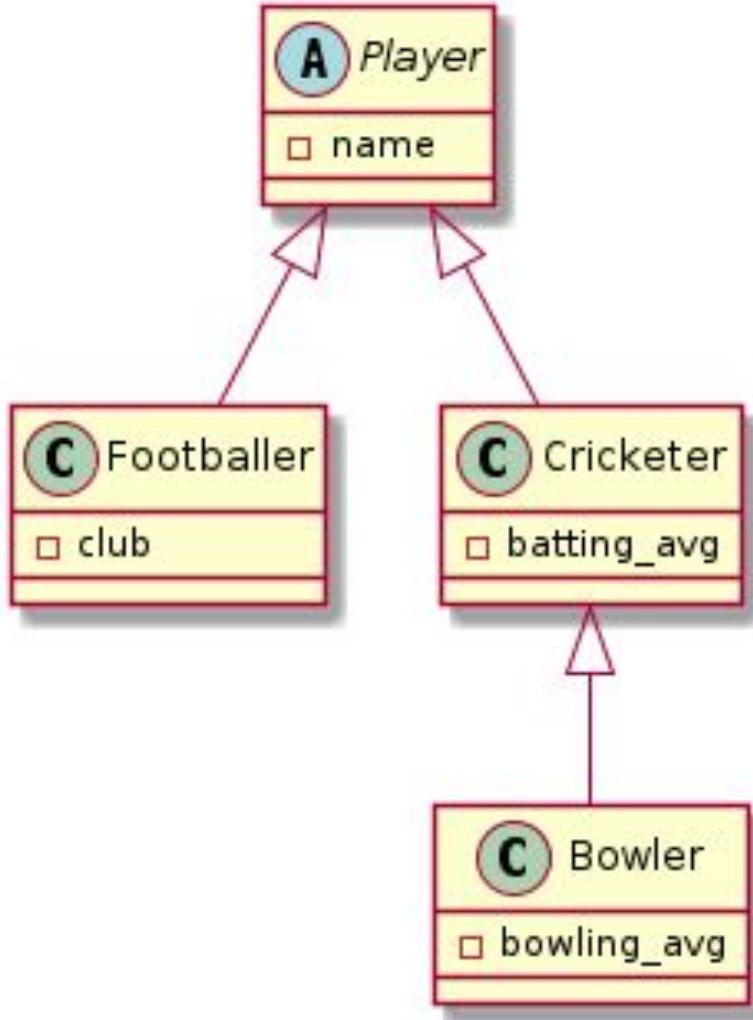
# Traduction de l'héritage



Des sportifs qui peuvent être des footballeurs ou des joueurs de crickets

Les “Bowlers” sont les joueurs de crickets qui lancent la balle

# Exemple



Footballer

nom	club	...
Bobba	Stade Rennais	...

Bowler

nom	batting	bowling	...
Jango	1.2	3.4	...

# Méthodes de traduction

- Arborescence d'héritage == 1 table relationnelle (single table)
- $\forall$  class instanciable, 1 class == 1 table relationnelle
- $\forall$  class (même abstraite), 1 class == 1 table relationnelle

# Arborescence d'héritage == 1 table relationnelle

<b>id</b>	<b>type</b>	<b>nom</b>	<b>club</b>	<b>batting</b>	<b>bowling</b>	<b>...</b>
1	Footballer	Bobba	Stade Rennais	null	null	...
2	Bowler	Jango	null	1.2	3.4	...

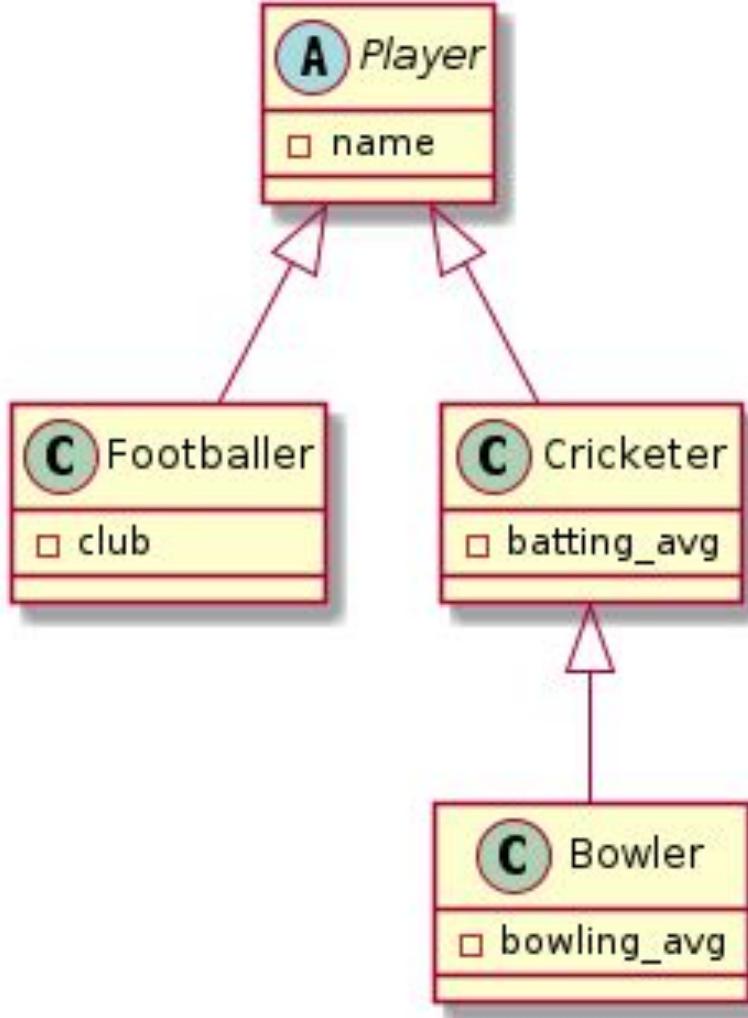
# Avantages

- Simple
- Par défaut
- Courante
- Requêtes et associations polymorphiques

# Inconvénients

- BDD “gruyère” avec possiblement bcp de valeurs NULL
- Impossibilité de mettre certaines contraintes d’intégrité

# ∀ class instanciable, 1 class == 1 table relationnelle



Footballer

nom	id	club	...
Bobba	1	Stade Rennais	...

Cricketer

nom	id	batting	...
Fett	2	7.2	...

Bowler

nom	id	batting	bowling	...
Jango	3	1.2	3.4	...

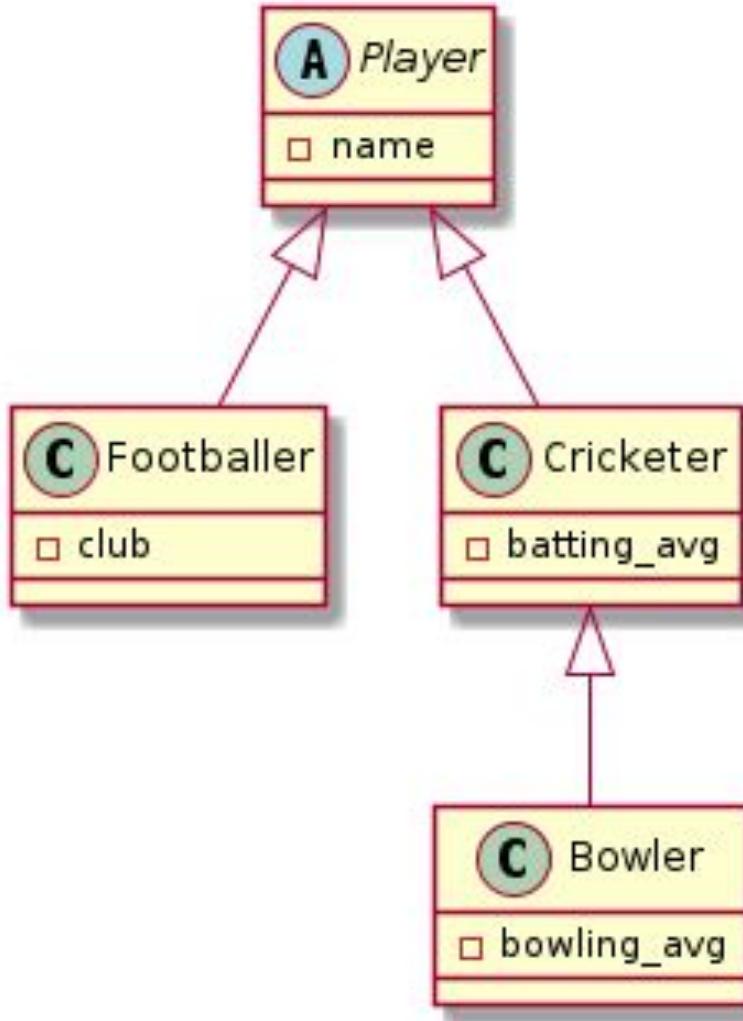
# Avantages

- Naturelle
- Claire
- Pas besoin de faire des jointures

# Inconvénients

- Traduction d'associations polymorphiques impossible
- Requêtes polymorphiques difficiles
- À éviter

∀ class (même abstraite), 1 class == 1 table relationnelle



Player

id	name	Type	...
1	Bobba	F	...
2	Fett	C	...
3	Jango	B	...

Footballer

id	club	...
1	Stade Rennais	...

Cricketer

id	batting	...
2	7.2	...
3	1.2	...

Bowler

id	bowling	...
3	3.4	...

# Préservation de l'identité

- Les données d'un objet sont réparties sur plusieurs tables
- Identité préservée en donnant la même clé primaire aux lignes dans les différentes tables
- Jointure entre les tables pour récupérer les données des classes filles

# Avantages

- Simple : bijection tables - classes
- Requêtes et associations polymorphiques possibles et faciles

# Inconvénients

- ↑ complexité de la hiérarchie => ↑ des jointures
- Les jointures sont coûteuses et donc ↓ performance

# Solution ?

- Aucune solution parfaite
- Seulement des solutions partielles :
  - ignorer les contraintes d'intégrité (prob solution 1)
  - $\emptyset$  polymorphisme (prob solution 2)
  - coût (prob solution 3)

# Stratégie de chargement des objets associés

- Lorsqu'un objet est construit depuis les données de la BDD, il y a 2 stratégies concernant les objets associés :
  - récupération immédiate et création des objets associés
  - création des objets associés quand l'application en a besoin  
=> Réalisé avec le “lazy loading”

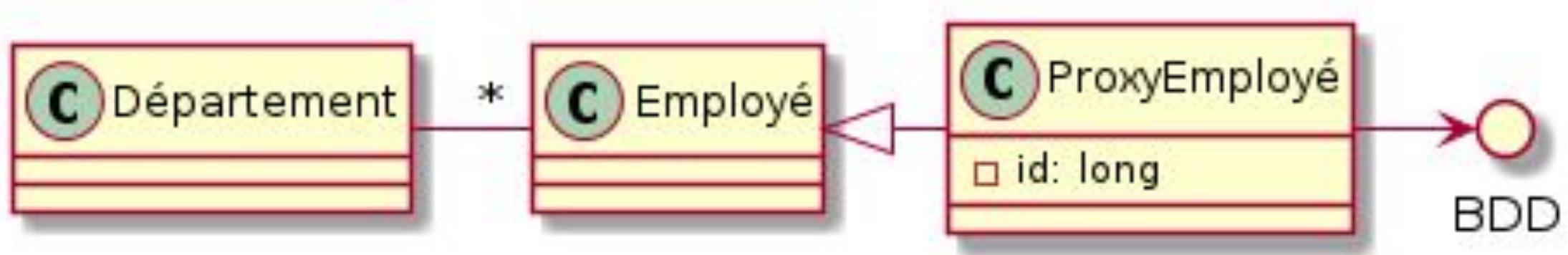
# Exemple

- Recherche dans la BDD d'une facture
- Crédation de l'objet ***Facture*** correspondant
- Est-ce qu'on doit charger les données et créer les objets ***LigneFacture*** associés ?
- Si oui, doit-on charger les données et créer les objets ***Produit*** associés ?
- Si oui...

# Problèmes :

- Soit on risque de créer un nombre très grand d'objets
- Soit on enchaîne les requêtes parce qu'on n'a pas récupéré suffisamment de données (N+1 Select)
- L'environnement de dev peut “cacher” ces problèmes, et lors de la mise en production, les performances ne sont pas acceptables

# Le “lazy loading” example



- Repose sur le design pattern Proxy
- Construit des objets proxy, et interrogera la BDD à la demande

# Conclusion ORM

- Le mapping objet - relationnelle n'est pas une tâche facile
- Les frameworks ORM réduisent grandement le code à produire pour faire de mapping
- Mais tout ne peut être automatisé

# TypeORM : ORM pour TypeScript

- Inspiré par Hibernate et Doctrine
- Simple, flexible et direct
- Tout en “decorators”

# TypeORM : Repositories & Entities

- Repositories : se charge du lien avec la BDD: connection, requête, transformation des données en objet
- Entities : objets que l'on veut sauvegarder l'état en base de données

# Repositories

```
1  @Injectable()
2  export class StudentsService {
3      constructor(
4          @InjectRepository(User)
5          private repository: Repository<User>
6      ) {}
7
8      async getById(idToFind: long): Promise<User> {
9          return await this.repository
10         .findOne({
11             id: Equal(idToFind)
12         });
13     }
14
15 }
```

# Entities

```
1  @Entity()  
2  export class Student {  
3      @PrimaryGeneratedColumn()  
4      public id: number;  
5      @Column()  
6      public name: string;  
7      @ManyToOne(type => Address)  
8      public address: Address;  
9      // ...  
10 }
```

- Cette classe est à sauvegarder en base (== 1 table)
- C'est la PK
- Colonne Simple
- Association

# Colonnes

```
1  @Column()
2  public attribute: number;
3  @PrimaryColumn()
4  public primaryKey: number;
5  @PrimaryGeneratedColumn()
6  public generatedPrimaryKey: number;
7  @Column('uuid')
8  public id: uuid;
9  @Column({type: 'int'})
10 public n: number;
```

# Entité embarquée

```
1 export class Name {  
2     @Column()  
3     public firstname: string;  
4     @Column()  
5     public lastname: string;  
6 }  
7 @Entity()  
8 export class User {  
9     @Column(type => Name)  
10    public name: Name;  
11    // ...  
12 }  
13 @Entity()  
14 export class Employee {  
15     @Column(type => Name)  
16     public name: Name;  
17     // ...  
18 }
```

Table USER

firstname	varchar
lastname	varchar
...	...

Table EMPLOYEE

firstname	varchar
lastname	varchar
...	...

# Associations : OneToOne

```
1 @Entity()  
2 export class User {  
3     @OneToOne(type => Profile)  
4     public profile: Profile;  
5     // ...  
6 }  
7 @Entity()  
8 export class Profile {  
9     @OneToOne(type => User)  
10    @JoinColumn()  
11    public user: User;  
12    // ...  
13 }
```

## Table USER

<b>id</b>	<b>int</b>	<b>Primary Key</b>
...	...	...

## Table PROFILE

<b>id</b>	<b>int</b>	<b>Primary Key</b>
<b>userId</b>	<b>int</b>	<b>Foreign Key</b>
...	...	...

# Associations : ManyToOne

```
1 export class User {  
2     // ...  
3 }  
4 export class Photo {  
5     @ManyToOne(type => User)  
6     public user: User;  
7     // ...  
8 }
```

Table USER

id	int	Primary Key
...	...	...

Table PHOTO

id	int	Primary Key
userId	int	Foreign Key

# Associations : OneToMany

```
1 export class User {  
2     @OneToMany(() => Photo,  
3             photo => photo.user)  
4     public photos: Photo[]  
5     // ...  
6 }  
7 export class Photo {  
8     @ManyToOne(type => User,  
9             user => user.photos)  
10    public user: User;  
11    // ...  
12 }
```

## Table USER

<b>id</b>	<b>int</b>	<b>Primary Key</b>
...	...	...

## Table PHOTO

<b>id</b>	<b>int</b>	<b>Primary Key</b>
<b>userId</b>	<b>int</b>	<b>Foreign Key</b>
...	...	...

# Associations : ManyToMany

```
1 export class User {  
2     @ManyToMany(() => Role)  
3     @JoinTable()  
4     roles: Role[];  
5 }  
6 export class Role { /* ... */ }
```

Table ROLE-USER

<b>idRole</b>	<b>int</b>	<b>Primary Key,</b> <b>Foreign Key</b>
<b>idUser</b>	<b>int</b>	<b>Primary Key,</b> <b>Foreign Key</b>

Table USER

<b>id</b>	<b>int</b>	<b>Primary Key</b>
...	...	...

Table ROLE

<b>id</b>	<b>int</b>	<b>Primary Key</b>
...	...	...

# Héritage avec TypeORM

## Footballer

nom	id	club	...
Bobba	1	Stade Rennais	...

## Cricketer

nom	id	batting	...
Fett	2	7.2	...

## Bowler

nom	id	batting	bowling	...
Jango	3	1.2	3.4	...

```
1  @Entity()
2  export abstract class Player {
3      @PrimaryGeneratedColumn()
4      public id: number;
5      @Column()
6      public name: string;
7  }
8  @Entity()
9  export class Footballer extends Player {
10     @Column()
11     public club: string;
12 }
13 @Entity()
14 export class Cricketer extends Player {
15     @Column()
16     public batting_avg: number;
17 }
18 @Entity()
19 export class Bowler extends Cricketer {
20     @Column()
21     public bowling_avg: number;
22 }
```

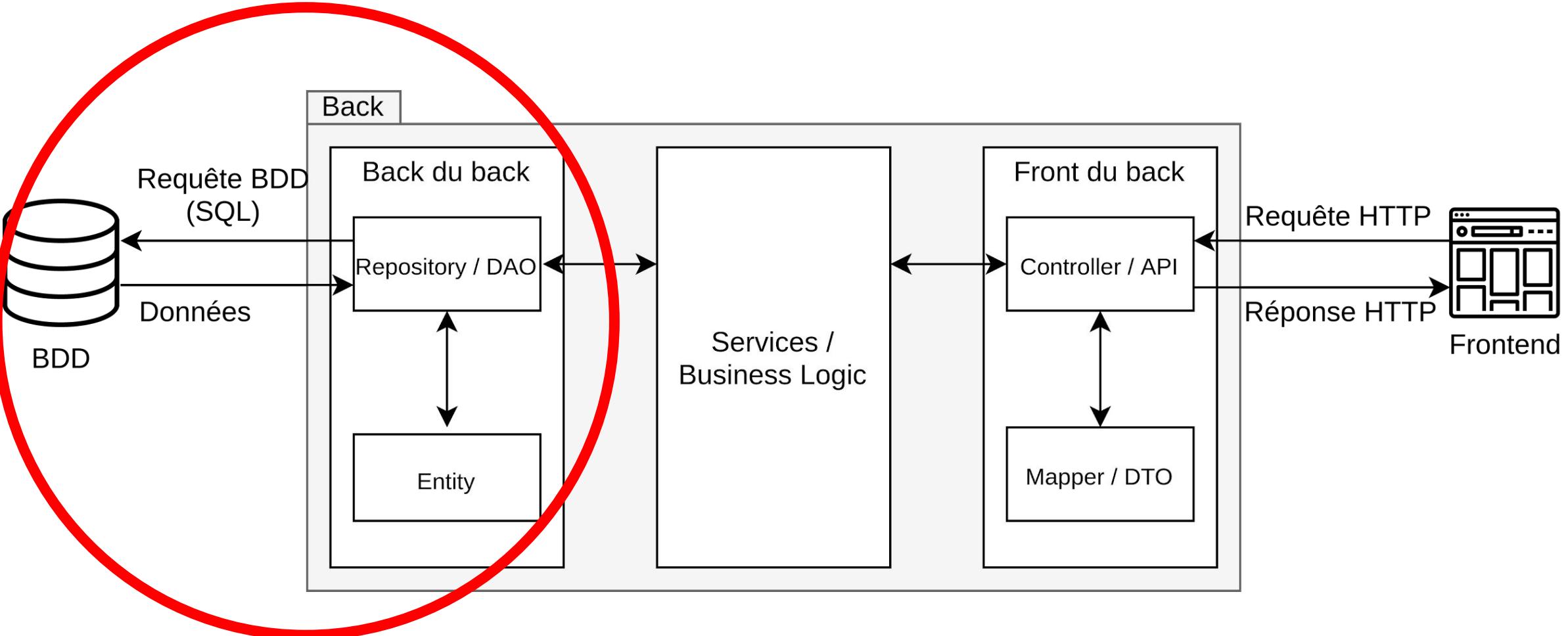


# Héritage avec TypeORM

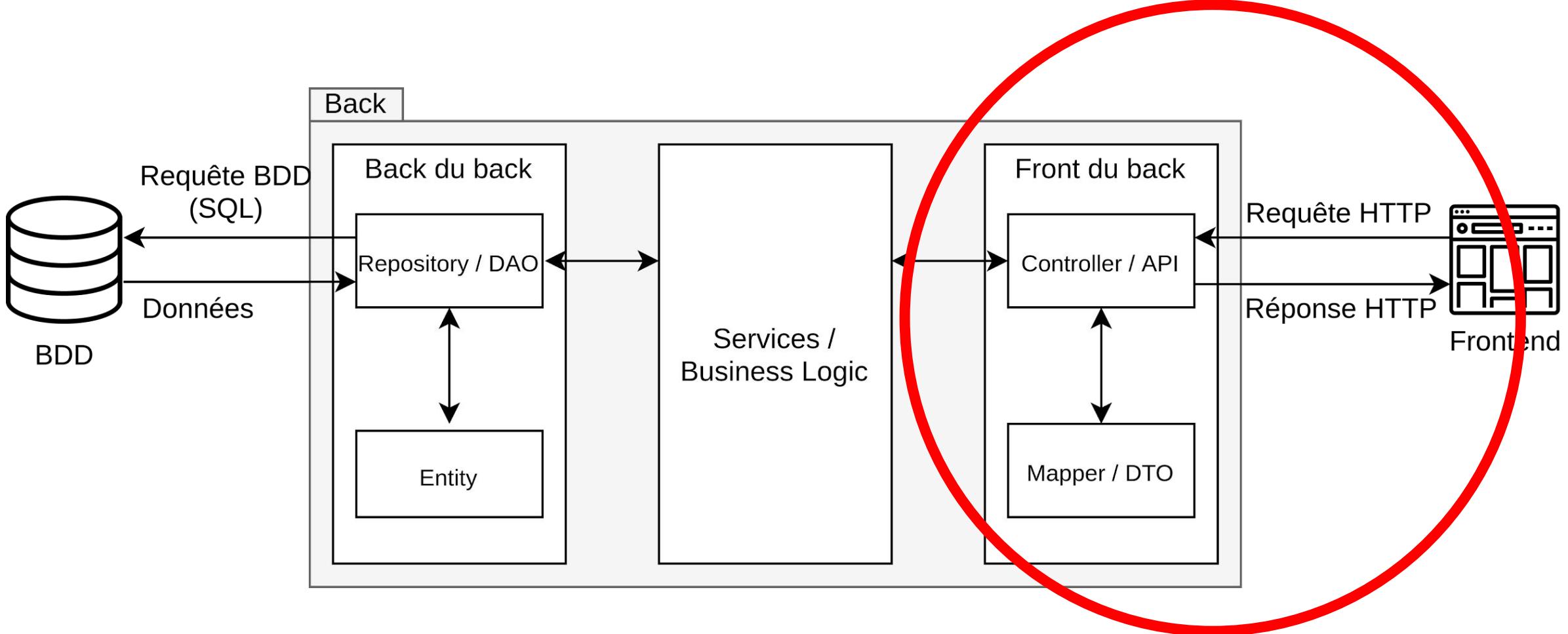
i d	type	no m	club	batti ng	bowli ng
1	Footbal ler	Bob ba	Stade Renn ais	null	null
2	Bowler	Jan go	null	1.2	3.4

```
1  @Entity()  
2  @TableInheritance({ column: { type: "varchar", name: "type" } })  
3  export abstract class Player {  
4      @PrimaryGeneratedColumn()  
5      public id: number;  
6      @Column()  
7      public name: string;  
8  }  
9  @ChildEntity()  
10 export class Footballer extends Player {  
11     @Column()  
12     public club: string;  
13 }  
14 @ChildEntity()  
15 export class Criketer extends Player {  
16     @Column()  
17     public batting_avg: number;  
18 }  
19 @ChildEntity()  
20 export class Bowler extends Criketer {  
21     @Column()  
22     public bowling_avg: number;  
23 }
```

# Partie vue : Back du back



# Prochaine partie : Front du Back



# API REST, OpenAPI et bonnes pratiques

- API REST : bonnes pratiques
- Un mot sur le standard OpenAPI

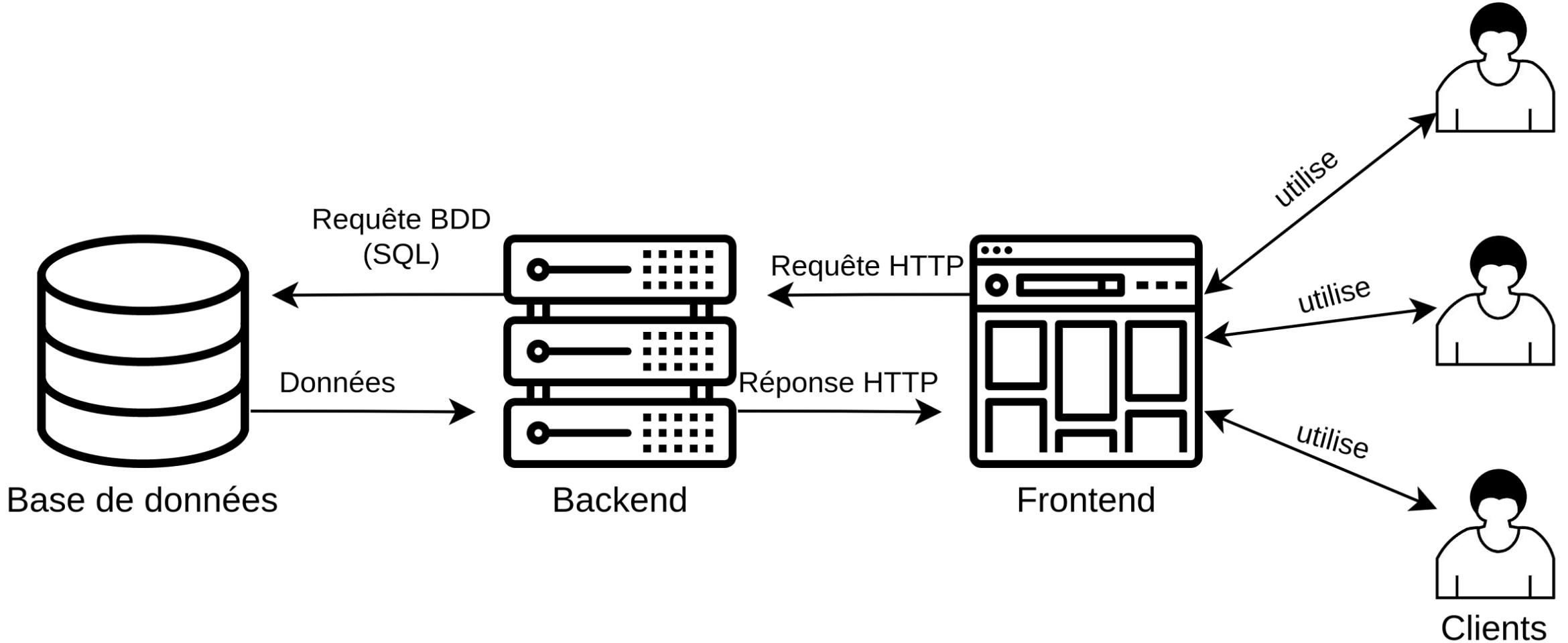
# Intro à SOA

- Service Oriented Architecture
- Agnostique du protocole de communication
- Repose sur des normes W3C : SOAP et WSDL
- Outillé, structuré et spécifié
- Lourd
- 0 Performance

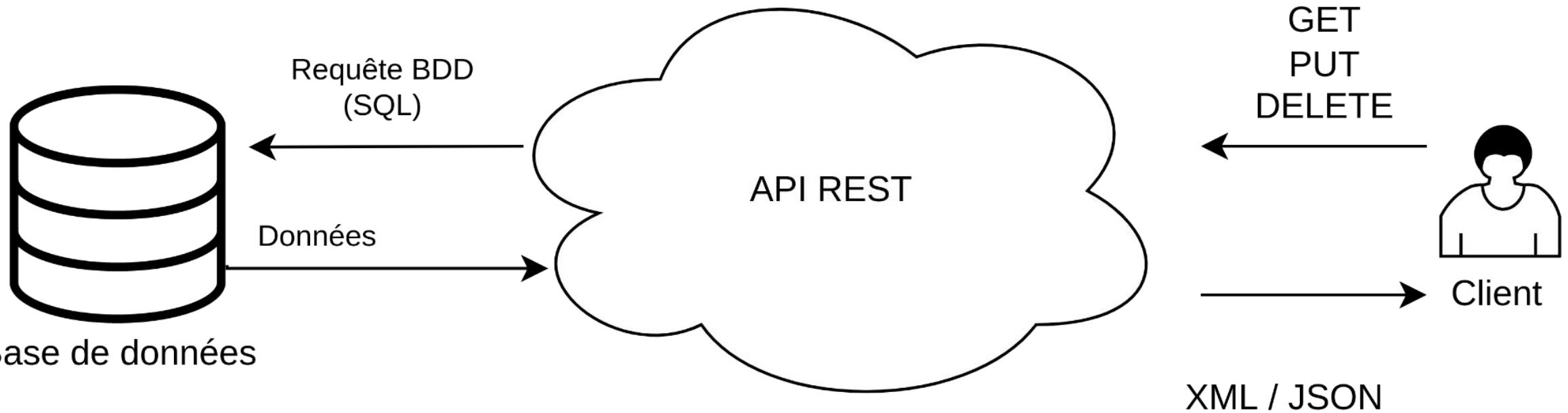
# Intro à REST

- Orienté Ressources
- REpresentational State Transfert ; Inventé par R Fielding (2000)
- Statelessness
- Basé sur une sémantique des méthodes HTTP
- Basé sur les codes de retour

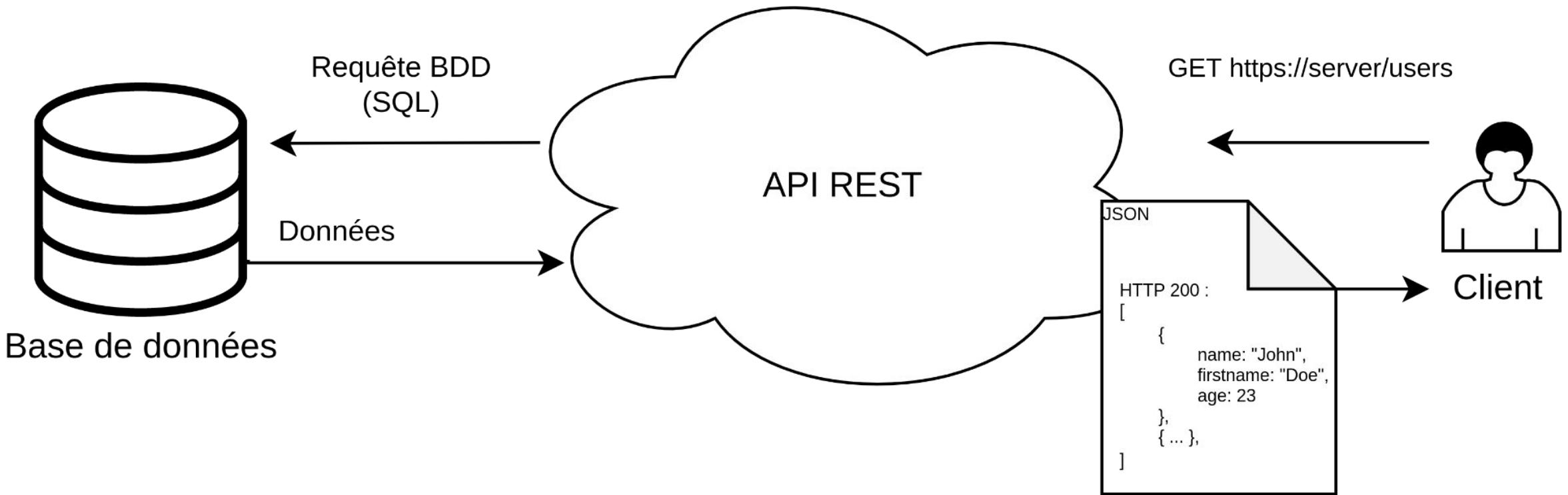
# Architecture REST



# Architecture REST



# Architecture REST



# Avantages de REST

- Simple et Souple (vs SOA)
- Très bonnes performances, capacité de monter en charge
- Portabilité
- Résilience

# API REST : bonnes pratiques

- A priori, pas de règle stricte
- MAIS ! De bonnes pratiques pour aider les consommateurs (ou vous-mêmes)

# API REST : bonnes pratiques

- `https://server/resources?id=12435`
- `https://server/resources/12435`
- Utiliser des mots standards, et non du métier
- Utiliser des noms, pas des verbes
- Les ressources au pluriel
- Cohérence dans toutes l'api
  - `snake_case` ou `CamelCase`
  - code de retour unifié

# API REST : bonnes pratiques - Codes de Retour

- Fournir un code de retour à chaque requête
- 200 -> OK
- 201 -> Resource created
- 400 -> bad request
- 401 -> Unauthorized
- 403 -> Forbidden
- 404 -> Not Found
- 500 -> Internal Server Error

# API REST : bonnes pratiques - CRUD

- Implémenter pour toutes les ressources, un CRUD

<b>Create</b>	<b>Retrieve</b>	<b>Update</b>	<b>Delete</b>
<b>POST</b>	<b>GET</b>	<b>PUT</b>	<b>DELETE</b>
Crée une ressource	Récupère une ressource	Modifie une ressource	Supprime une ressource

# API REST : bonnes pratiques - Versionning

- <https://server/v1/resources>
- <https://server/v1/resources>
- <https://server/v1.1/resources>
- <https://server/v1.2/resources>

# API REST : bonnes pratiques - Réponses Partielles

- Être capable de retourner une partie des données pour ne pas encombrer la bande passante

```
$ GET /users/1?fields=id,firstname
{ "id": 1, "firstname": "John" }
```

# API REST : bonnes pratiques - Tri des Données

- Offrir la possibilité de trier les données retournées

```
$ GET /users?sort=firstname
[{"id": 1654, "firstname": "aA", "name": "toto", ... },
 {"id": 8713, "firstname": "aB", "name": "tutu", ... },
 ...
]
```

# API REST : bonnes pratiques - Pagination

- Pouvoir gérer le plus grand volume de données page par page

```
$ GET /users?range=10-21
[ { "id": 11, "firstname": "Bobba", "name": "Fett", ... },
  { "id": 12, "firstname": "Jango", "name": "Fett", ... },
  ...
]
```

# API REST : bonnes pratiques - Mots-Clés Réserveés

- Mots-clés pour les apis qui retournent des listes :

```
$ GET /users/count
30000
$ GET /users/first
{
  "id": 1,
  "firstname": "John",
  ...
}
$ GET /users/last
{
  "id": 29999,
  "firstname": "Nobody",
  ...
}
```

# API REST : bonnes pratiques - Filtrage

- Pouvoir filtrer les données pour n'afficher que des données requises par l'utilisateur

```
$ GET /users?firstname=John
[ { "id": 1, "firstname": "John", "name": "Doe", ... },
  { "id": 123, "firstname": "John", "name": "Dupont", ... },
  ...
]
```

# Standard OpenAPI

- Spécifie un format de documentation d'API
- Issu du projet Swagger
- Des vues graphiques permettent de visualiser et même de tester l'API
- Démo : <https://github.com/OpenAPITools/openapi-petstore>

# Standard OpenAPI : exemple

The screenshot displays the Petstore API documentation, which is a standard OpenAPI specification. It includes two main sections: 'pet' and 'store'. The 'pet' section contains various operations for managing pets in a store, such as updating existing pets, adding new pets, finding pets by status or tags, and deleting pets. The 'store' section contains an operation for getting pet inventories by status. Each operation is represented by a colored button indicating the HTTP method (e.g., PUT, POST, GET, DELETE) and a brief description of its purpose. A lock icon is present next to each operation, likely indicating a secured endpoint.

**pet** Everything about your Pets

PUT /pet Update an existing pet

POST /pet Add a new pet to the store

GET /pet/findByStatus Finds Pets by status

GET /pet/findByTags Finds Pets by tags

GET /pet/{petId} Find pet by ID

POST /pet/{petId} Updates a pet in the store with form data

DELETE /pet/{petId} Deletes a pet

POST /pet/{petId}/uploadImage uploads an image

**store** Access to Petstore orders

GET /store/inventory Returns pet inventories by status

# Standard OpenAPI : exemple

**GET** /pet/{petId} Find pet by ID 

Returns a single pet

**Parameters** 

Name	Description
<b>petId</b> * required integer (path)	ID of pet to return

# Standard OpenAPI : exemple

GET /pet/{petId} Find pet by ID 

Returns a single pet

**Parameters** 

Name	Description
<b>petId</b> * required <small>integer (path)</small>	ID of pet to return <input type="text" value="1"/>

**Execute** **Clear**

# Standard OpenAPI : exemple

The screenshot shows a user interface for testing an API. At the top, there are two buttons: "Execute" (highlighted in blue) and "Clear". Below this is a section titled "Responses" which contains a "Curl" block with the following command:

```
curl -X GET "http://localhost:8080/v3/pet/1" -H "accept: application/xml" -H "api_key: special-key"
```

Below the curl block is a "Request URL" field containing:

```
http://localhost:8080/v3/pet/1
```

Under the "Responses" section, there is a "Server response" table:

Code	Details
200	<b>Response body</b> <pre>&lt;Pet&gt;   &lt;id&gt;1&lt;/id&gt;   &lt;category&gt;     &lt;id&gt;2&lt;/id&gt;     &lt;name&gt;Cats&lt;/name&gt;   &lt;/category&gt;   &lt;name&gt;Cat 1&lt;/name&gt;</pre>

# Standard OpenAPI : exemple

200

## Response body

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Pet>
  <category>
    <id>1</id>
    <name>Hola</name>
  </category>
  <id>1</id>
  <name>Perrito</name>
  <photoUrls>
    <photoUrl>/tmp/inflector2222377630615120221.tmp</photoUrl>
  </photoUrls>
  <status>available</status>
  <tags>
    <tag>
      <id>1</id>
      <name>Bizcocho</name>
    </tag>
  </tags>
</Pet>
```

# Standard OpenAPI : exemple

Curl

```
curl -X 'GET' \
'https://petstore3.swagger.io/api/v3/pet/23' \
-H 'accept: application/xml'
```

Request URL

```
https://petstore3.swagger.io/api/v3/pet/23
```

Server response

Code	Details
404	Error: Not Found Response body Pet not found

# Outils Swagger

- Swagger Editor : éditeur d'API pour la création d'APIs suivant les spécifications d'OpenAPI
- Swagger UI : visualisations des spécifications OpenAPI dans une interface utilisateur interactive
- Swagger Codegen : générer des squelettes de code serveur et de code à partir de spécification OpenAPI

# Sécurité

- Sécuriser son API
- Sécuriser ses mots de passe
- HTTPs
- Authorization & Authentication

# Sécuriser son API

- 5 Mythes autour de la sécurité :
  - Sans lui demander, le développeur fournit une solution sécurisée
  - Seules quelques personnes savent exploiter les failles des applications web
  - SSL suffit à protéger mon site web
  - Un firewall suffit à me protéger des attaques
  - Une faille sur une application web n'est pas importante

# Sécuriser son API

- OWASP : Open Web Application Security Project
- OWASP ZAP (Zed Attack Proxy) : Outil d'analyse des failles de sécurité des applications web
- Liens utiles :
  - [We're under attack! 23+ Node.js security best practices](#)
  - [Fixing OWASP Top 10](#)

# Exemple d'attaque : Denial of Service attack

- But : rendre inaccessible l'application web
- Méthode : surcharger le serveur de requêtes
- Très simple à mettre en place
- Contre-mesure : limiter le nombre de requêtes par client
- Solution : frontal gateway (nginx), load balancer, firewall ou intergiciel (comme express-rate-limit)
- En plus, ces solutions protègent des attaques par force brute pour la recherche de mot de passe

# Exemple d'attaque : ClickJacking

- But : récupérer des informations privées
- Méthode : embarquer le site dans une iframe et afficher des éléments graphiques pour faire cliquer l'utilisateur sur des éléments piégés
- Contre-mesure : indiquer au navigateur que le contenu ne peut pas être embarqué dans une balise frame, iframe ou object
- Solution : utiliser le paramètre X-Frame-Options dans le header

# Helmet

- Helmet réunit des modules pour se protéger d'un certain nombre de failles



```
1 import * as helmet from 'helmet';
2 // somewhere in your initialization file
3 app.use(helmet());
```

# Sans Helmet

```
$ curl -i http://localhost:3000/v1/users/
```

...

HTTP/1.1 200 OK

**X-Powered-By: Express**

Content-Type: application/json; charset=utf-8

Content-Length: 367

ETag: W/"16f-52pT8zxFPOAtUF1b4F57cArTZs"

Date: Mon, 11 Feb 2019 20:08:26 GMT

Connection: keep-alive

# Avec Helmet

```
$ curl -i http://localhost:3000/v1/users/
```

...

HTTP/1.1 200 OK

X-DNS-Prefetch-Control: off

X-Frame-Options: SAMEORIGIN

Strict-Transport-Security: max-age=15552000; includeSubDomains

X-Download-Options: noopen

X-Content-Type-Options: nosniff

X-XSS-Protection: 1; mode=block

# Npm audit

```
$ npm audit  
# npm audit report
```

```
axios <0.21.1  
Severity: high  
Server-Side Request Forgery -  
https://npmjs.com/advisories/1594  
fix available via `npm audit fix`
```

# Sécuriser les mots de passe : stockage

- On peut déléguer la gestion de mot de passe à un Identity provider (IdP)
- Mais il arrive que pour de petites applications, la gestion des mots de passe est faite en interne
- Aucun problème à condition de ne jamais stocker en clair les mots de passe dans la base !

# Solution : chiffrer les mdps ?

- Et si on chiffrait les mots de passe ?
- Ça ne déplace que le problème sur les clés de chiffrage !
- Si la clé est volée, la sécurité entière est compromise
- Il existe des boîtiers hardware sécurisés, mais c'est très cher...

# Solution : utilisation du hash

- Au lieu du mot de passe, on stock son hash
- Mais il faut utiliser une fonction de hachage avec les bonnes propriétés (comme Sha-256) !
- Ajout d'un (grain) de sel pour augmenter la sécurité
- Pour identifier un user, on compare le hash stocké au hash calculé à partir de l'input de l'user (plus le sel).

# Utilisation du hash : exemple

“m0nPa\$\$W0rD” + “this\_is\_a\_secret\_salt”



Sha-256()



a2003c53f87c1e5ef6efcebb3f53c74c3720887494c4a112fe39fcd2  
03ecf3b3 ← C'est ce qu'on stockera  
dans la BDD

# HTTPs : objectifs

- HTTP est lisible par des outils comme WireShark (sniffer, analyse de réseau)
- HTTPs : chiffrement SSL / TLS entre le client et le serveur
- HTTPs : sécurisation du protocole HTTP avec 3 composantes :
  - Intégrité
  - Protection des échanges (Privacy)
  - Authentification

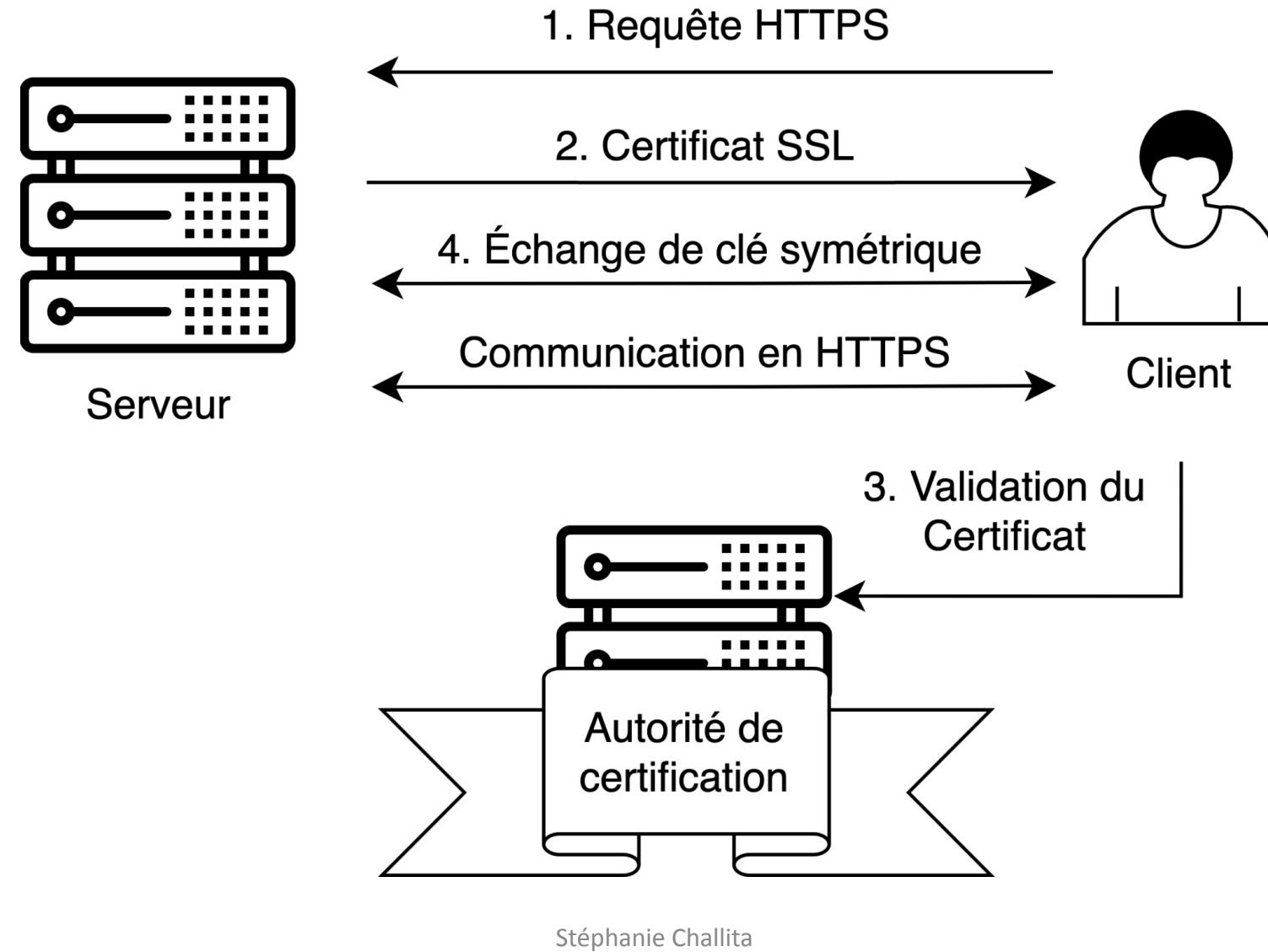
# HTTPs : fonctionnement

- Utilise un certificat
- Chiffrage asymétrique : clé publique / clé privée pour l'envoi de la clé de chiffrage symétrique
- Chiffrage du message HTTP avant envoi (client)
- Déchiffrage du message à la réception (serveur)
- L'ensemble de la pile HTTP est cryptée : Header + Body. (seul l'ip reste lisible)

# HTTPs : certification

- Requis pour le handshake et le chiffrement
- Délivré par des Autorités de confiance (Verisign, Thawte, etc)
- Coût : entre 200€ et 500€
- Basé sur le standard X509
- Certificat auto-signé pour les tests

# HTTPs : certification



# HTTPs : authentification

- Lourd, donc souvent fait autrement
- Avoir un certificat par poste est requis
- Authentification de l'application cliente, et non pas de l'user
- Mise en place délicate : création, distribution et update des certificats

# OAuth2 : authorization

- Protocole de délégation d'autorisation (pas d'authentification)
- le **access\_token** est transmis dans le header HTTP :
  - Authorization: Bearer 34EF5EF9.5435DEE.54533EE6E
- Le serveur vérifie le token et sa validité à chaque requête

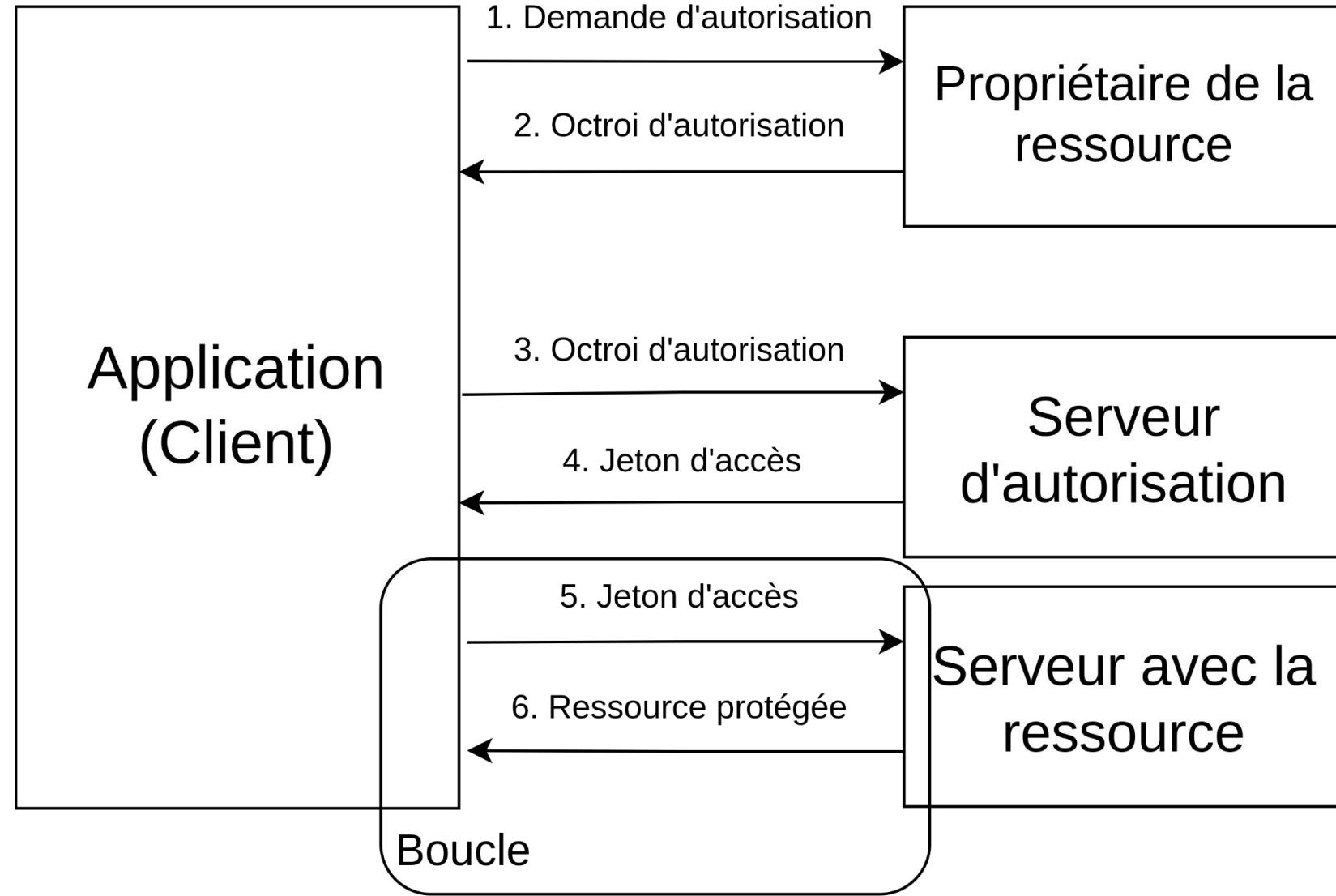
# OAuth2 : acteurs

- **Resource Owner** : Propriétaire des ressources, il est le seul à pouvoir déléguer les autorisations
- **Resource Server** : Machine qui hoste les ressources
- **Authorization Server** : Serveur d'autorisation
- **Client** : Application qui souhaite accéder à une ressource

# OAuth2 : scénario

- Je (resource owner) souhaite donner à une application sur mon smartphone (client) des droits sur une ressource qui m'appartient et qui est hébergée par un tiers (Resource server)
- Exemple : donner un accès à mon compte dropbox pour que l'appli puisse y stocker des données

# OAuth2 : workflow



# OpenID connect : OAuth2 + JWT

- OAuth2 s'occupe des autorisations
- OIDC ajoute une couche d'authentification grâce au JWT

# JSON Web Token (JWT)

- Format standardisé

headers . content . signature

- Chaque partie est encodée en base 64

headers

Algorithme à  
utiliser pour  
valider le jeton

content

Données au  
format JSON

signature

Signature (hash)  
du headers et  
du content

# JWT Exemple

## Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ  
JuYW1lIjoiQm9iYmEgRmV0dCIiImlkIjoyM30.O  
_bmDGagYCISGPizQU62hwXVQkpaGYiAdnWusr6I  
ViU
```

## Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYOUT: DATA

```
{  
  "name": "Bobba Fett",  
  "id": 23  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)  secret base64 encoded
```

# OIDC : id\_token et access\_token

- id\_token : contient les informations concernant l'utilisateur
- access\_token : le même que OAuth2 mais standardisé (JWT), donne l'autorisation d'accès aux ressources

# OIDC : workflow

