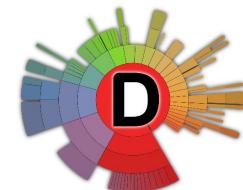


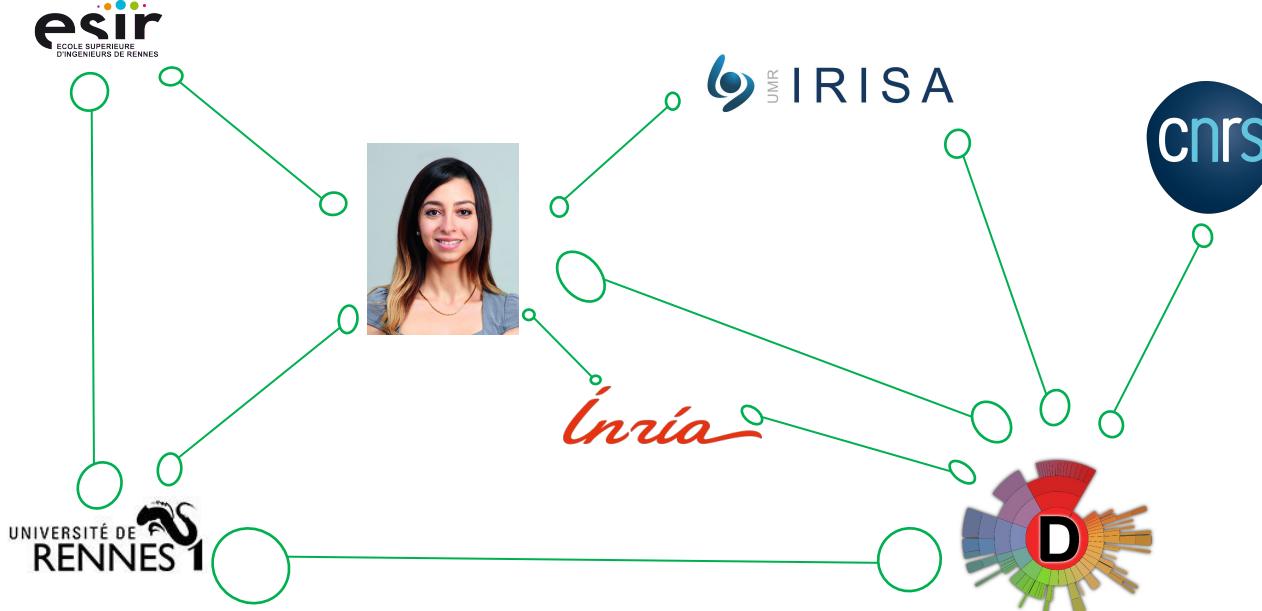
# Web - ESIR 2

Cours développement Web - partie Backend  
Septembre 2025



# Qui suis-je ?

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



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

<https://www.diverse-team.fr/>

# Organisation générale

- 7 CMs sur le Backend
- 5 CMs sur le Frontend
- 18 TPs -> Projet

# Organisation - partie Backend

- 7 cours (~10h)
- 10 séances projet (15h)

- 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

- 5 cours (~8h)
- 8 séances projet (12h)

- 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 (3h) le **16/01/2026** de **13:15 à 16:15**

## 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 (3h) le **16/01/2026** de **13:15 à 16:15**

## Partie Frontend

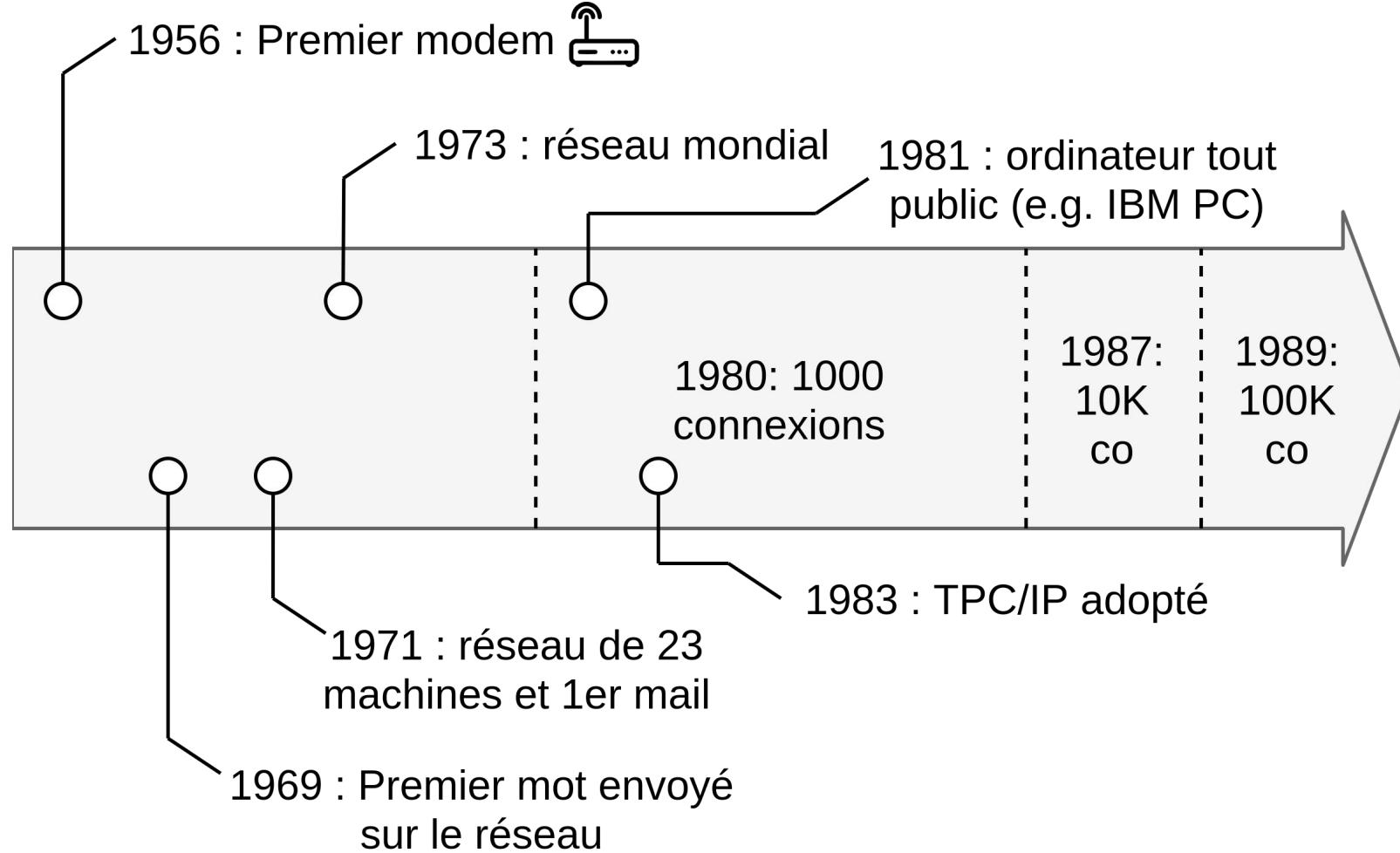
- S'authentifier en tant qu'utilisateur
- Gérer un utilisateur :  
    création, mise à jour et suppression
- Gérer une association :  
    création, mise à jour et suppression
- Lister les utilisateurs et les associations
- Rechercher un utilisateur ou une association

# Table des matières

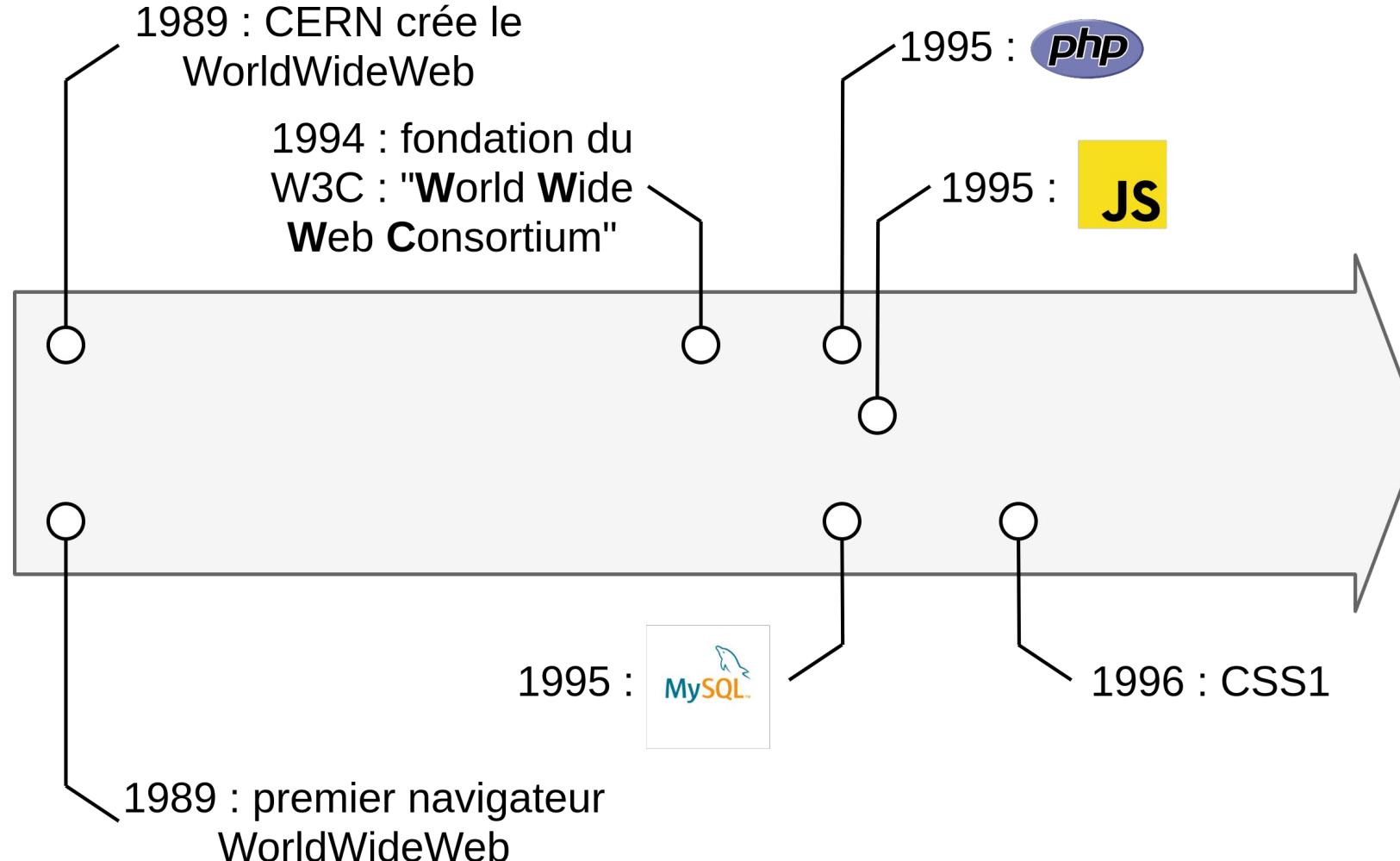
- Généralités
- Coup d'oeil sur le frontend (côté client)
- Le backend en détails (côté server)
- Un peu de technique avec NestJS
- De la base de données aux objets (ORM)
- API REST, OpenAPI & bonnes pratiques
- Sécurité : Autorisation & Authentification
- Accessibilité

# Généralités

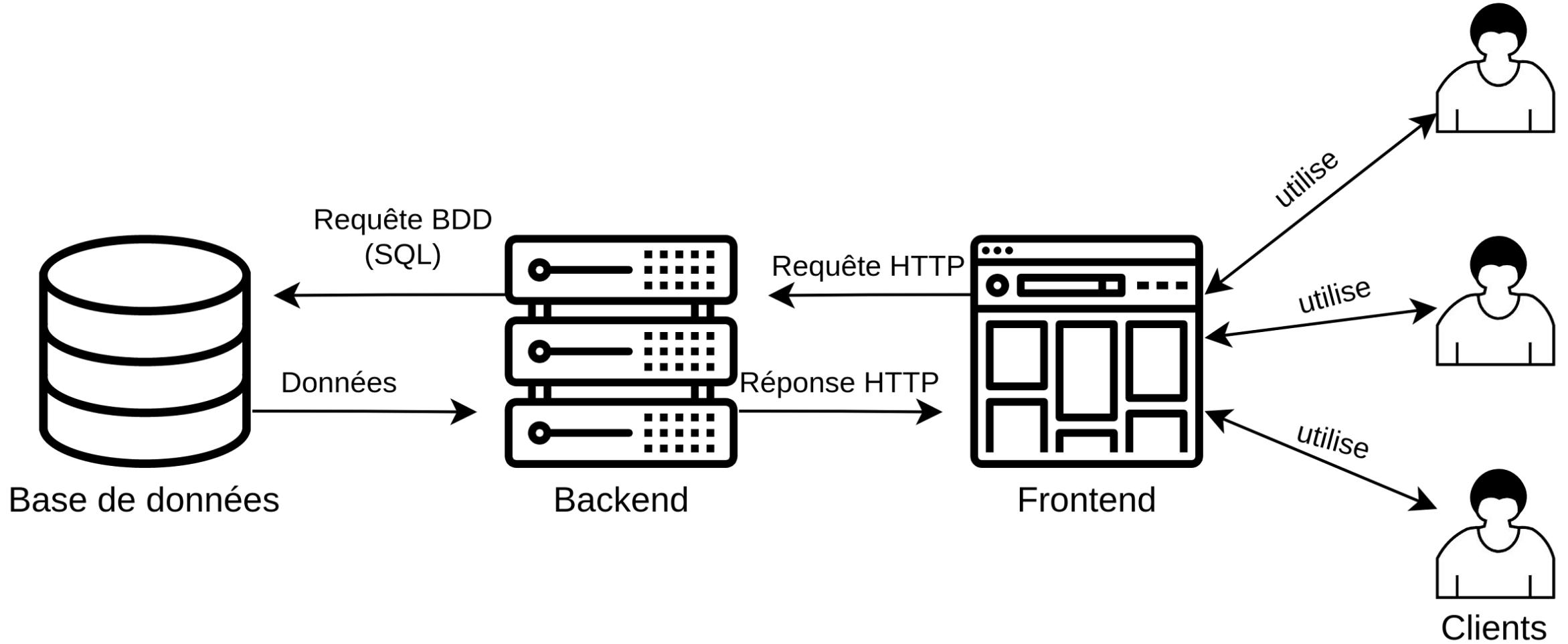
# Généralités : histoire du web



# 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 2020, 10.000 développeurs<sup>1</sup>, avec une demande stable en Europe et en croissance en France<sup>2</sup>
- Innovations
- Diversité

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

2: <https://www.cedefop.europa.eu/en/tools/skills-online-vacancies/ict-specialists>

# 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 !

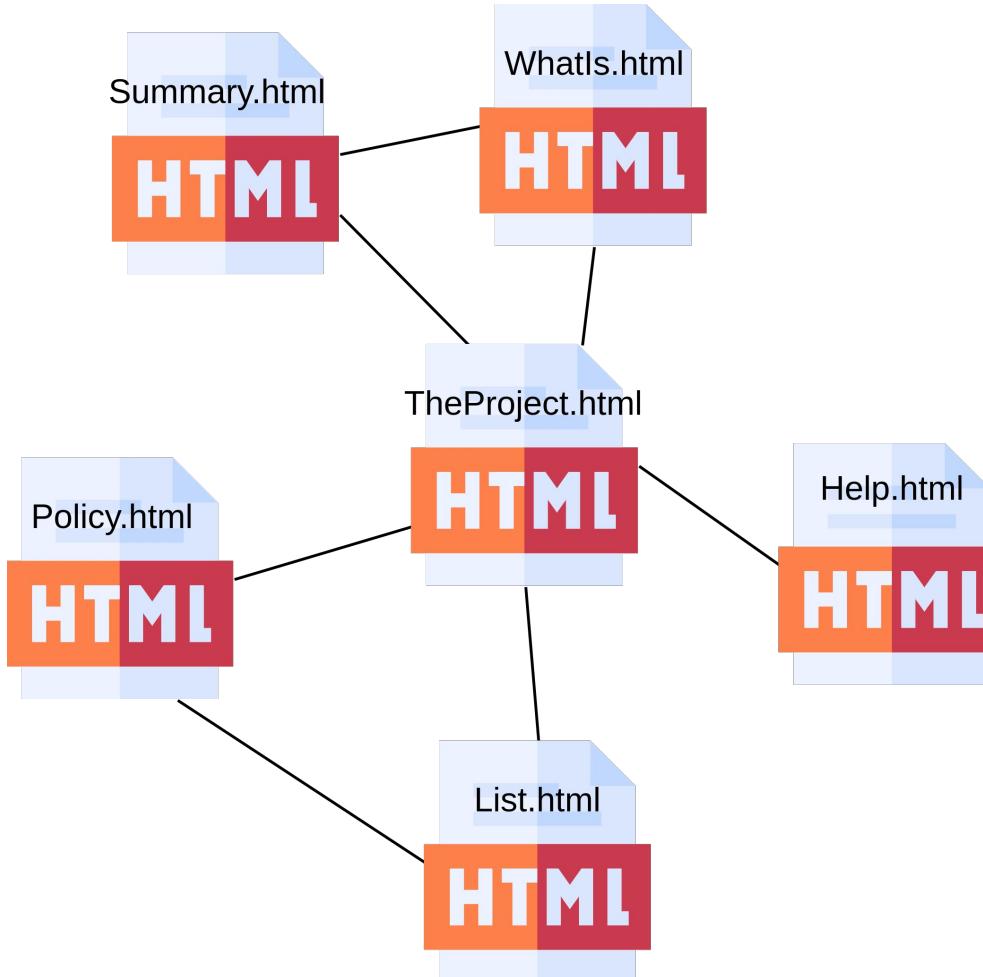
- Un cours fullstack développeur
- Un cours de TypeScript

# 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 ?

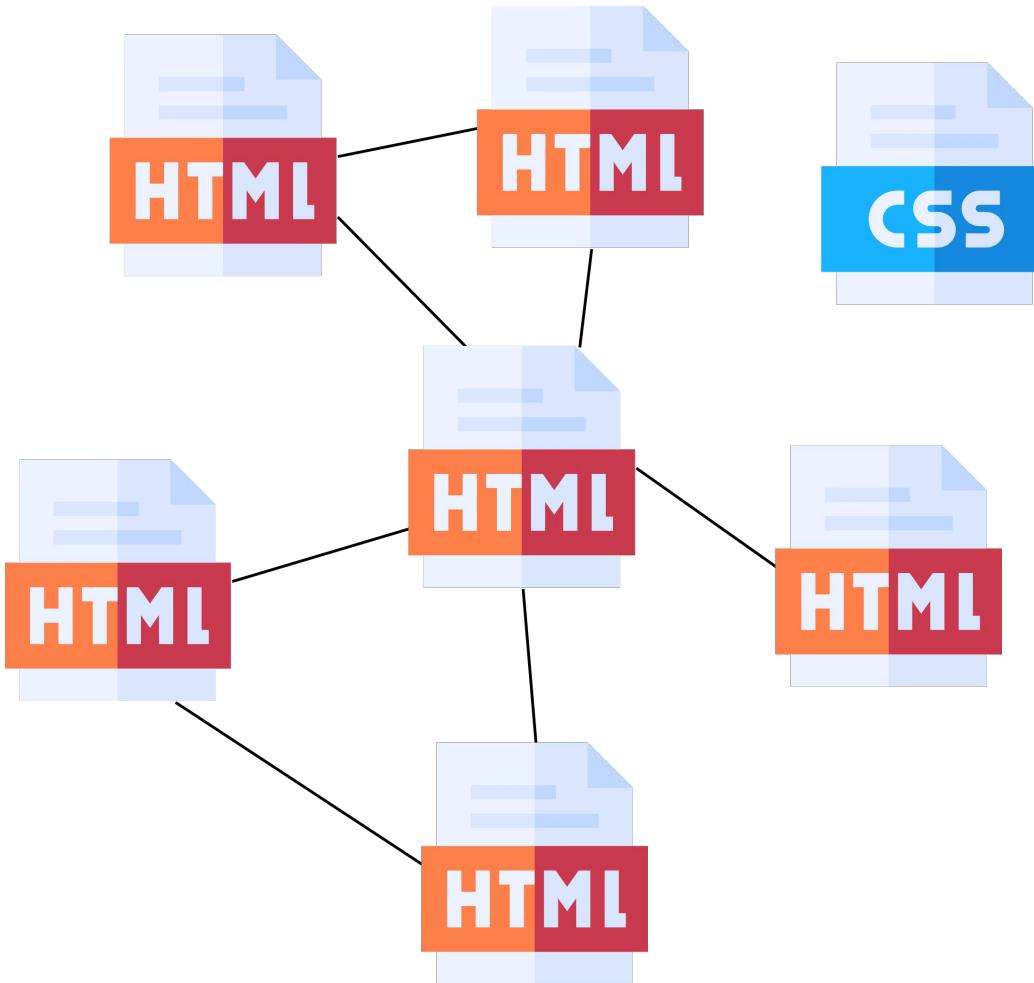
If you would like to support the web..

### Getting code

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

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

# HTML & CSS



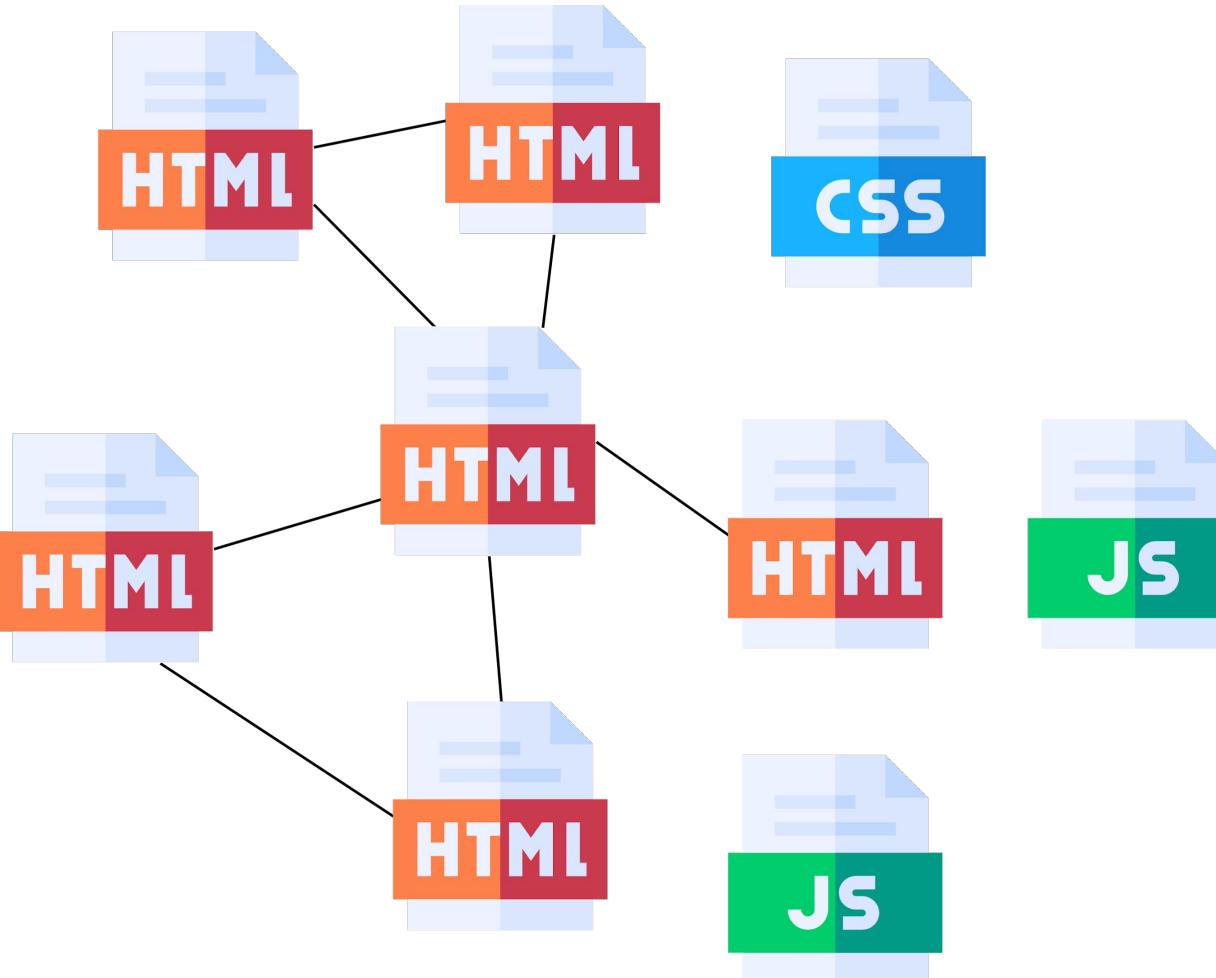
```
h1 {  
    font-family: courier, courier-new, serif;  
    font-size: 20pt;  
    color: blue;  
    border-bottom: 2px solid blue;  
}  
p {  
    font-family: arial, verdana, sans-serif;  
    font-size: 12pt;  
    color: #6B6BD7;  
}  
.red_txt {  
    color: red;  
}
```

# HTML & CSS



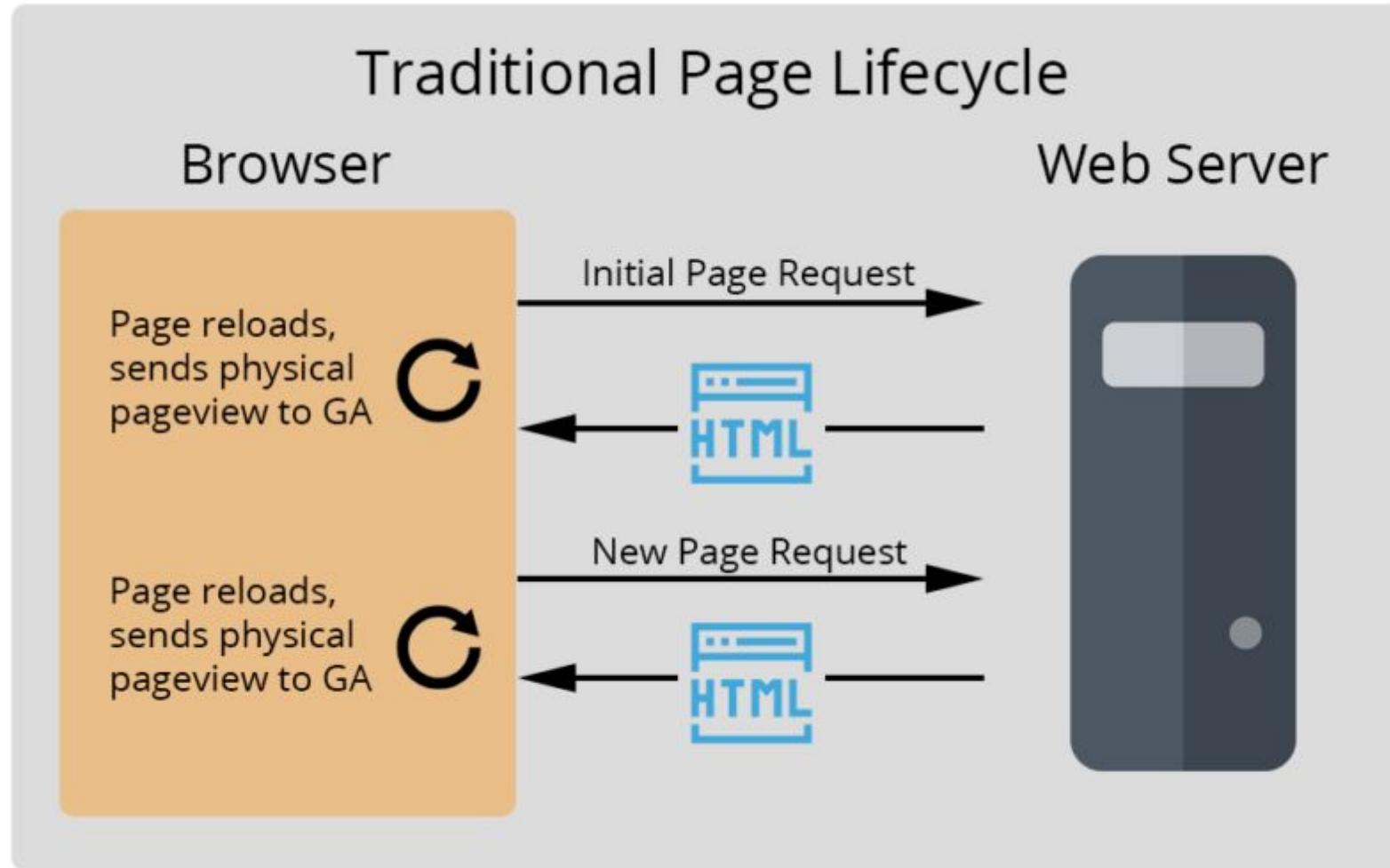
source: <http://krakoukas.com/mieux-avant/wayback-machine/>

# JavaScript

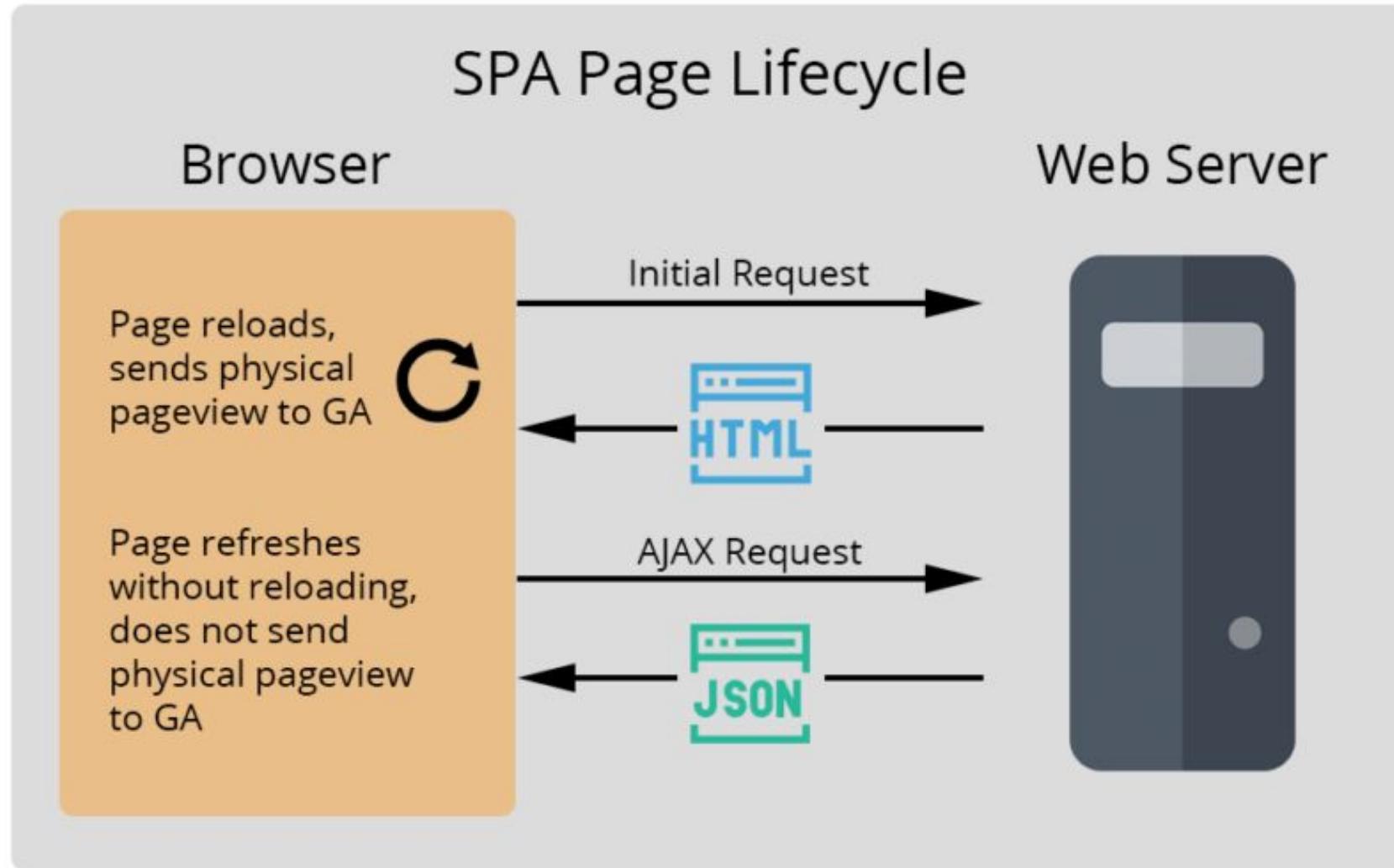


```
$function() {  
  
    function saveState() {  
        if (this.options.saveState == true) {  
            if (this.state == "normal") {  
                $.ajax({  
                    url : "bla"  
                });  
            } else {  
                $.ajax({  
                    url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

# Multi Pages Applications (MPA)

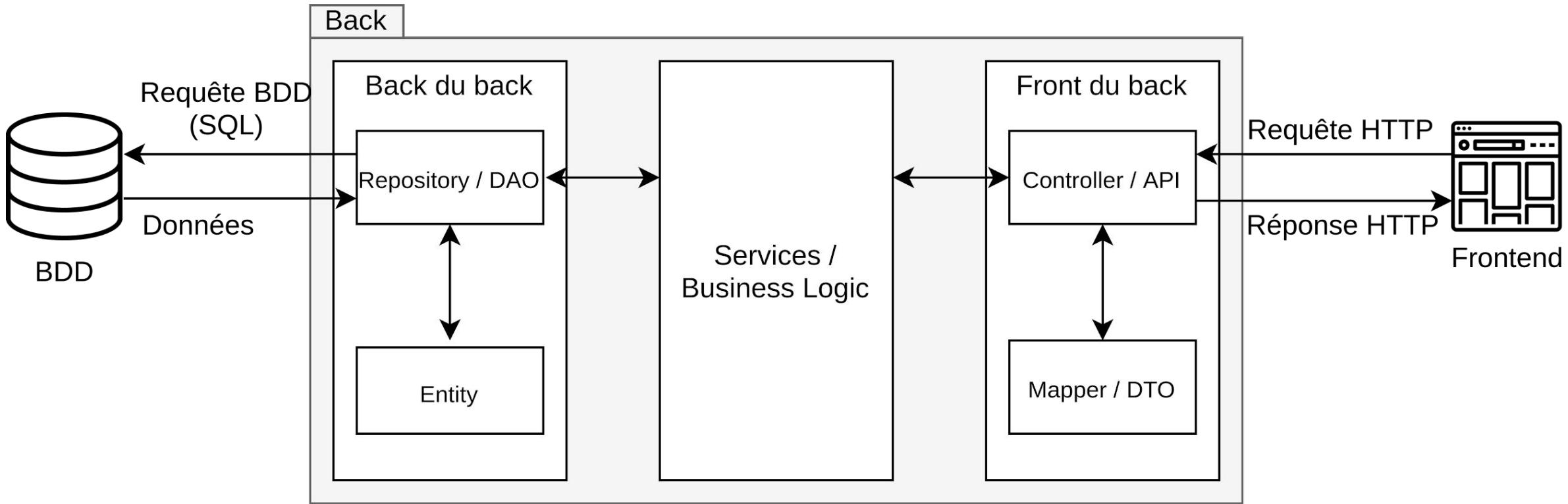


# Single Page Application (SPA)

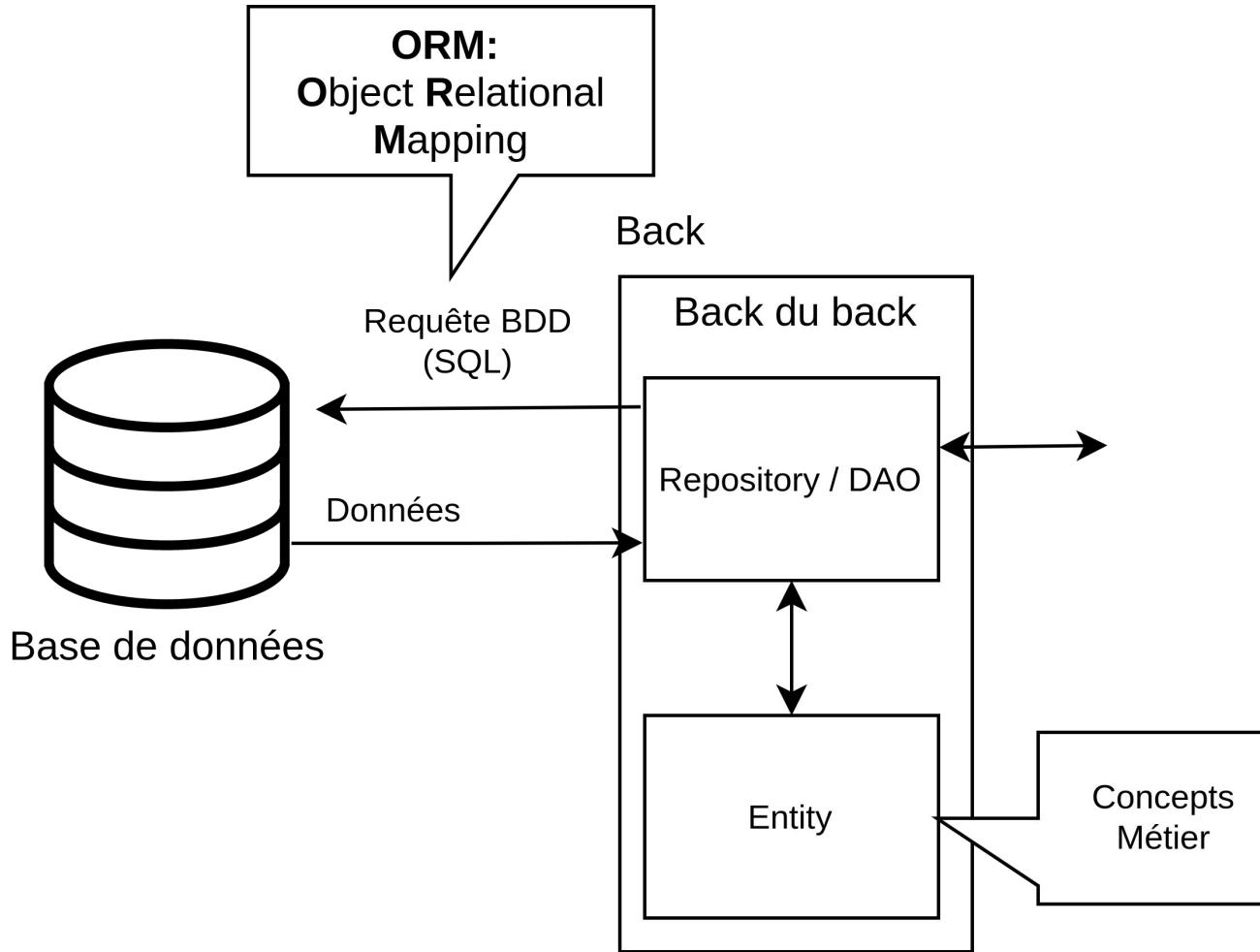


# Le backend en détail

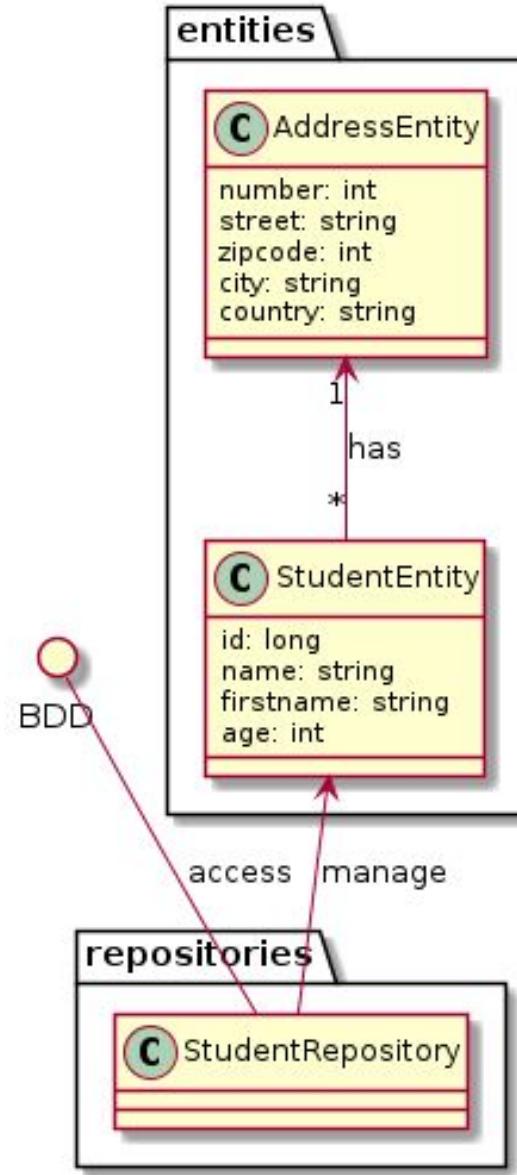
# Aperçu du backend



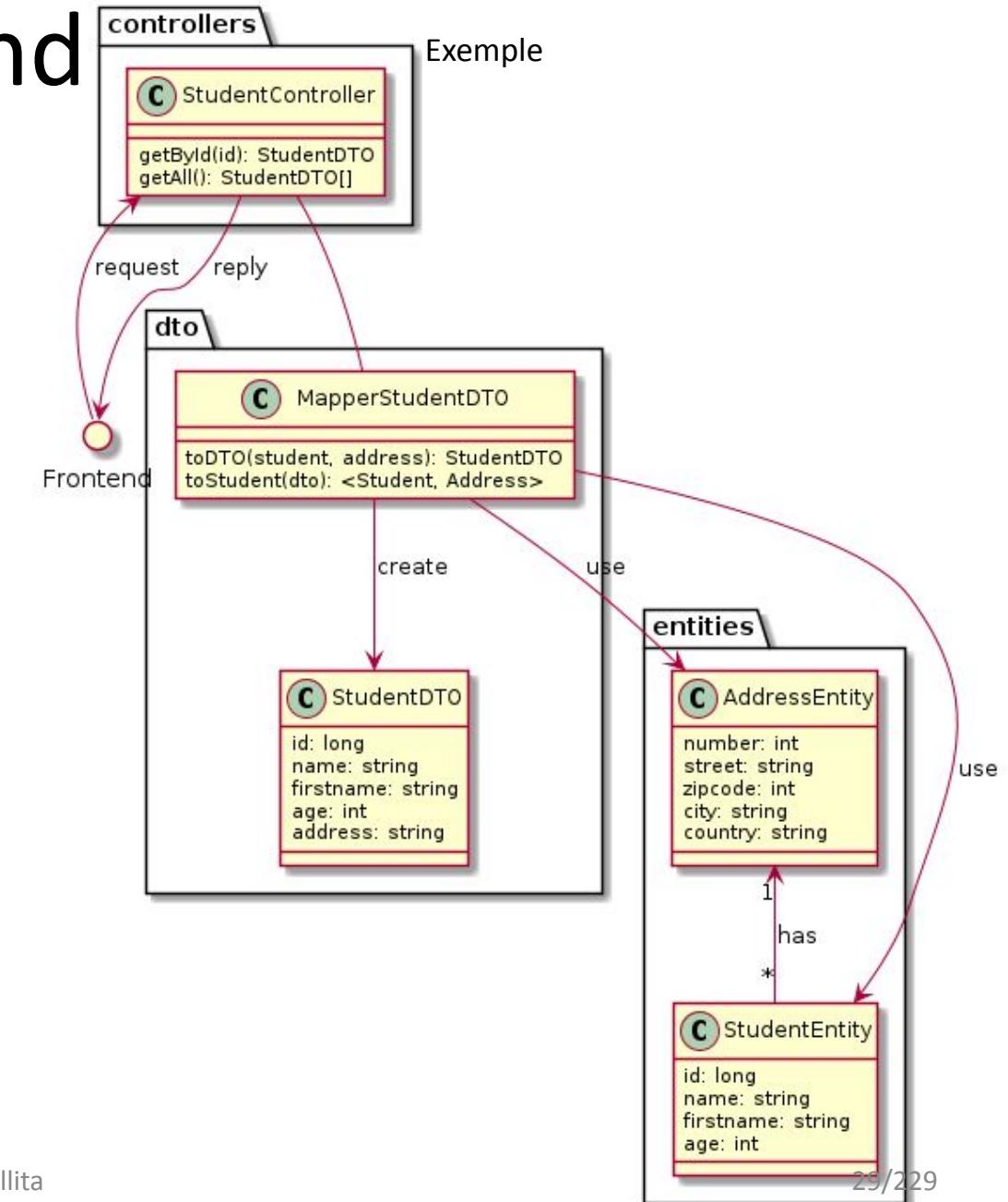
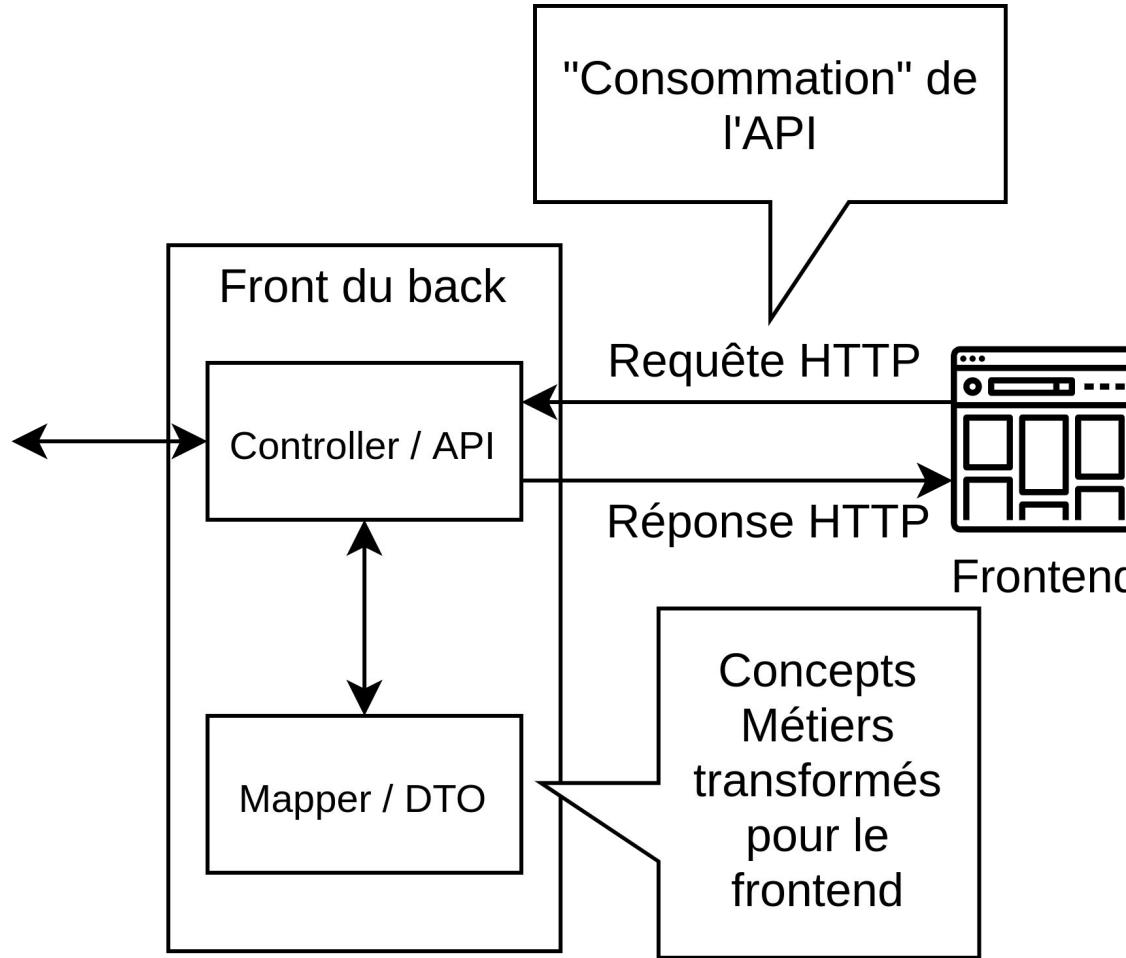
# Aperçu du back du backend



Exemple

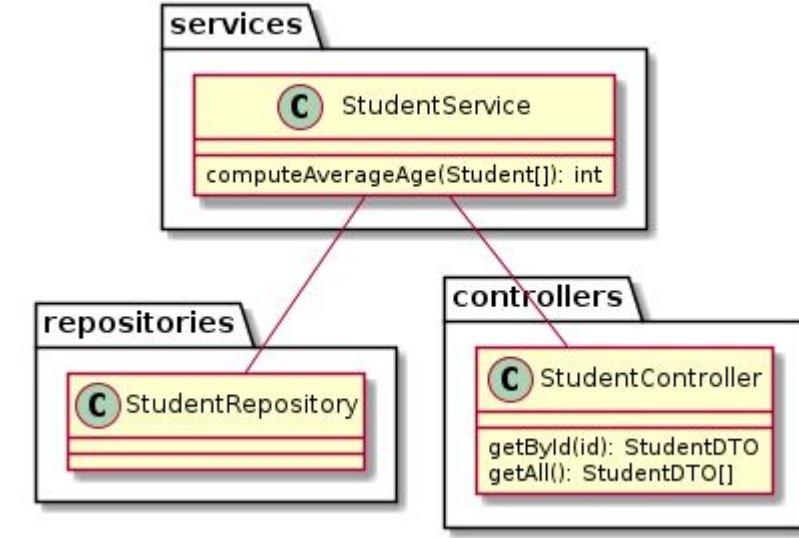
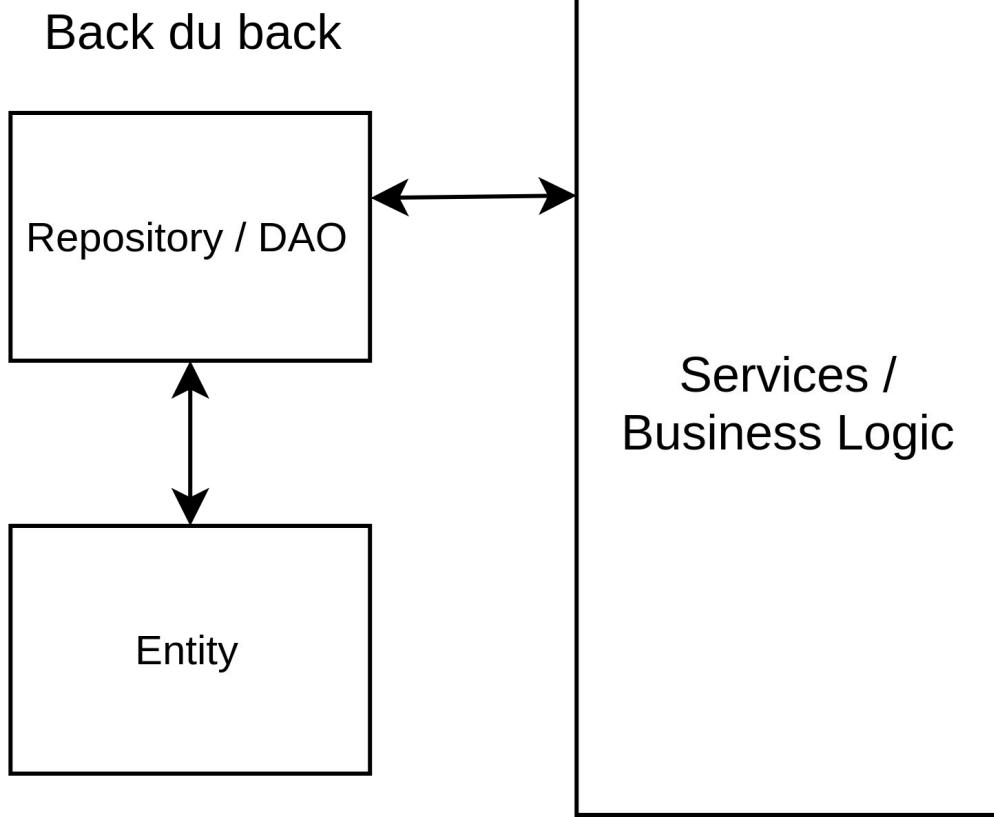


# Aperçu du front du backend



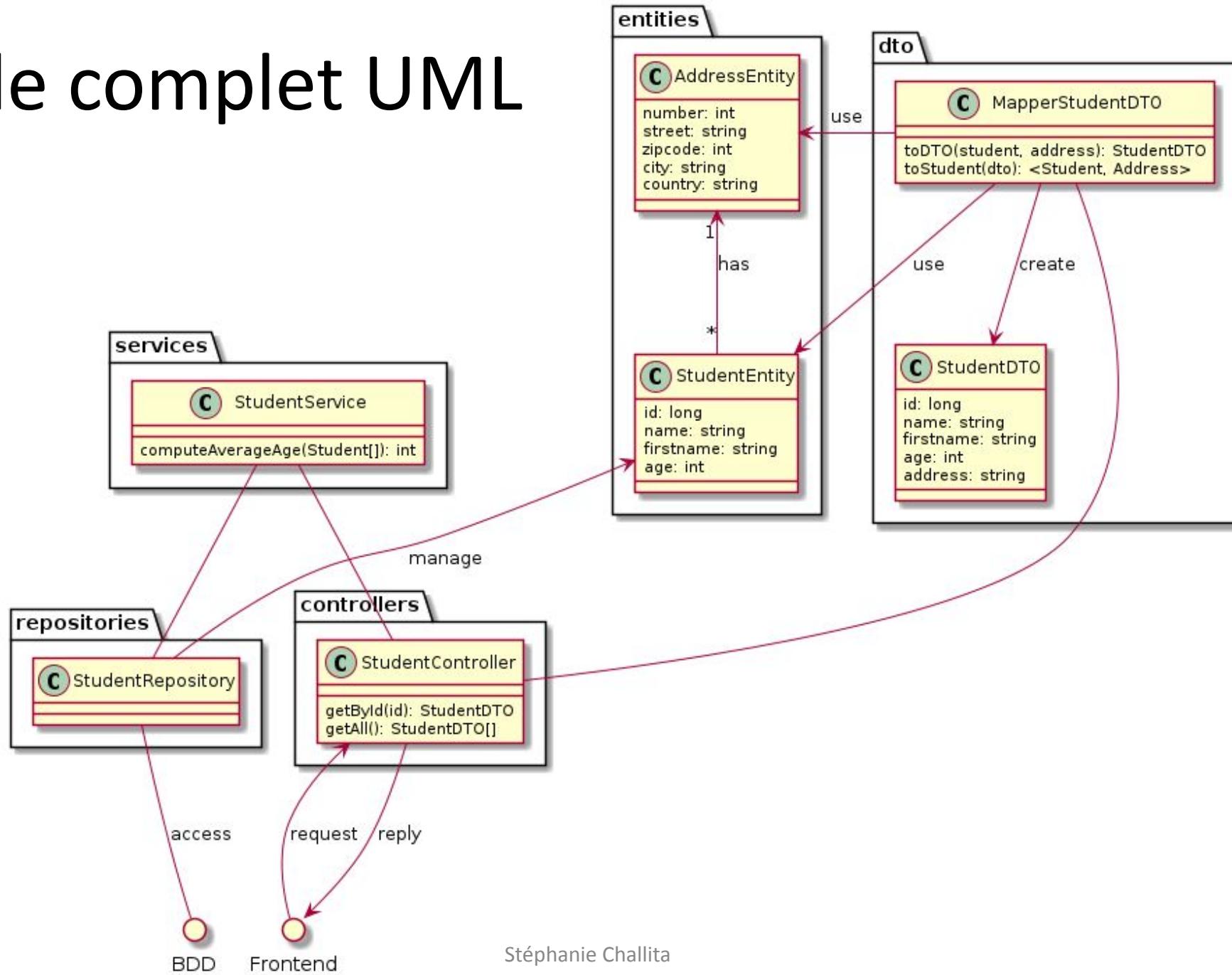
# Aperçu des services du backend

Exemple



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

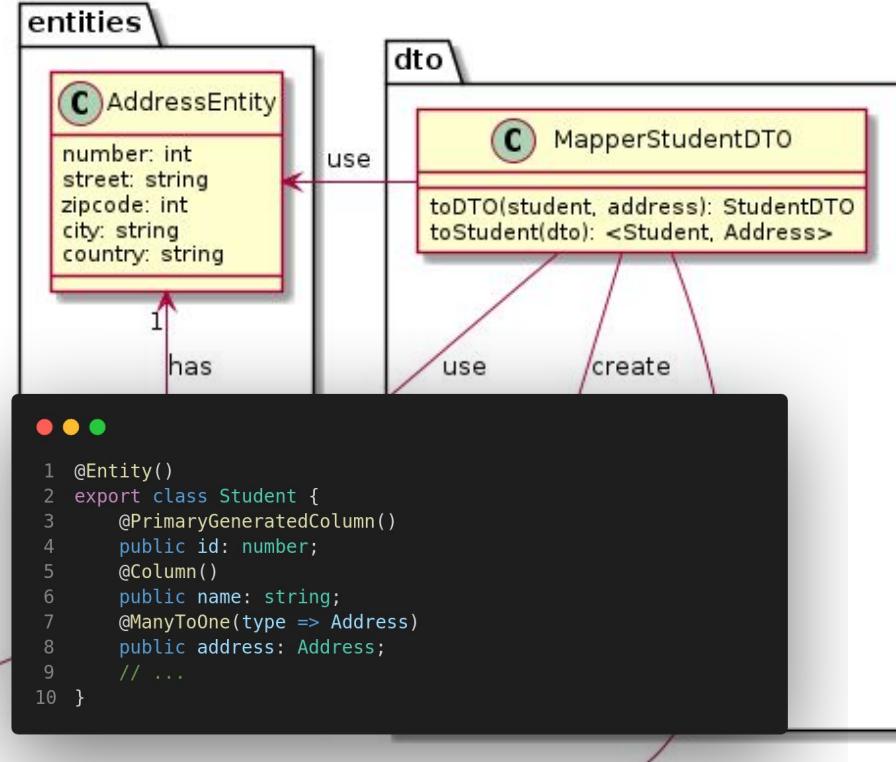
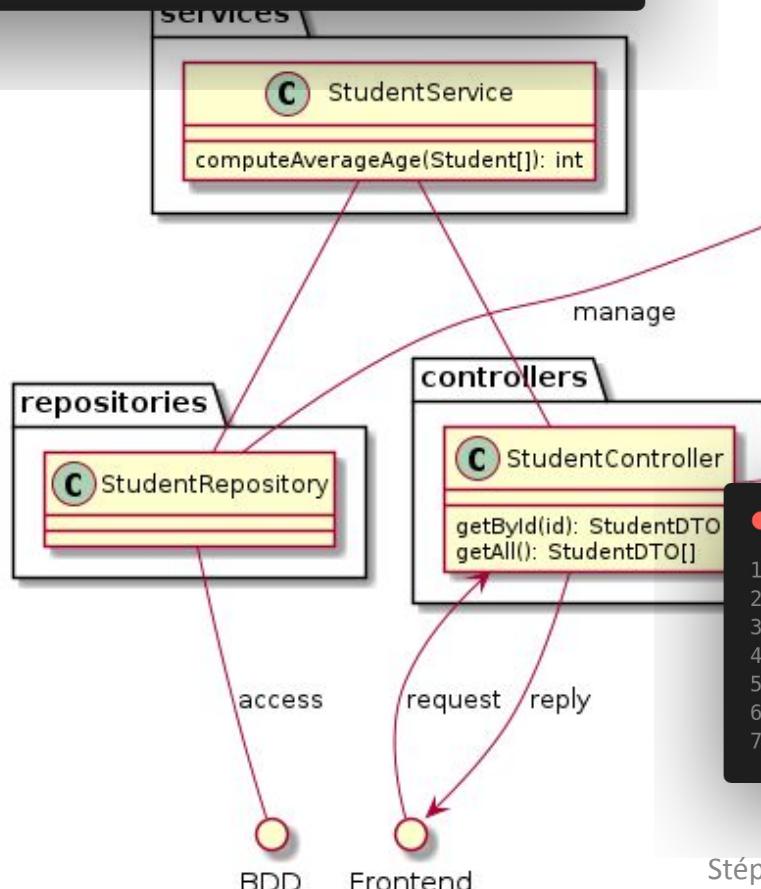
# Exemple complet UML



# Code snippets



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



# 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
- Les services renvoient les valeurs, après traitement aux contrôleurs qui se chargeront de construire la réponse adéquate

# 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 pour le web

- En JS/TS, le code est exécuté ligne par ligne (synchronement) par défaut
- Certaines opérations prennent du temps, par exemple :
  - Requêtes HTTP (fetch)
  - Lecture/écriture de fichiers
  - Accès à une base de données
- Si on attend que ces opérations finissent synchronement, le programme serait bloqué

**C'est là qu'intervient l'asynchronisme**

# Asynchronisme: Intro

- **Définition** : Permet à un programme de continuer à s'exécuter pendant qu'une tâche longue est en cours
- **Exemples** : appels API, lecture de fichiers, accès base de données
- **Avantage** : aucune attente inutile → meilleure réactivité et performance

# Les 3 approches principales

- Callback
- Promise
- `async / await`

# Callback

```
1  function fetchData(callback: (data: string) => void) {  
2      setTimeout(() => {  
3          callback("Données reçues !");  
4      }, 1000);  
5  }  
6  
7  console.log("Avant fetch");  
8  fetchData((data) => {  
9      console.log(data);  
10 });  
11 console.log("Après fetch");
```

- callback = fonction passée en argument
- Exécutée après que l'opération est terminée
- Problème : code vite illisible si callbacks imbriqués (“callback hell”)

# Promise

```
1  function fetchData(): Promise<string> {
2      return new Promise((resolve) => {
3          setTimeout(() => {
4              resolve("Données reçues !");
5          }, 1000);
6      });
7
8
9  console.log("Avant fetch");
10 fetchData().then((data) => {
11     console.log(data);
12 });
13 console.log("Après fetch");
```

- Une Promise représente une valeur qui sera disponible plus tard
- resolve() fournit le résultat une fois prêt
- On utilise .then() pour récupérer le résultat
- Plus lisible que les callbacks, mais peut encore s'enchaîner lourdement

# async / await

```
1  function fetchData(): Promise<string> {
2      return new Promise((resolve) => {
3          setTimeout(() => {
4              resolve("Données reçues !");
5          }, 1000);
6      });
7
8
9  async function main() {
10    console.log("Avant fetch");
11    const data = await fetchData();
12    console.log(data);
13    console.log("Après fetch");
14
15
16  main();
```

- `async` rend une fonction asynchrone
- `await` attend la résolution de la `Promise`
- Code beaucoup plus lisible et proche du synchrone
- Recommandé pour la plupart des cas modernes

# 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
- Recherche

# 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

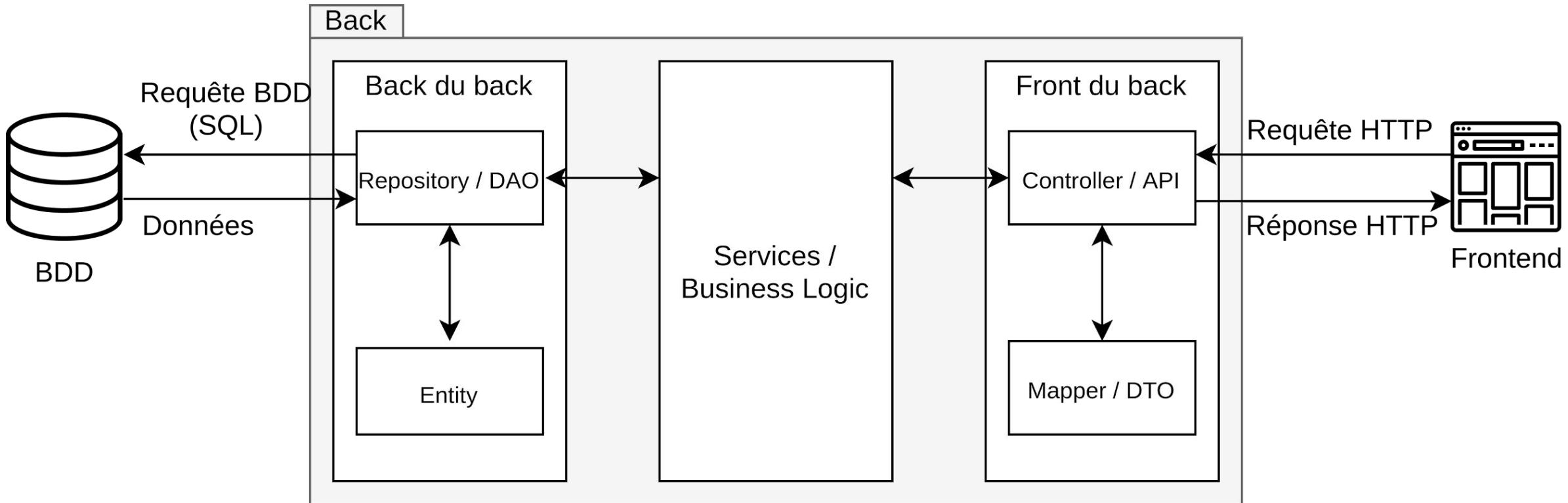
# POO donc pourquoi pas un SGBD objet ?

- 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)**

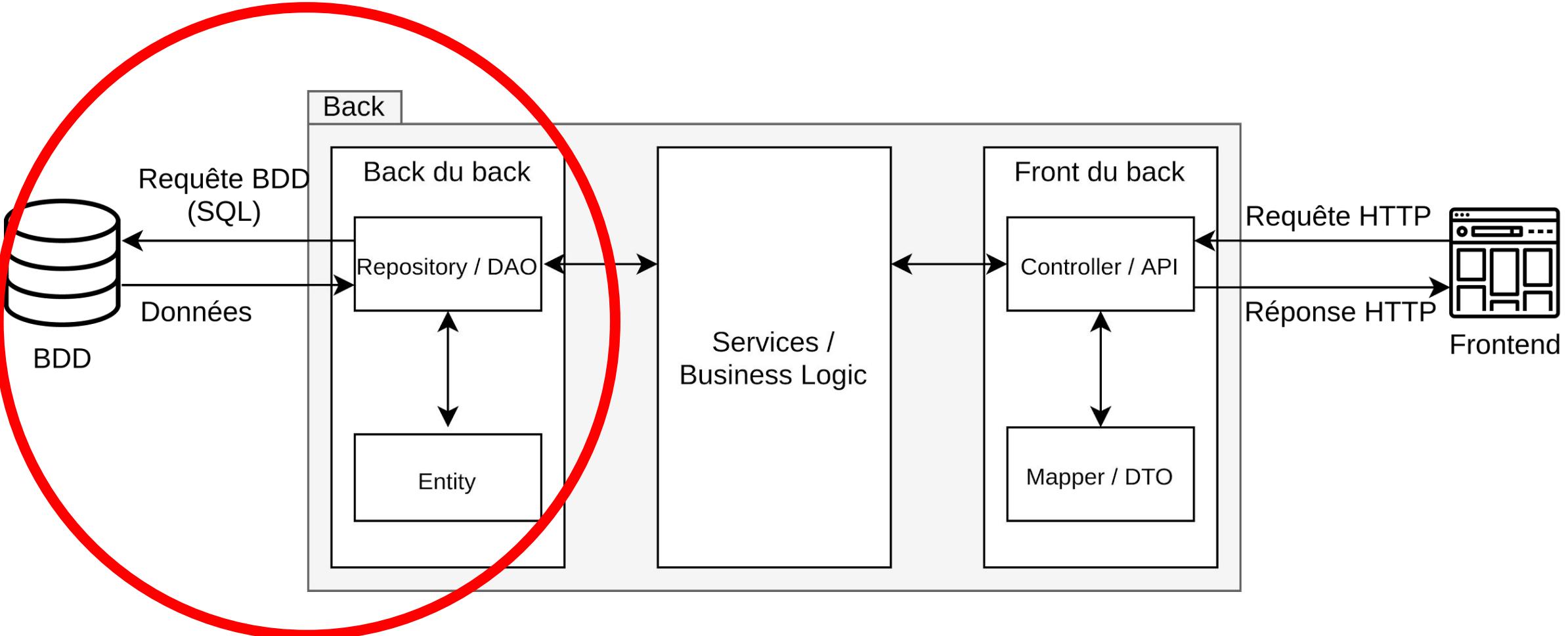
# Objet -> BDD, quelles sont les difficultés ?

- 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 ?



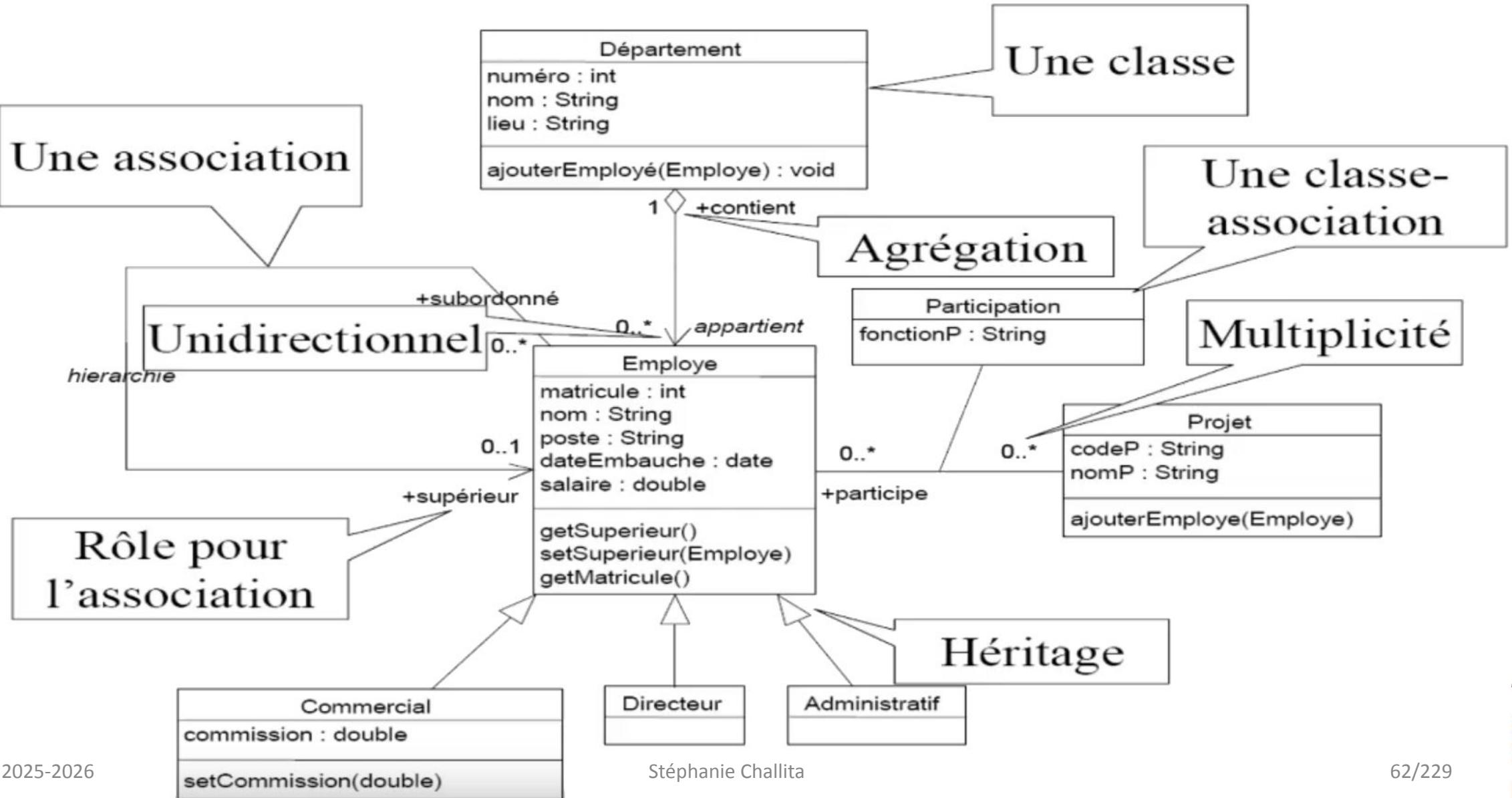
# Contenu de cette partie du cours

- 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

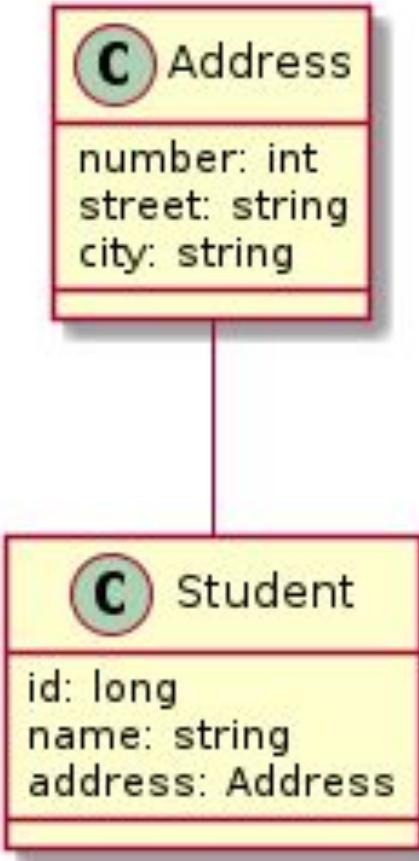


Table Student

<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 }
```

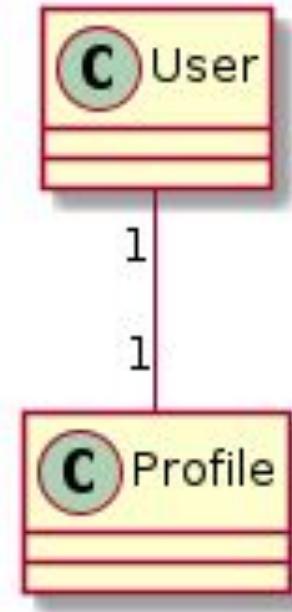
## Java (Hibernate)

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

# Traduction des associations

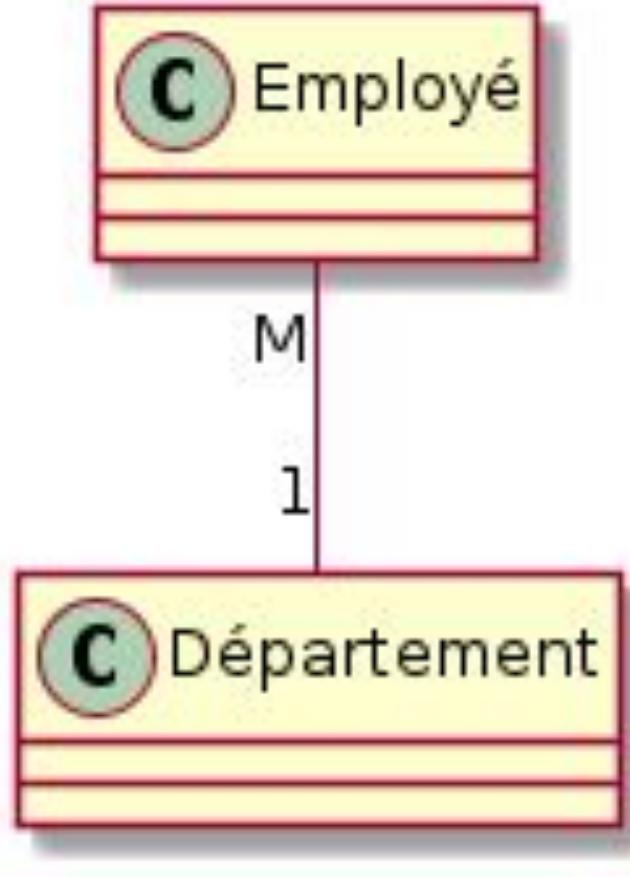
- 3 associations à distinguer :
  2. Un attribut d'instance représentant “l'autre” objet (1:1, M:1)
  3. Un attribut d'instance de type collection représentant tous les autres objets (1:M, M:N)
  4. Une classe association (M:N)

# Exemple d'association 1 (1:1)



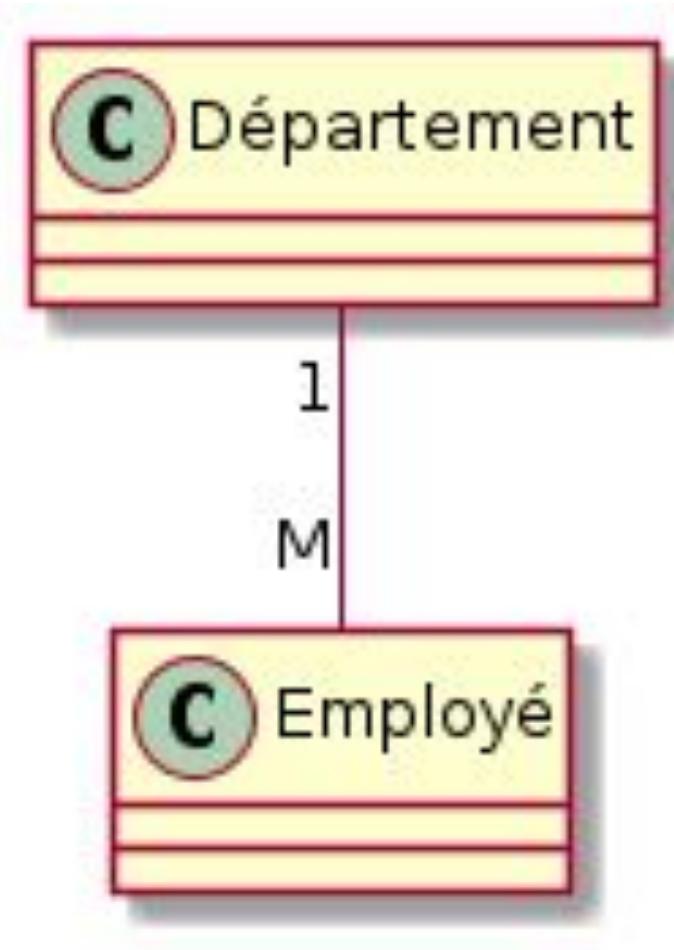
```
1 export class User {  
2     profile: Profile  
3 }  
4 export class Profile {  
5     user: User  
6 }
```

# Exemple d'association 2 (M:1)



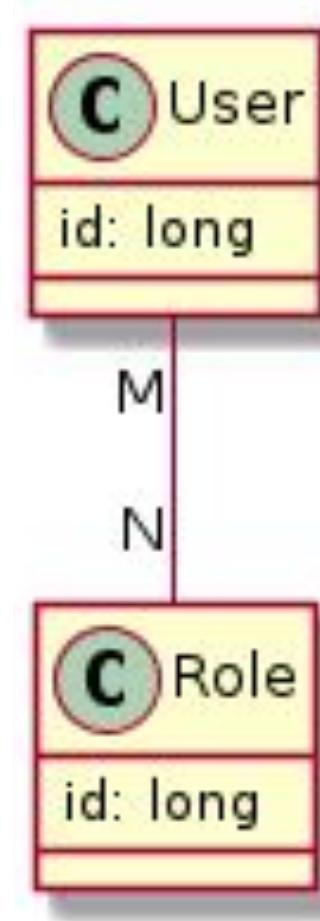
```
1 export class Département {  
2     employés: Employé[];  
3 }  
4 export class Employé {  
5     département: Département;  
6 }
```

# Exemple d'association 2 (1:M)



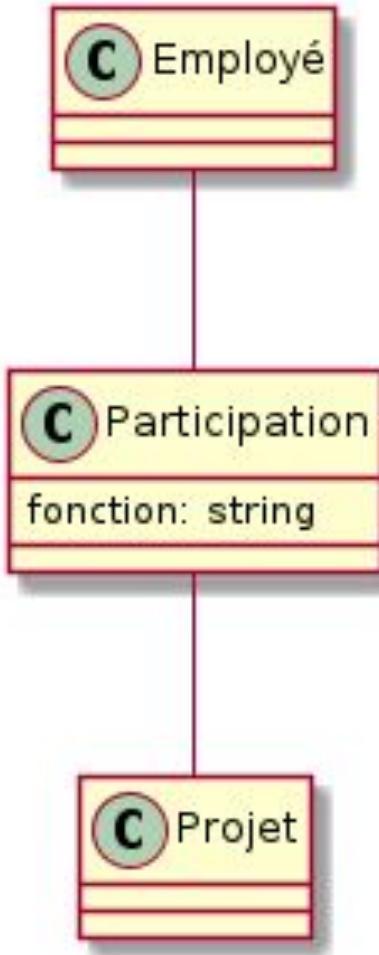
```
1 export class Département {  
2     employés: Employé[];  
3 }  
4 export class Employé {  
5     département: Département;  
6 }
```

# 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)

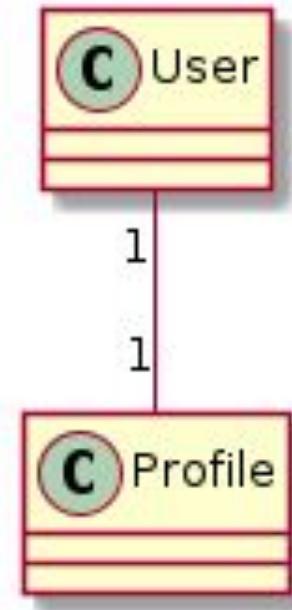


Table User

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

Table Profile

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

# Exemple d'association 2 (1:M)

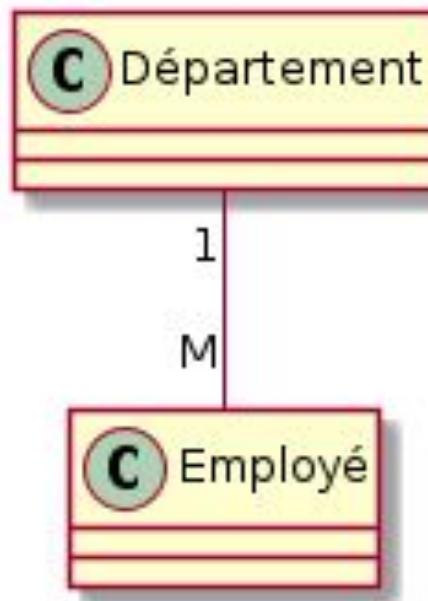


Table Département

<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 3 (M:N)

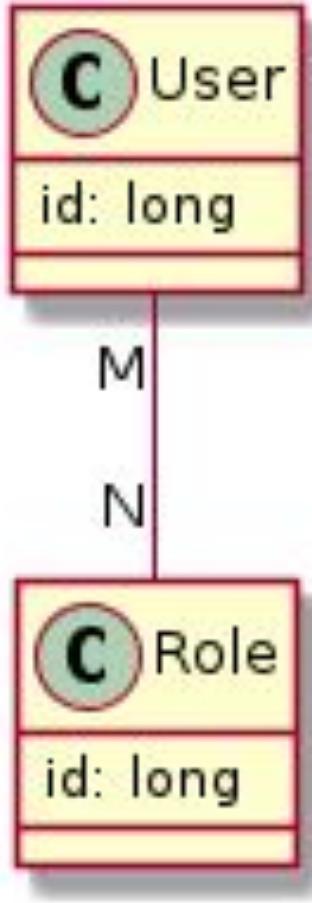


Table User

<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

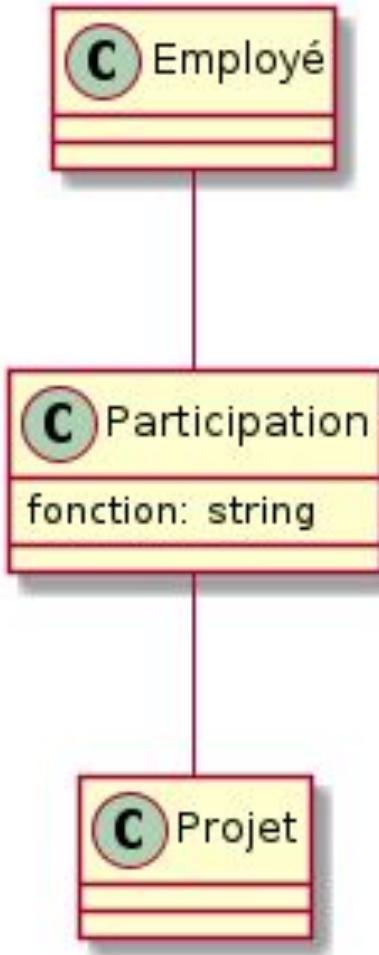


Table Employé

<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

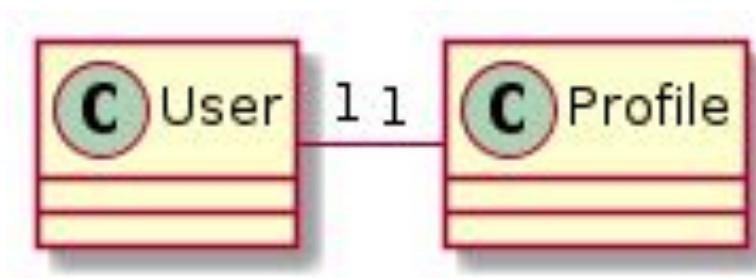
<b>id</b>	...
1	...
2	...

# 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

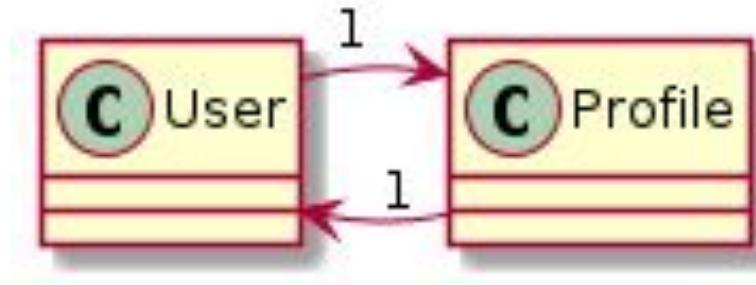


User

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

Profile

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



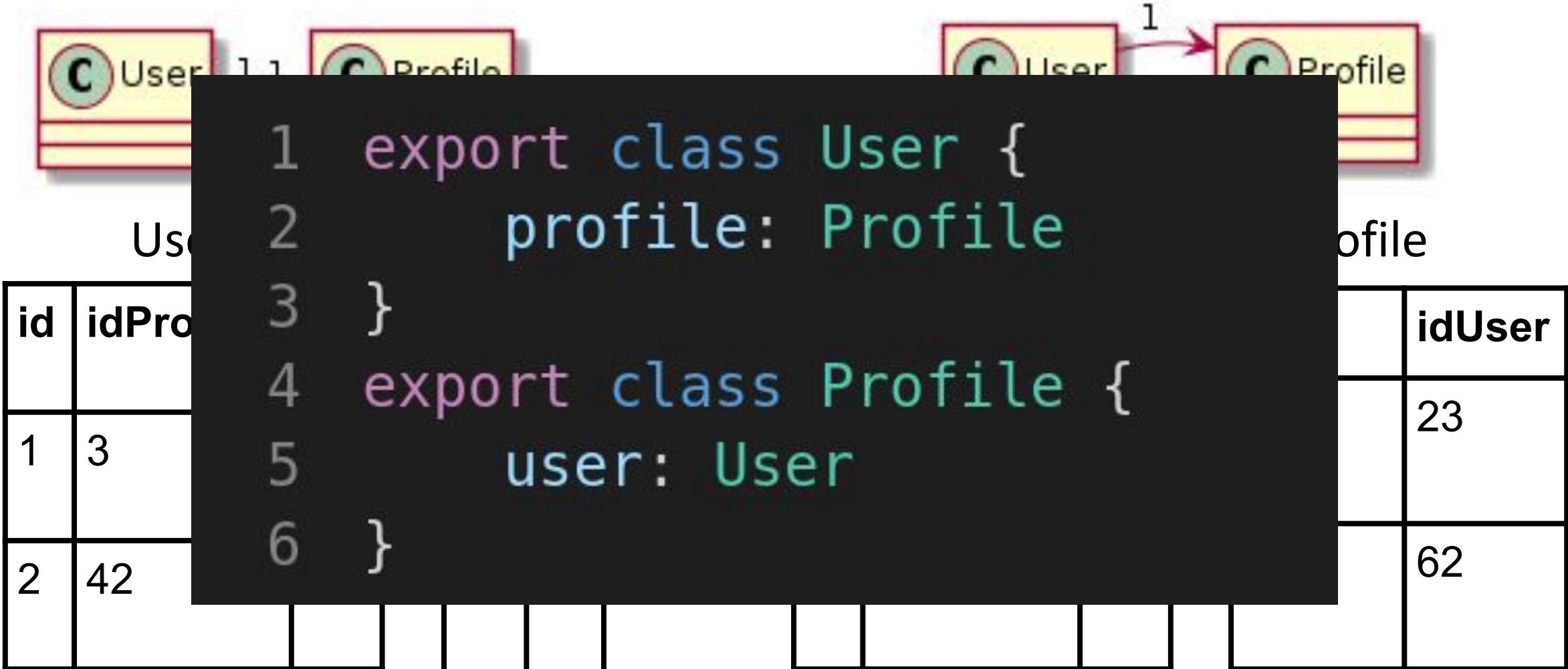
User

<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



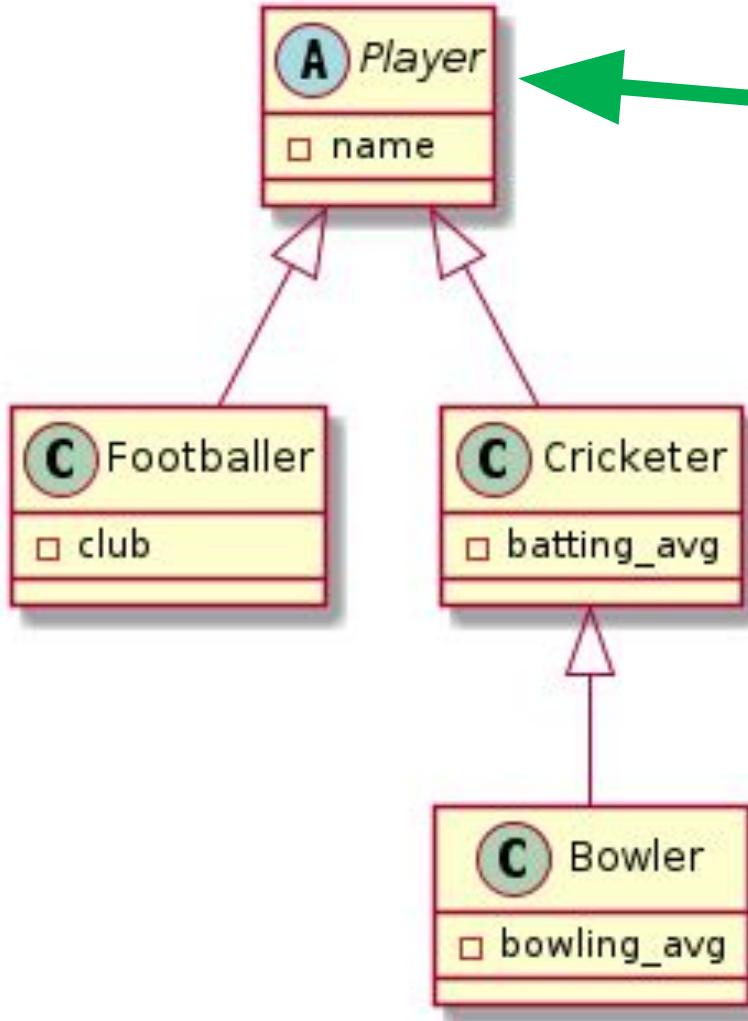
# Objet dépendant

- 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 un produit  
Suppression de la ligne → on garde le produit intact
- Cas typique où il faut donner ces informations au framework

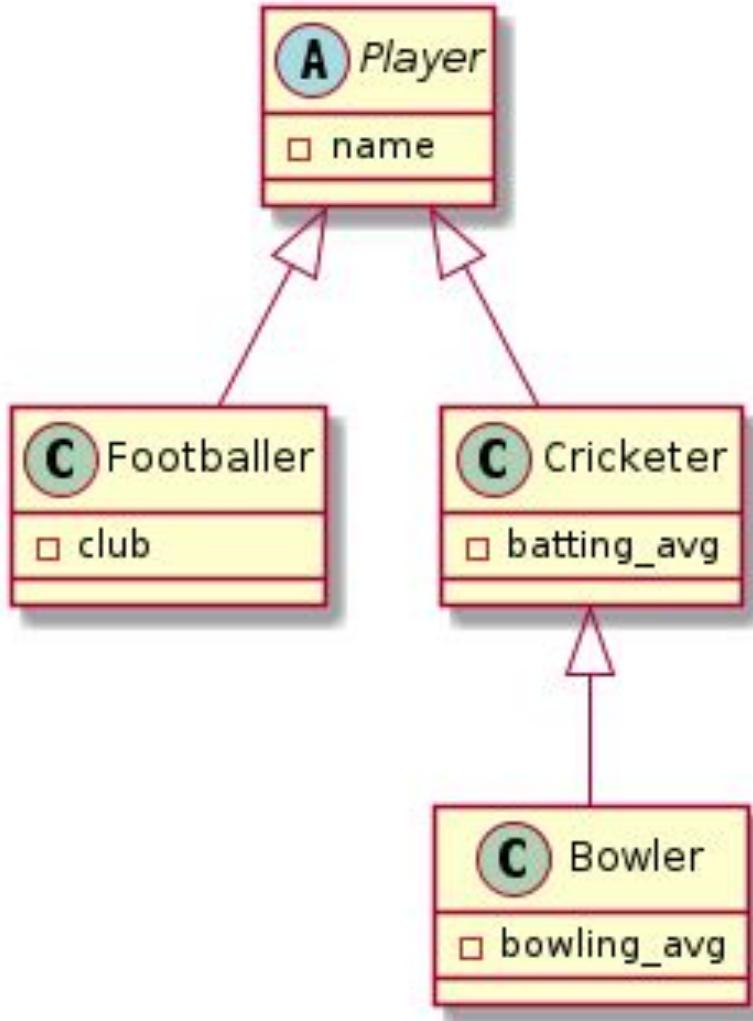
# Traduction de l'héritage



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	...

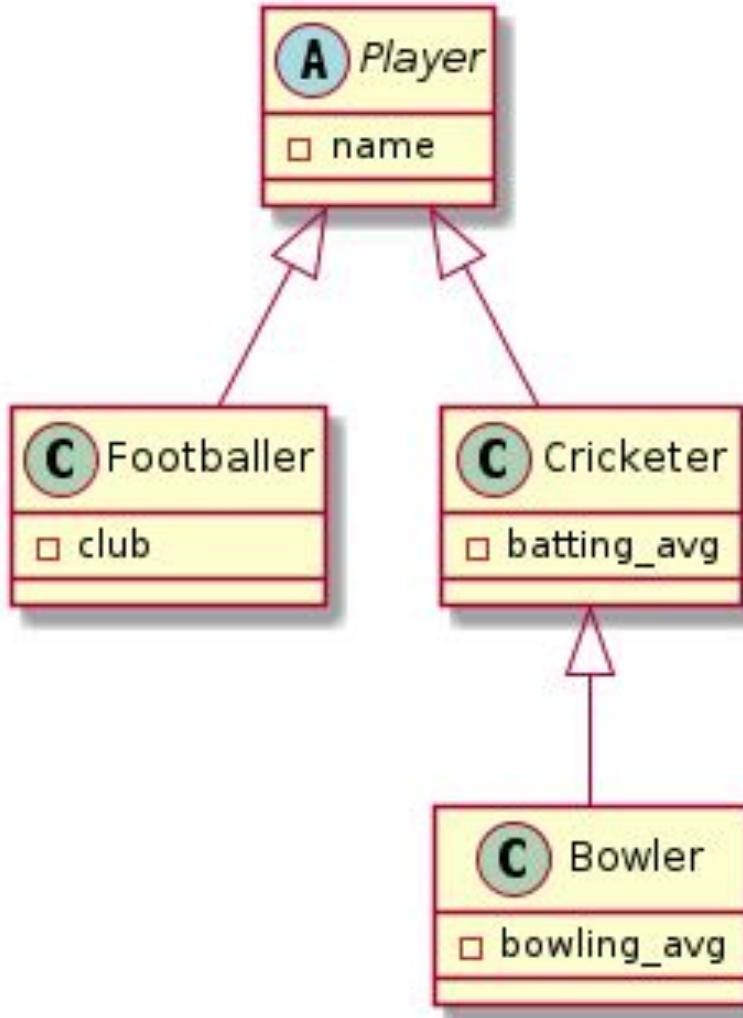
# Avantages

- 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	...

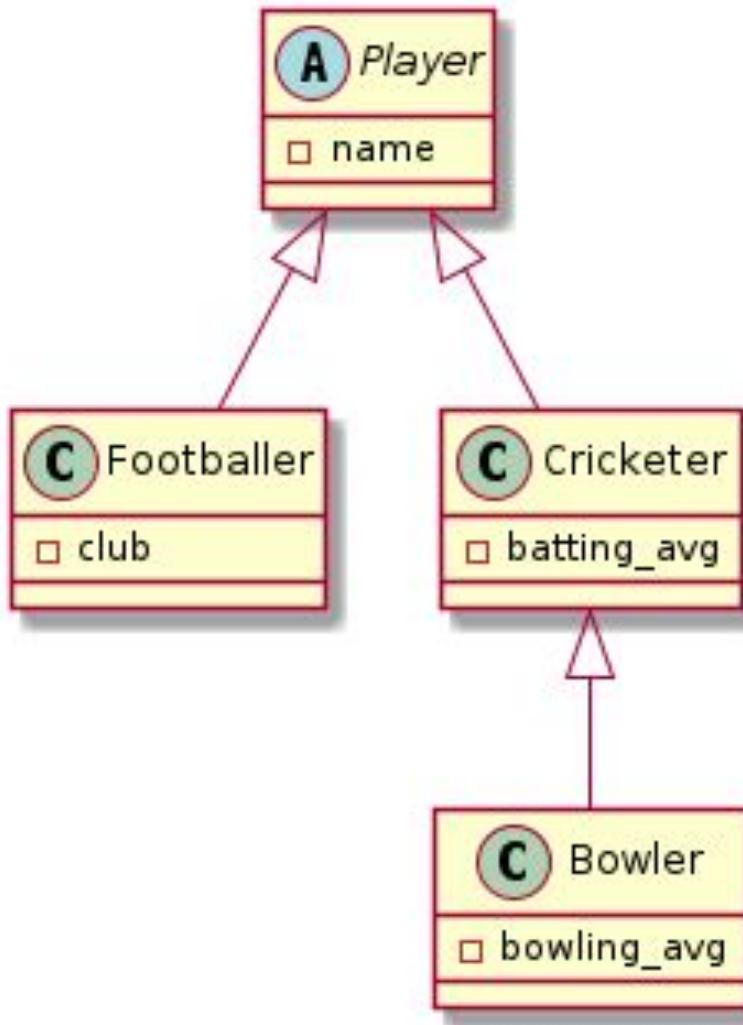
# Avantages

- 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”

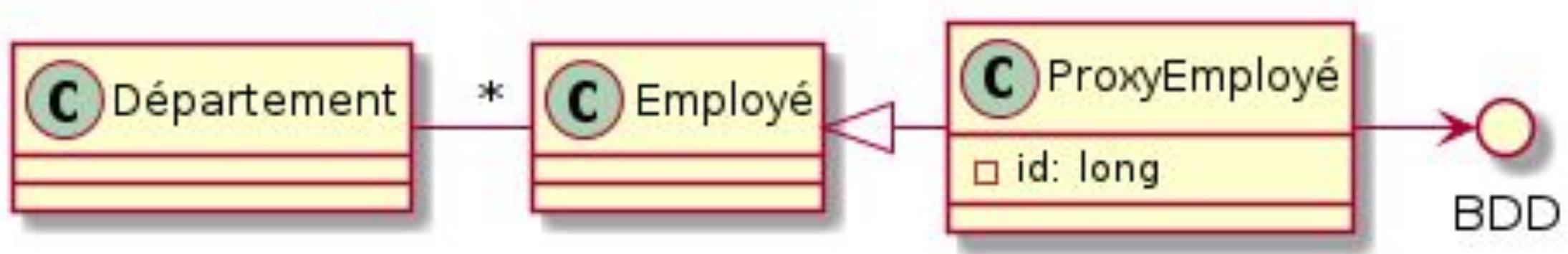
# Exemple

- Recherche dans la BDD d'une facture
- Création 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

```
@Entity()  
export class Student {  
    @PrimaryGeneratedColumn()  
    public id: number;  
    @Column()  
    public name: string;  
    @ManyToOne(() => Address)  
    public address: Address;  
    // ...  
}
```

- 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

```
export class Name {  
    @Column()  
    public firstname: string;  
    @Column()  
    public lastname: string;  
}  
@Entity()  
export class User {  
    @Column(() => Name)  
    public name: Name;  
    // ...  
}  
@Entity()  
export class Employee {  
    @Column(() => Name)  
    public name: Name;  
    // ...  
}
```

Table USER

firstname	varchar
lastname	varchar
...	...

Table EMPLOYEE

firstname	varchar
lastname	varchar
...	...

# Associations : OneToOne

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

Table PROFILE

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

Table USER

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

# Associations : OneToOne

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

## Table PROFILE

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

## Table USER

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

# Associations : OneToOne

```
1 @Entity()  
2 export class Profile {  
3     @OneToOne(() => User)  
4     @JoinColumn()  
5     user: User  
6     //...  
7 }  
8 @Entity()  
9 export class User {  
10    @OneToOne(() => Profile)  
11    @JoinColumn()  
12    profile: Profile  
13    //...  
14 }
```

Table PROFILE

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

Table USER

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

# Associations : ManyToOne

```
@Entity()  
export class User {  
    // ...  
}  
  
@Entity()  
export class Photo {  
    @ManyToOne(() => User)  
    public user: User  
    // ...  
}
```

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 : OneToMany

```
@Entity()  
export class User {  
    @OneToMany(  
        () => Photo,  
        photo => photo.user  
    )  
    public photos: Photo[];  
    // ...  
}  
  
@Entity()  
export class Photo {  
    @ManyToOne(  
        () => User,  
        user => user.photos  
    )  
    public user: User  
    // ...  
}
```

Table USER

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

Table PHOTO

id	int	Primary Key
userId	int	Foreign Key
...	...	...

# 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>
...	...	...

# Clé composite à partir de 2 clés étrangères

```
@Entity()  
export class Role {  
  
    @ManyToOne(() => User, {eager: true})  
    @PrimaryColumn()  
    @JoinColumn()  
    public user: User;  
  
    @ManyToOne(() => Association, {eager: true})  
    @PrimaryColumn()  
    @JoinColumn()  
    public association: Association;  
  
    @Column()  
    public name: string;  
}
```

Table ROLE

idUser	int	Primary Key, Foreign Key
idAssociation	int	Primary Key, Foreign Key
name	string	

# 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	...

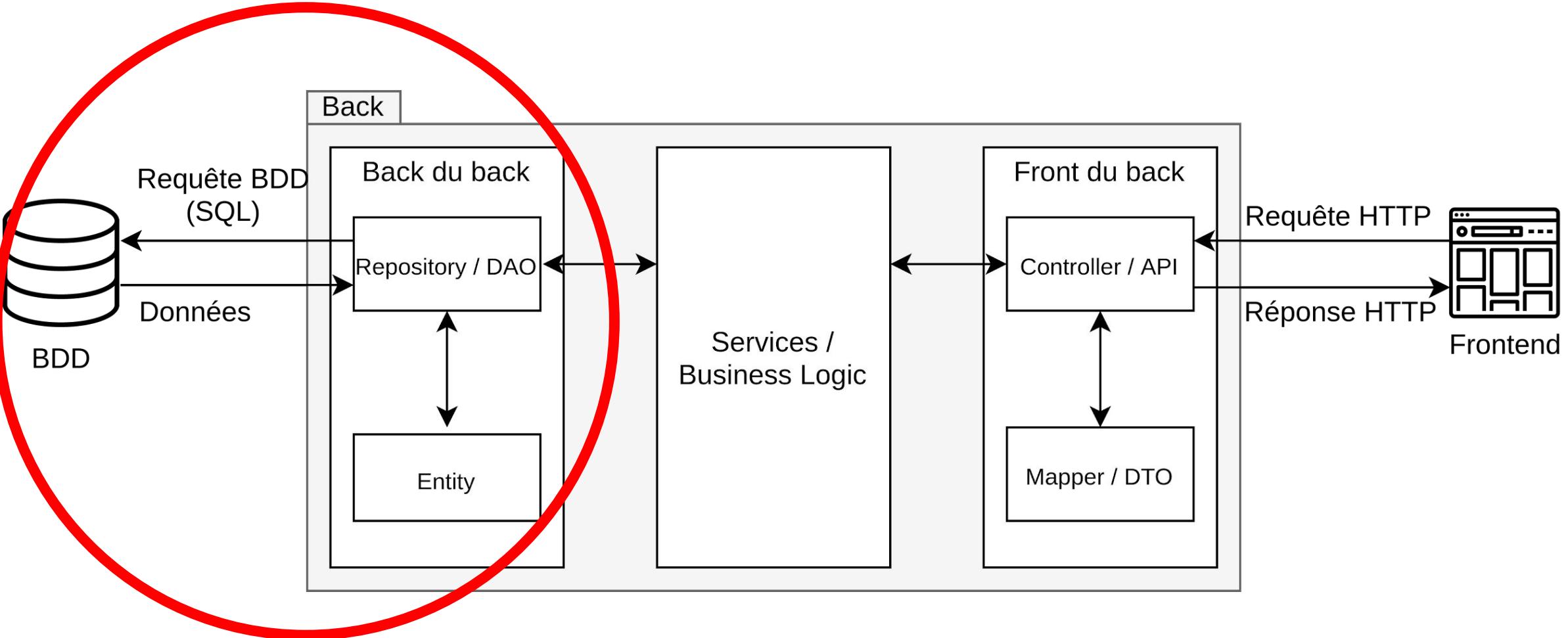
```
export abstract class Player {  
    @PrimaryGeneratedColumn()  
    public id: number;  
    @Column()  
    public name: string;  
}  
@Entity()  
export class Footballer extends Player {  
    @Column()  
    public club: string;  
}  
@Entity()  
export class Cricketer extends Player {  
    @Column()  
    public batting_avg: number;  
}  
@Entity()  
export class Bowler extends Cricketer {  
    @Column()  
    public bowling_avg: number;  
}
```

# Héritage avec TypeORM

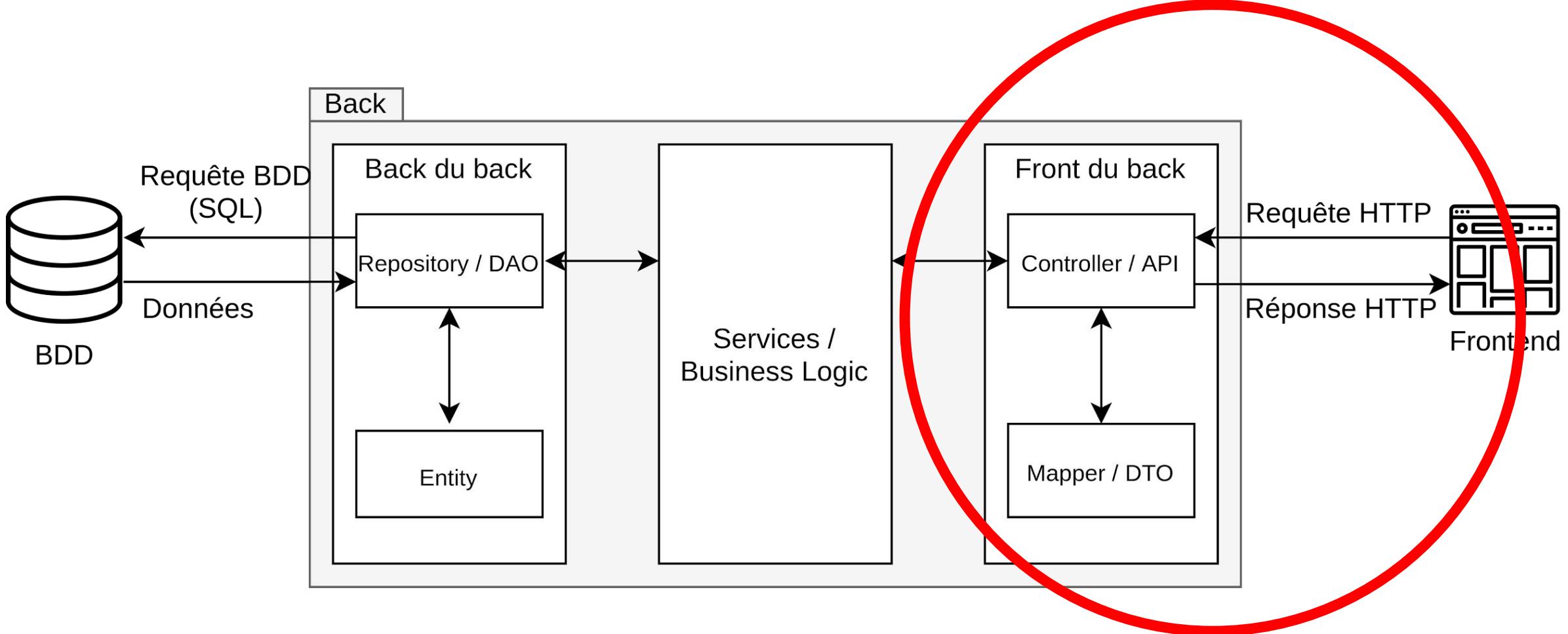
i d	type	no m	club	batti ng	bowli ng
1	Footbal ler	Bob ba	Stade Renn ais	null	null
2	Bowler	Jan go	null	1.2	3.4

```
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

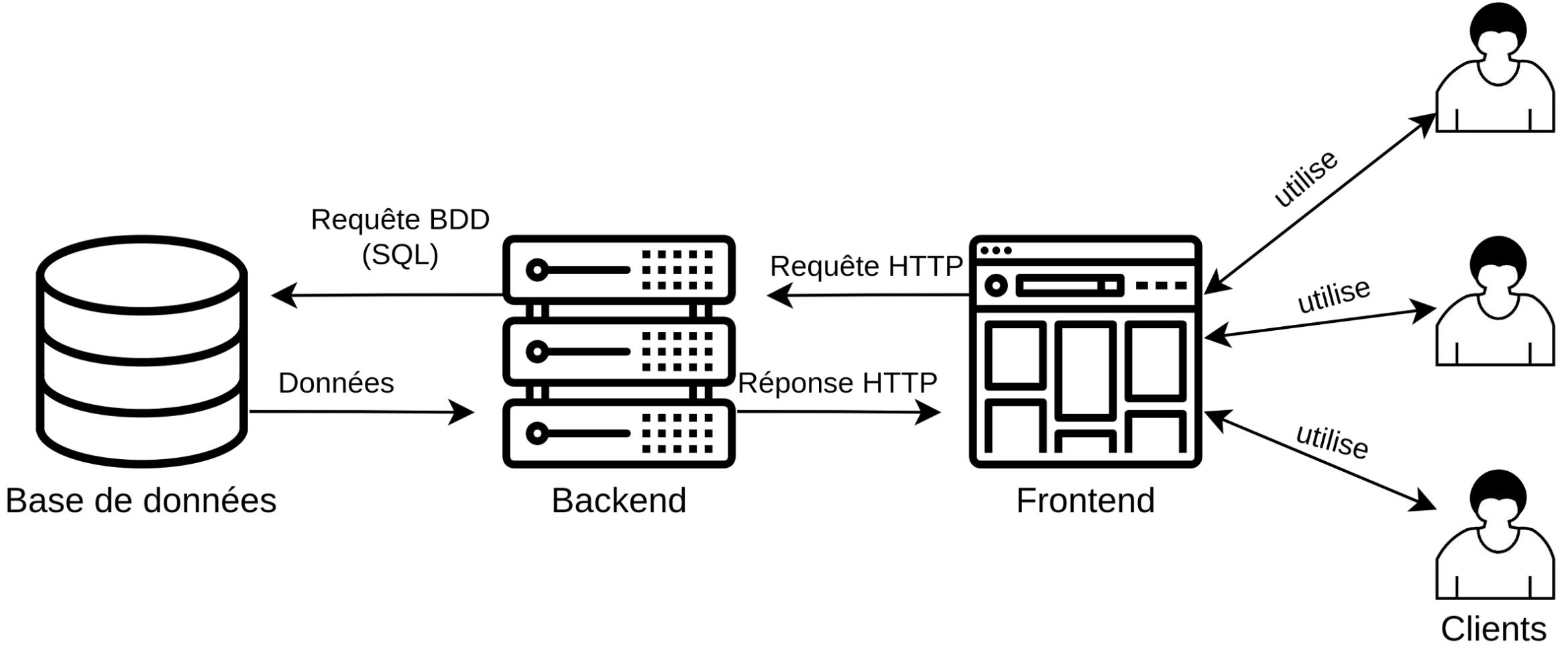
# Intro à SOA

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

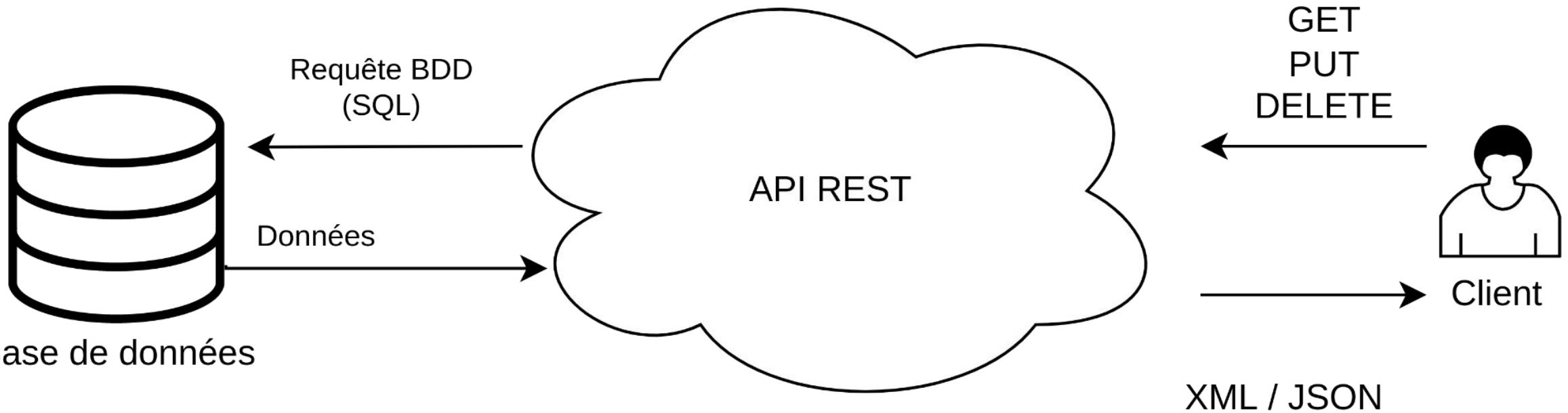
# Intro à REST

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

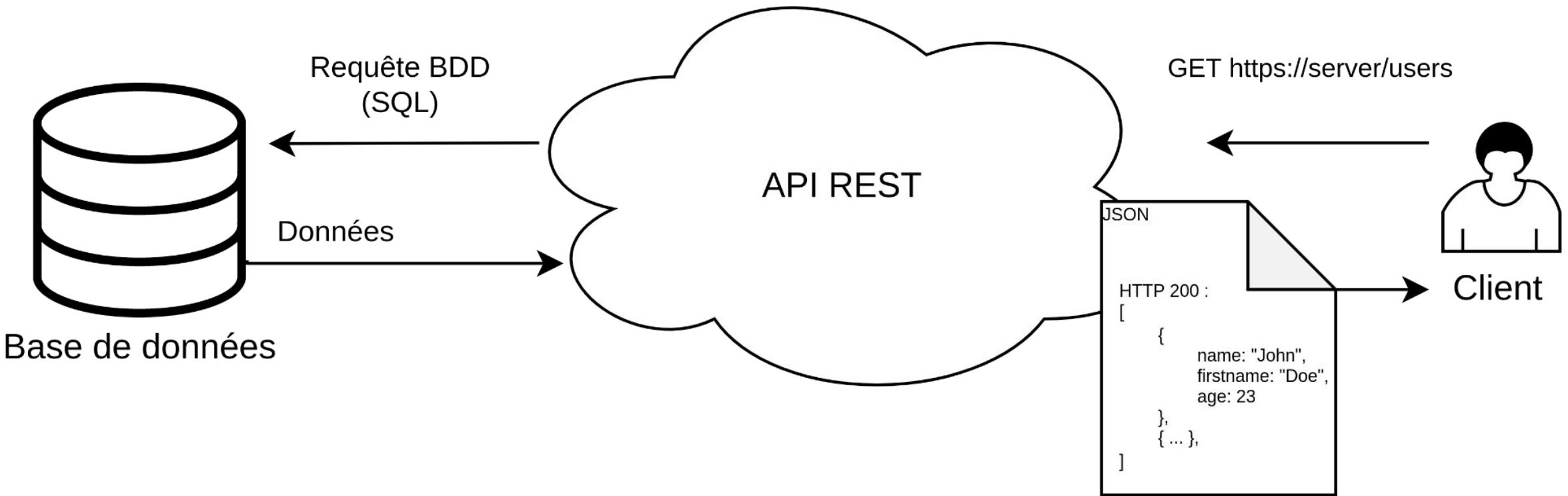
# Architecture REST



# 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

# CRUD : Créer, Lire, Mettre à jour, Supprimer

- Implémenter pour toutes les ressources, un CRUD

Create	Read	Update	Delete
Ajouter une ressource	Récupérer une ou plusieurs ressource(s)	Modifier une ressource	Supprimer une ressource
Créer un nouvel utilisateur	Récupérer la liste des utilisateurs	Mettre à jour le nom de l'utilisateur 42	Supprimer l'utilisateur 42

# Associer CRUD aux verbes HTTP

- Implémenter pour toutes les ressources, un CRUD

Create	Read	Update	Delete
POST	GET	PUT ou PATCH	DELETE
POST /users pour créer un user	GET /users ou GET /users/42	PUT /users/42 pour modifier	DELETE /users/42 pour supprimer

# Créer une ressource avec POST

- **But** : Créer une nouvelle ressource
- **Comportement** : Envoie des données au serveur (payload)
- **Effet** : Le serveur génère une nouvelle ressource
- **Idempotence** : Non idempotent : plusieurs requêtes = plusieurs créations
- **Exemple** : POST /users → création d'un nouvel utilisateur

# Lire une ressource ou des ressources avec GET

- **But** : Lire ou récupérer une ressource
- **Comportement** : Envoie une requête sans corps (body)
- **Effet** : Le serveur retourne les données de la ressource
- **Idempotence** :  Idempotent : plusieurs requêtes = même réponse
- **Exemple** :

GET /users → liste complète

GET /users/42 → utilisateur 42

# Remplacer une ressource avec PUT

- **But** : Mettre à jour complètement une ressource existante
- **Comportement** : Envoie une représentation complète dans le corps
- **Effet** : Remplace la ressource cible par la nouvelle version
- **Idempotence** : Idempotent : plusieurs requêtes identiques ont le même effet
- **Exemple** : PUT /users/42 avec toutes les données utilisateur

# Modifier partiellement une ressource avec PATCH

- **But** : Mettre à jour partiellement une ressource
- **Comportement** : Envoie uniquement les champs à modifier dans le corps
- **Effet** : Le serveur applique les modifications sur la ressource existante
- **Idempotence** : Idempotent (en principe) : mêmes requêtes = même état final
- **Exemple** : PATCH /users/42 avec seulement l'email modifié

# Supprimer une ressource avec DELETE

- **But** : Supprimer une ressource existante
- **Comportement** : Envoie une requête sans corps
- **Effet** : Le serveur supprime la ressource ciblée
- **Idempotence** : Idempotent : plusieurs suppressions ont le même effet
- **Exemple** : `DELETE /users/42` supprime l'utilisateur 42

# Les codes HTTP : communiquer le résultat d'une requête

- Chaque requête HTTP reçoit une réponse avec un code
- Ce code indique le résultat : succès, erreur, redirection...
- Le code guide le client dans le traitement de la réponse
- Les codes sont classés en 5 catégories (classes) :
  - 1xx : Informations
  - 2xx : succès
  - 3xx : Redirections
  - 4xx : Erreurs client
  - 5xx : Erreurs serveur

# Codes HTTP 1xx : réponses intermédiaires

100	Continue	Le client peut poursuivre la requête
101	Switching Protocols	Le serveur accepte de changer de protocole
103	Early Hints	Préchargement possible avant la réponse finale (HTTP/2)

# Codes HTTP 2xx : réussite des requêtes

200	Ok	Réponse standard réussie avec corps
201	Created	Ressource créée avec succès
204	No content	Requête réussie sans corps en réponse

# Codes HTTP 3xx : redirections

301	Moved Permanently	Ressource déplacée définitivement vers une nouvelle URL
302	Found	Ressource temporairement disponible ailleurs
303	See other	Redirection vers une autre URL, souvent après POST
304	Not Modified	La ressource n'a pas changé, utiliser la version en cache

# Codes HTTP 4xx : erreurs côté client

400	Bad Request	Requête mal formée ou invalide
401	Unauthorized	Authentification requise ou échouée
403	Forbidden	Accès interdit, même si authentifié
404	Not Found	Ressource non trouvée

# Codes HTTP 5xx : erreurs côté serveur

500	Internal Server Error	Erreur interne inattendue du serveur
502	Bad Gateway	Mauvaise réponse d'un serveur intermédiaire
503	Service Unavailable	Service temporairement indisponible
504	Gateway Timeout	Timeout lors de la communication avec un serveur

# Les codes HTTP clés à retenir

1xx	100, 101, 103	Informations (réponses intermédiaires)
2xx	200, 201, 204	Succès
3xx	301, 302, 303, 304	Redirections
4xx	400, 401, 403, 404	Erreurs client
5xx	500, 502, 503, 504	Erreurs serveur

# 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", ... ,
 },
 ...
 ]
```

# Une API REST, c'est bien... mais le client est livré à lui-même

GET /users/42

```
1  [
2      "id": 42,
3      "name": "Alice",
4      "email": "alice@example.com"
5  ]
```

- Et maintenant ?
  - Le client doit connaître /users/42/orders, /users/42/friends, etc.

# HATEOAS : ajouter des liens dans les réponses

```
1  {
2      "id": 42,
3      "name": "Alice",
4      "_links": {
5          "self": { "href": "/users/42" },
6          "orders": { "href": "/users/42/orders" },
7          "friends": { "href": "/users/42/friends" }
8      }
9  }
```

- Le serveur fournit les prochaines actions possibles
- Le client n'a plus besoin de connaître les routes à l'avance
- Le modèle devient navigable comme un site web

# HATEOAS : une contrainte REST oubliée

- Définition (Fielding, 2000) :

*“A REST application must be driven by hypermedia provided dynamically by the server.”*

- Le client ne doit interagir que via des liens hypermedia
- Les transitions d'état passent par des URLs dynamiques
- Le client ne doit pas construire les URLs lui-même

# Une interaction pilotée par les liens

GET /accounts/123

```
1  {
2      "id": 123,
3      "balance": 100,
4      "_links": {
5          "self":   { "href": "/accounts/123" },
6          "deposit": { "href": "/accounts/123/deposit", "method": "POST" },
7          "withdraw": { "href": "/accounts/123/withdraw", "method": "POST" },
8          "close":    { "href": "/accounts/123/close", "method": "DELETE" }
9      }
10 }
```

- Le serveur décrit ce que le client peut faire
- Le client n'invente pas d'URL, il suit les liens
- L'état applicatif est piloté par le hypermedia

# HATEOAS : quels avantages concrets ?

- Avantages
  - Client générique : fonctionne sans connaître les routes à l'avance
  - Maintenance facilitée : on peut changer les URLs côté serveur
  - Évolution fluide : ajout de fonctionnalités sans casser le client
  - Moins d'erreurs : les transitions autorisées sont explicites
- Limites
  - Peu utilisé dans les APIs REST "modernes"
  - Implémentation plus lourde côté serveur
  - Nécessite un design rigoureux

# HATEOAS aujourd’hui : encore pertinent ?

- HATEOAS reste un idéal architectural :
  - RESTful = ressources + verbes + hypermedia
  - Favorise le découplage client / serveur
  - Donne un cadre clair pour les transitions d'état

# Standard OpenAPI

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

# Pourquoi utiliser OpenAPI ?

- Communication claire entre équipes (backend, frontend, QA)
- Réduction des ambiguïtés (interprétations différentes des endpoints ou des formats de données) grâce à un contrat formel
- Favorise le développement **API-First**
- Documentation vivante, toujours synchronisée avec le code

# Structure d'un document OpenAPI

```
openapi: 3.0.0
info:
  title: API Produits
  version: 1.0.0
servers:
  - url: https://api.maboutique.com/v1
paths:
  /products/{id}:
    get:
      summary: Détail d'un produit
      parameters:
        - name: id
          in: path
          required: true
          schema: { type: integer }
      responses:
        '200':
          description: Produit trouvé
```

# Outils Swagger

- Swagger Editor : éditeur de descripteurs 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

# Standard OpenAPI : exemple Swagger UI

The screenshot shows the Swagger UI interface for a standard OpenAPI specification. It displays two main sections: 'pet' and 'store'.

**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 Swagger UI

The screenshot shows a standard OpenAPI endpoint definition using the Swagger UI. The endpoint is a GET request to `/pet/{petId}` with the description "Find pet by ID". A note below says "Returns a single pet". On the right, there's a lock icon and a "Try it out" button. The "Parameters" section is currently selected, showing a table with one row. The table has two columns: "Name" and "Description". The single parameter is `petId`, marked as required (\*), with type `integer` and path `(path)`. Its description is "ID of pet to return".

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

# Standard OpenAPI : exemple Swagger UI

GET /pet/{petId} Find pet by ID 

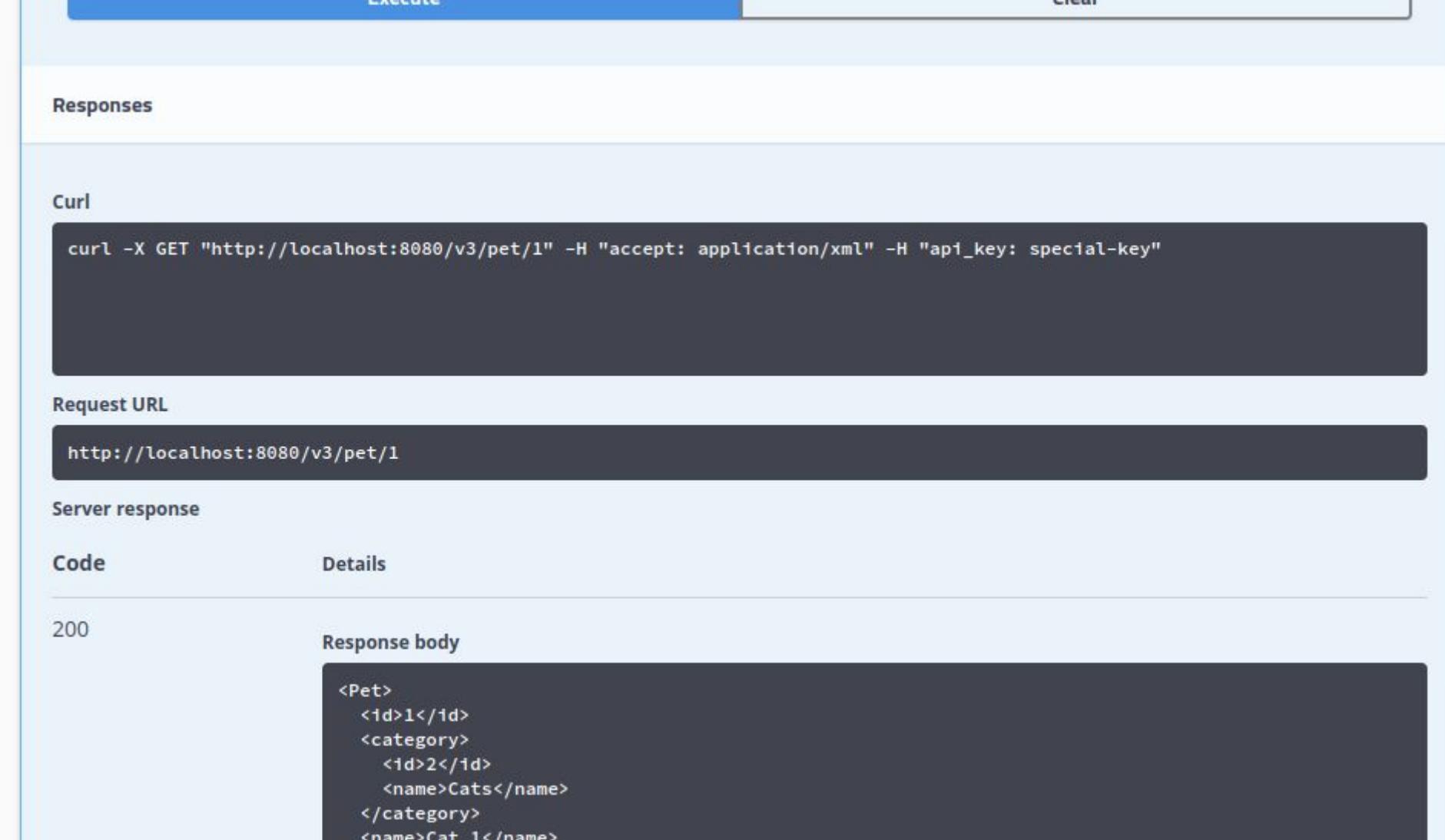
Returns a single pet

**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 Swagger UI



The screenshot shows the Swagger UI interface for a REST API. At the top, there is a blue bar with two buttons: "Execute" on the left and "Clear" on the right. Below this, the main area is titled "Responses".  
**Curl:**  

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

  
**Request URL:**  

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

  
**Server response:**  

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 Swagger UI

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 Swagger UI

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 <b>Pet not found</b>

# Bonnes pratiques avec OpenAPI

- Utiliser des schémas réutilisables dans components

# Bonnes pratiques avec OpenAPI

- Utiliser des schémas réutilisables dans components

```
components:  
  schemas:  
    Product:  
      type: object  
      properties:  
        id: { type: integer }  
        name: { type: string }  
        price: { type: number, format: float }
```

# Bonnes pratiques avec OpenAPI

- Utiliser des schémas réutilisables dans components
- Documenter tous les codes de retour HTTP (200, 400, 404, 500...)
- Fournir des exemples concrets de requêtes et réponses
- Séparer les fichiers YAML par ressource pour les grands projets
- Intégrer la validation OpenAPI dans la CI/CD

# Limites d'OpenAPI

- Conçu que pour les **APIs REST**
- Peut devenir **verbeux** pour les grandes APIs
- La génération de code produit un squelette, **pas une implémentation complète**
- Nécessite une **maintenance régulière** pour rester à jour -> manque de fiabilité
- **Expressivité limitée** -> ambiguïté et problèmes d'utilisabilité

# 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](#)

# OWASP Top 10

<https://owasp.org/www-project-top-ten/>

- 1. Broken Access Control** : accès non autorisés
- 2. Cryptographic Failures** : données mal protégées
- 3. Injection** : code malveillant inséré
- 4. Insecure Design** : conception vulnérable
- 5. Misconfiguration** : paramètres faibles
- 6. Outdated Components** : bibliothèques vulnérables
- 7. Auth Failures** : failles d'authentification
- 8. Integrity Failures** : dépendances non vérifiées
- 9. Logging Failures** : attaques non détectées
- 10. Server-Side Request Forgery**: serveur manipulé pour accéder à des ressources internes

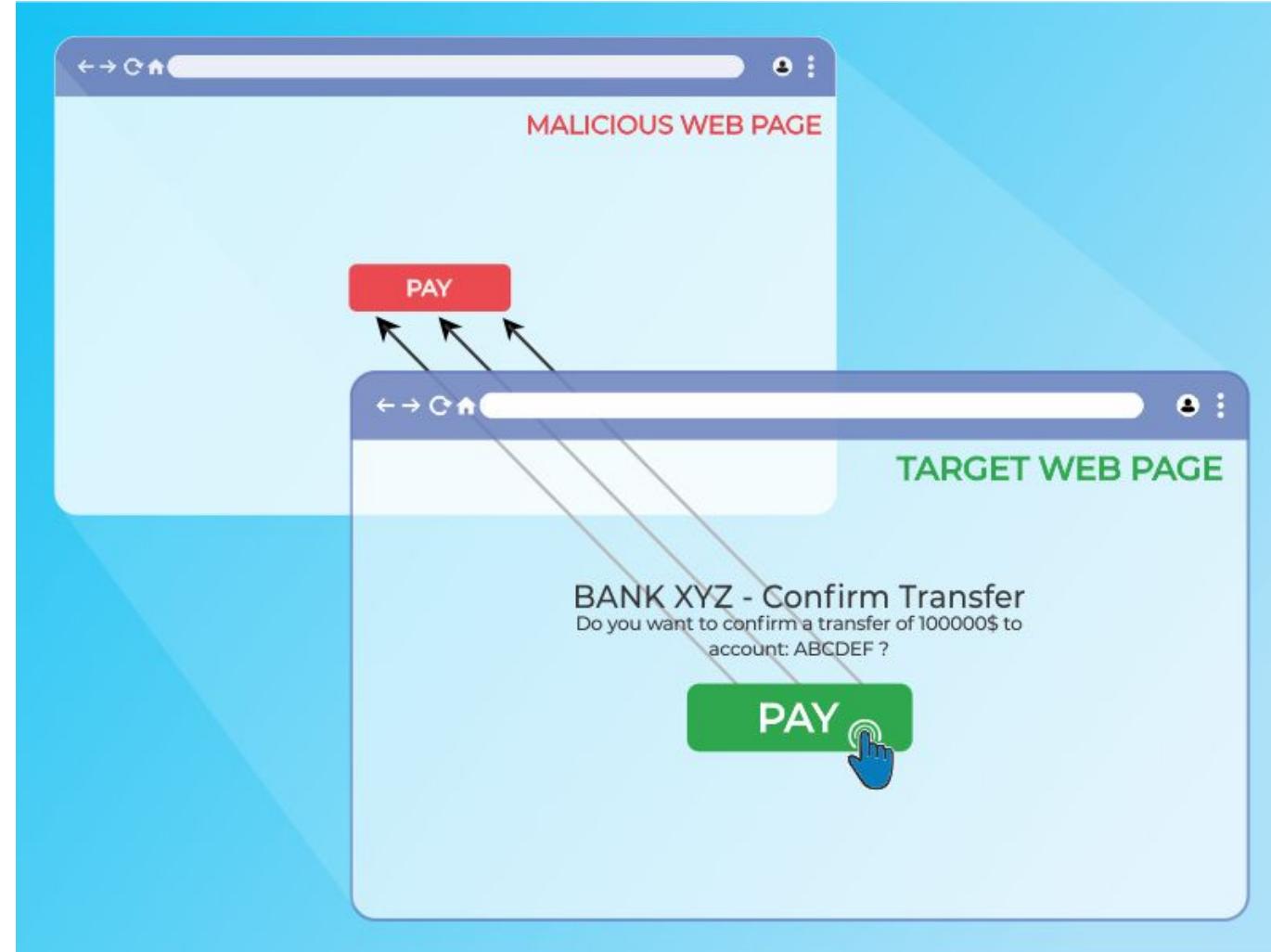
# Exemple d'attaque : Denial of Service attack

- 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 des réponses http

# Exemple d'attaque : ClickJacking



[https://owasp.org/www-project-web-security-testing-guide/v41/4-Web\\_Application\\_Security\\_Testing/11-Client\\_Side\\_Testing/09-Testing\\_for\\_Clickjacking](https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/11-Client_Side_Testing/09-Testing_for_Clickjacking)

# XSS (Cross-Site Scripting)

- Vulnérabilité permettant à un attaquant de faire injecter et exécuter du code externe (souvent du JS) par le client<sup>[1]</sup>
- Conséquences courantes : **détournement de session, vol d'identifiants utilisateurs, déni de service (DoS)**
- Peut aussi conduire à des virements bancaires non autorisés et au traçage d'utilisateurs si des **Service Workers** sont impliqués<sup>[2]</sup>
- Connu publiquement depuis le début des années 2000, **problème de sécurité le plus fréquent et le plus critique des applications web**

[1]: <https://www.sciencedirect.com/science/article/abs/pii/S1084804518302042?via%3Dihub>

[2]: <https://dl.acm.org/doi/10.1145/3471621.3471845>

# XSS (Cross-Site Scripting)

- Se produit lorsqu'une donnée contrôlée par l'attaquant est envoyée vers un point dangereux côté client, comme `innerHTML`, `eval`, `setTimeout`, `script.src`, `document.write`, etc.
- Principales variantes
  - **Reflected XSS** : le serveur renvoie directement au client l'entrée de l'attaquant (souvent via URL)
  - **Stored XSS** : entrée stockée dans la base et renvoyée automatiquement (commentaires, forums). Très dangereux
  - **DOM-based XSS** : côté client, le JavaScript manipule des données non vérifiées dans le DOM

# XSS (Cross-Site Scripting) - exemple<sup>[1]</sup>

- Voilà un exemple classique de vulnérabilité XSS basée sur le DOM. Regardons ce code ligne par ligne et analysons ce qui peut mal tourner :

```
const params = new URLSearchParams(document.location.search);  
const lang = params.get("lang");  
document.write(`<script src="${lang}/${lang}.js"></script>`);
```

[1]: <https://dl.acm.org/doi/10.1145/2808425.2808432>

# XSS (Cross-Site Scripting) - exemple<sup>[1]</sup>

- La traduction charge des scripts JS via le paramètre  
?LANG=fr
- Si LANG="onerror="alert(1)", le code devient :  
`<script src="" onerror="alert(1)"></script>`
- Le guillemet ferme l'attribut src et l'onerror est exécuté
- Le navigateur ne trouve pas le script et lance alert()
- L'attaquant peut diffuser le lien via phishing ou ingénierie sociale

[1]: <https://dl.acm.org/doi/10.1145/2808425.2808432>

# XSS (Cross-Site Scripting) - mitigation

- **Sanitisation des entrées utilisateur :**
  - Considérer toutes les entrées comme potentiellement dangereuses
  - Utiliser des bibliothèques de filtrage largement testées pour nettoyer ou échapper les données avant toute injection dans le DOM
  - Valider le type, la longueur et le format des données reçues
- **Content Security Policy (CSP) :**
  - Définir des listes blanches pour les scripts et ressources autorisés, e.g., Content-Security-Policy: script-src 'self';
  - Seuls les scripts provenant du domaine d'origine seront exécutés

# CSRF (Cross-Site Request Forgery)

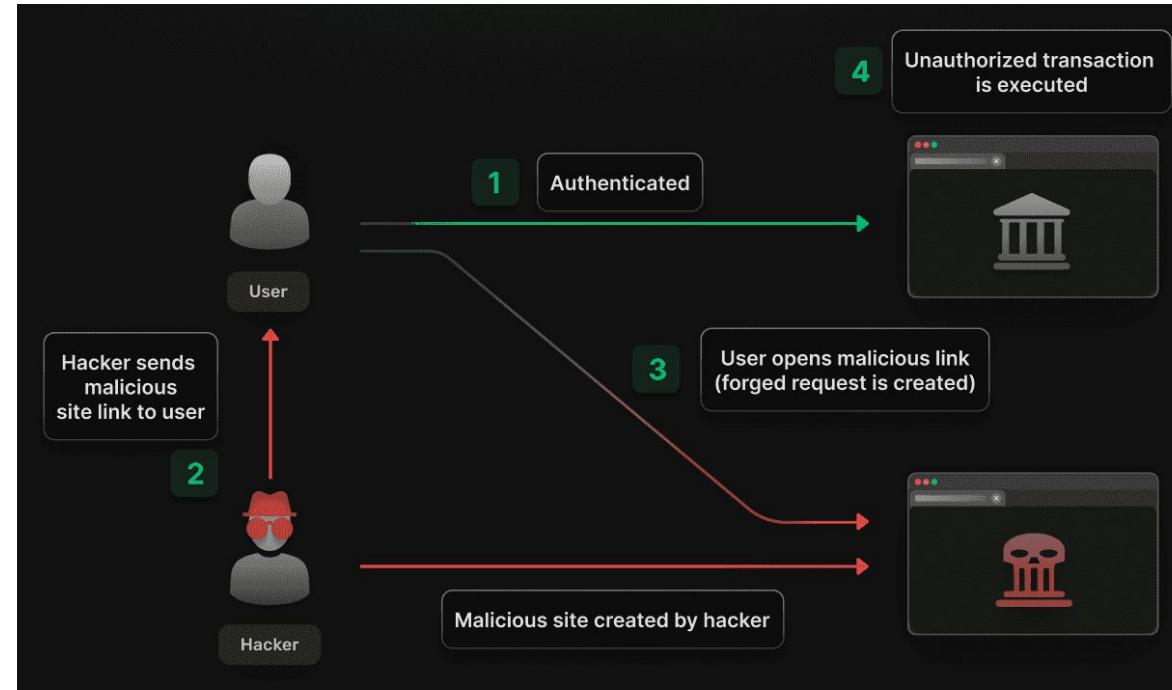
- L'attaquant fait effectuer à un utilisateur connecté une requête non désirée (à son insu) [1,2]
- Exploite les cookies d'authentification / tokens de session  
Les navigateurs joignent automatiquement ces données, même pour des requêtes cross-domain
- Conséquences courantes : changement de mot de passe, virements bancaires, modification de paramètres ou suppression de données

[1]: <https://ieeexplore.ieee.org/document/10646795>

[2]: [https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site\\_Request\\_Forgery\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html)

# CSRF (Cross-Site Request Forgery) - exemple

- Un utilisateur est connecté à un site web (`bank.com`) et navigue sur un site malveillant (`evil.com`)
- Ce dernier exécute alors une requête POST vers le premier, réalisant des transactions financières indésirables



[1]: <https://ieeexplore.ieee.org/document/10646795>

# CSRF (Cross-Site Request Forgery) - mitigation

- **Jeton CSRF** : valeurs aléatoires générées par le serveur et intégrées dans les formulaires ; si le serveur ne reçoit pas le jeton attendu, la requête HTTP est rejetée
- **En-têtes** Referrer ou Origin : contiennent le domaine d'origine de la requête (mais soulèvent des problèmes de confidentialité et sont partiellement supportés par les navigateurs)
- **Politique CORS** (Cross-Origin Resource Sharing)
- **Cookie** SameSite
- **Liste blanche de domaines** (Content Security Policy)
- **Proxys**

# 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());
```

# 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

# 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

# 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

# Pourquoi HTTPS ?

- HTTP = protocole non sécurisé (tout est lisible sur le réseau par des outils comme WireShark)
- Vulnérable aux attaques : sniffing, usurpation, modification des données
- HTTPS = HTTP + chiffrement TLS → confidentialité, intégrité, authentification
- Devenu indispensable pour les sites web (ex. navigateurs marquent "Non sécurisé" en HTTP)

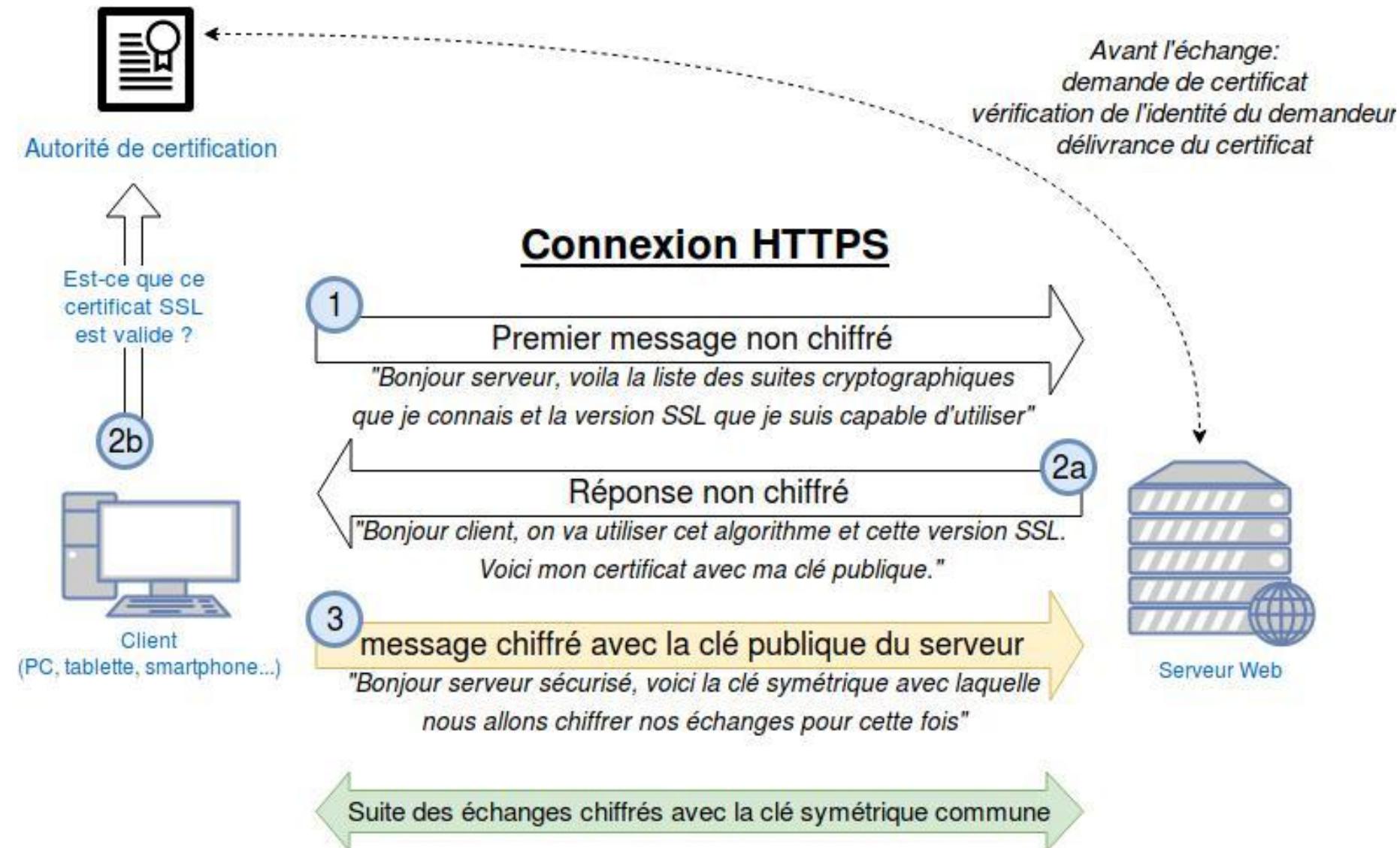
# HTTPS : objectifs

- **Intégrité** : éviter la modification des données en transit
- **Confidentialité (Privacy)** : protéger les échanges contre l'espionnage
- **Authentification** : vérifier l'identité du serveur

# Fonctionnement général

- Basé sur **SSL/TLS** (actuellement TLS 1.2 / 1.3)
- **Handshake initial** → négociation de la clé de session
- **Chiffrement asymétrique** (clé publique/clé privée) pour sécuriser l'échange de la clé de session générée par le client
- **Chiffrement symétrique** ensuite → messages plus rapides à chiffrer/déchiffrer
- Toute la pile HTTP est chiffrée (Headers + Body)
- Seule l'adresse IP reste lisible (nécessaire pour le routage)

# Handshake TLS



# Certificats numériques

- Basés sur le standard **X.509**
- Requis pour l'authentification et le chiffrement
- Délivrés par des **Autorités de Certification (CA)** : DigiCert, GlobalSign, etc.
- Contiennent : clé publique, identité du serveur, signature de la CA, validité

# Certificats numériques

- Version : 3 (standard actuel X.509v3)
- Issuer (Émetteur) : autorité de certification (par exemple: Let's Encrypt R3)
- Subject (Sujet) : le serveur concerné (www.example.com)
- Validity : date de début et d'expiration
- Public Key : clé publique associée au serveur
- Signature : signature numérique faite par la CA

# Certificats numériques

## Émis pour

Nom commun (CN) mail.google.com  
Organisation (O) <Ne fait pas partie du certificat>  
Unité d'organisation (OU) <Ne fait pas partie du certificat>

## Émis par

Nom commun (CN) WR2  
Organisation (O) Google Trust Services  
Unité d'organisation (OU) <Ne fait pas partie du certificat>

## Durée de validité

Émis le lundi 11 août 2025 à 21:23:14  
Expire le lundi 3 novembre 2025 à 20:23:13

## Empreintes SHA- 256

Certificat 7ada3ad285c59bde2a89c6519041e8187a69a671790039bd5d97dfe67f1  
c6d60  
Clé publique bb3f4f99fcbd1c8846918a2b7b9ddf4477bb7ea5d4100bc2ed5716a2d66  
0c7e6

# Types de certificats

- **Payants** : coût (200–500€ selon le type)
- **Gratuits** : ex. **Let's Encrypt** (automatisé, renouvellement 90 jours)
- **Auto-signés** : gratuits, mais non reconnus par défaut (utilisés en test)

# Limites et complexité

- Authentification forte via certificats côté client = lourde à gérer
- Besoin d'un certificat par poste → peu utilisé
- Distribution, renouvellement et mise à jour des certificats = délicat
- En pratique → on authentifie **l'application ou le serveur**, rarement l'utilisateur

# HTTPS aujourd'hui

- Standard obligatoire pour sites e-commerce, banques, réseaux sociaux
- Navigateurs modernes bloquent certaines fonctionnalités en HTTP (géolocalisation, caméra, etc.)
- Généralisation grâce à **Let's Encrypt** et l'automatisation

# Attaques possibles

- **Man-in-the-Middle** (si certificat compromis ou accepté manuellement)
- **Certificats frauduleux** (CA piratée ou malveillante)
- Vulnérabilités dans TLS anciennes versions (SSLv2/3, TLS 1.0/1.1 → obsolètes)

# Bonnes pratiques

- Toujours utiliser TLS 1.2 ou 1.3
- Renouveler les certificats à temps
- Forcer le HTTPS (HTTP Strict Transport Security ou HSTS)
- Utiliser Let's Encrypt pour automatiser
- Désactiver les suites de chiffrement faibles

# 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

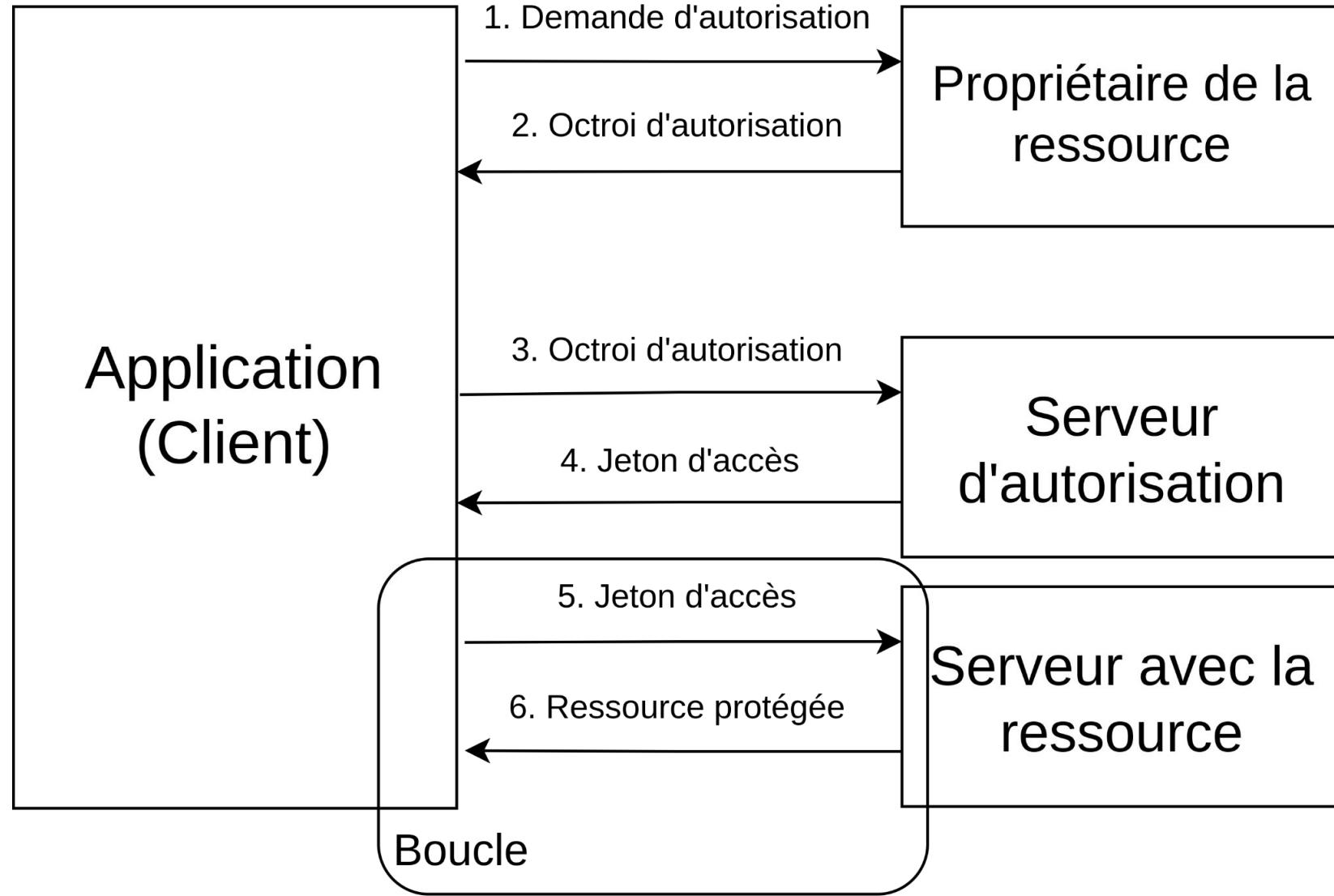
# OAuth2 : acteurs

- **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 sur 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  
JuYW1lIjoiQm9iYmEgRmV0dCI&lt;img alt="Copy icon" data-bbox="380 400 400 420"/>  
_bmDGagYCISGPizQU62hwXVQkpaGYiAdnWusr6I  
ViU
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYOUT: DATA

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

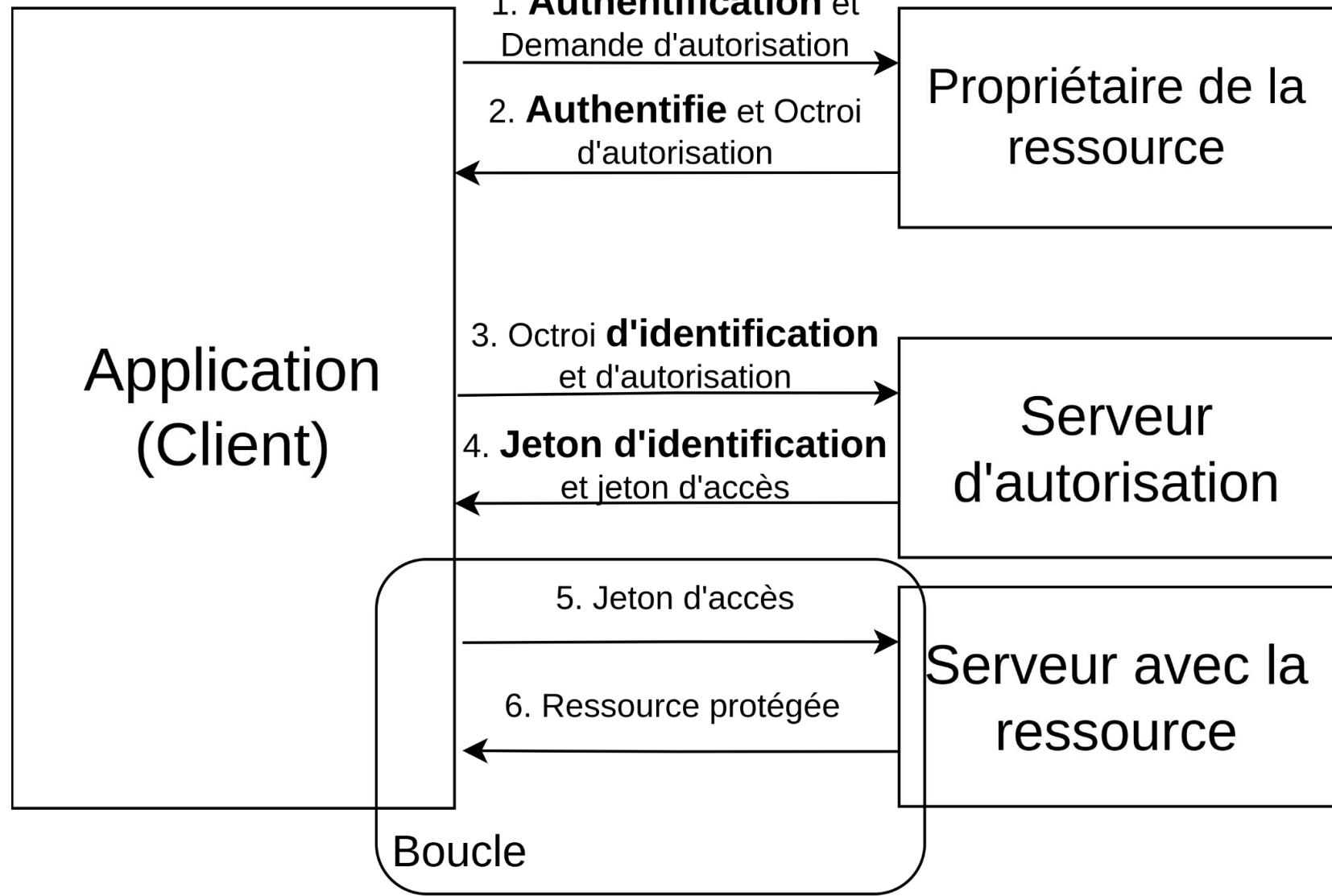
VERIFY SIGNATURE

```
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



# Au delà du Backend: Accessibilité des applications Web

- Pourquoi ?
- Les bonne pratiques

# Accessibilité des Applications Web

- Garantir que les sites et applis web soient utilisables par les personnes en situation de handicap
- Concerne la **vision, l'audition, la motricité, la cognition**
- Fondée sur les normes internationales **WCAG (Web Content Accessibility Guidelines)**

# Accessibilité des Applications Web - suite

- Le respect des normes WCAG peut améliorer considérablement votre **optimisation pour les moteurs de recherche** (*Search Engine Optimization, SEO*)
- De nombreux **validateurs** en ligne peuvent fournir des informations sur l'accessibilité (<https://wave.webaim.org/>)

# Enjeux

- Favoriser l'inclusion et l'égalité d'accès à l'information
- Répondre aux obligations légales (RGAA, directive européenne...)
- Améliorer l'expérience utilisateur pour tous
- Élargir l'audience et l'impact économique

# Principes WCAG (POUR)

- WCAG = Web Content Accessibility Guidelines (directives internationales du W3C)
- Basées sur 4 principes (POUR) :
  - **Perceivable** : contenu visible/entendable
  - **Utilisable** : navigation simple (clavier, souris)
  - **Compréhensible (Understandable)** : langage clair, interface cohérente
  - **Robuste** : compatible avec navigateurs et lecteurs d'écran
- Niveaux de conformité : A, AA (recommandé), AAA (avancé)

# Bonnes Pratiques

- Ajouter des textes alternatifs aux images
- Assurer un contraste suffisant entre texte et arrière-plan
- Permettre la navigation complète au clavier
- Utiliser une structure sémantique correcte (titres, listes...)
- Rendre les formulaires clairs et accessibles

# Bonnes Pratiques - exemples

## Images



Texte alternatif clair : « Photo d'un bouton panier »



Image sans description

## Navigation au clavier



Tous les champs accessibles avec Tab



Boutons utilisables uniquement à la souris

# Bonnes Pratiques - exemples

## Couleurs et contrastes

-  Texte noir sur fond blanc ou bleu foncé
-  Texte gris clair sur fond blanc (illisible)

## Messages d'erreur

-  « Veuillez entrer une adresse email valide »
-  « Erreur » sans explication

# Bonnes Pratiques - exemples

## Sous-titres



Vidéo avec sous-titres et transcription



Vidéo sans alternatives

## Formulaires



Champs avec étiquettes claires (Nom, Email...)



Champs sans labels, incompréhensibles pour un lecteur d'écran

# Bonnes Pratiques - exemples

## Graphiques

-  Courbes différencierées par couleurs + motifs (pointillés, traits)
-  Courbes uniquement rouges et vertes

## Web assets (images, vidéo)

-  Des images de qualité différente pour permettre au site d'être utilisé même dans des contextes de faible connectivité (**graceful degradation principle**)
-  Que des images de haute qualité