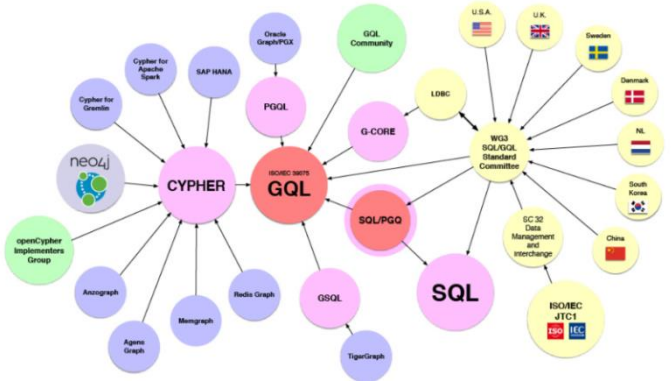
# Graph query language

## GraphQL/ WebService

# Chapitre 2

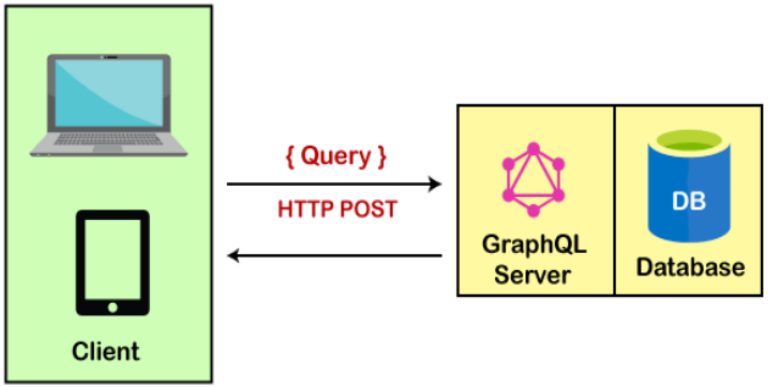



Pr. Sara Belattar

# GraphQL

**Historique / introduction**

- **GraphQL** (pour **Graph Query Language**) est un langage de requêtes et un environnement d'exécution, créé par Facebook en 2012 (qui a commencé à l'utiliser pour les applications mobiles en 2012 ), avant d'être publié comme projet open-source en 2015. Inscrit dans le modèle Client-Serveur, il propose une **alternative aux API REST.**



# GraphQL

Limitations de Rest

## Rest

- Une **API REST** suit une structuration claire orientée ressources
- Si toutefois la taille des données devient importante les problèmes suivants arrivent :
  - Envoi d’un grand nombre de données, parfois inutiles, sur le réseau
    - **“over-fetching”**
  - Besoin d’un grand nombre de points d’entrée **et de chemins** très longs pour définir la ressource

# GraphQL

## Avantages de GraphQL

- ✓ **GraphQL** structure les données sous la forme d'un graphe (**d'où son nom**) et offre un langage de requête pour parcourir les données et les récupérer de façon structurée

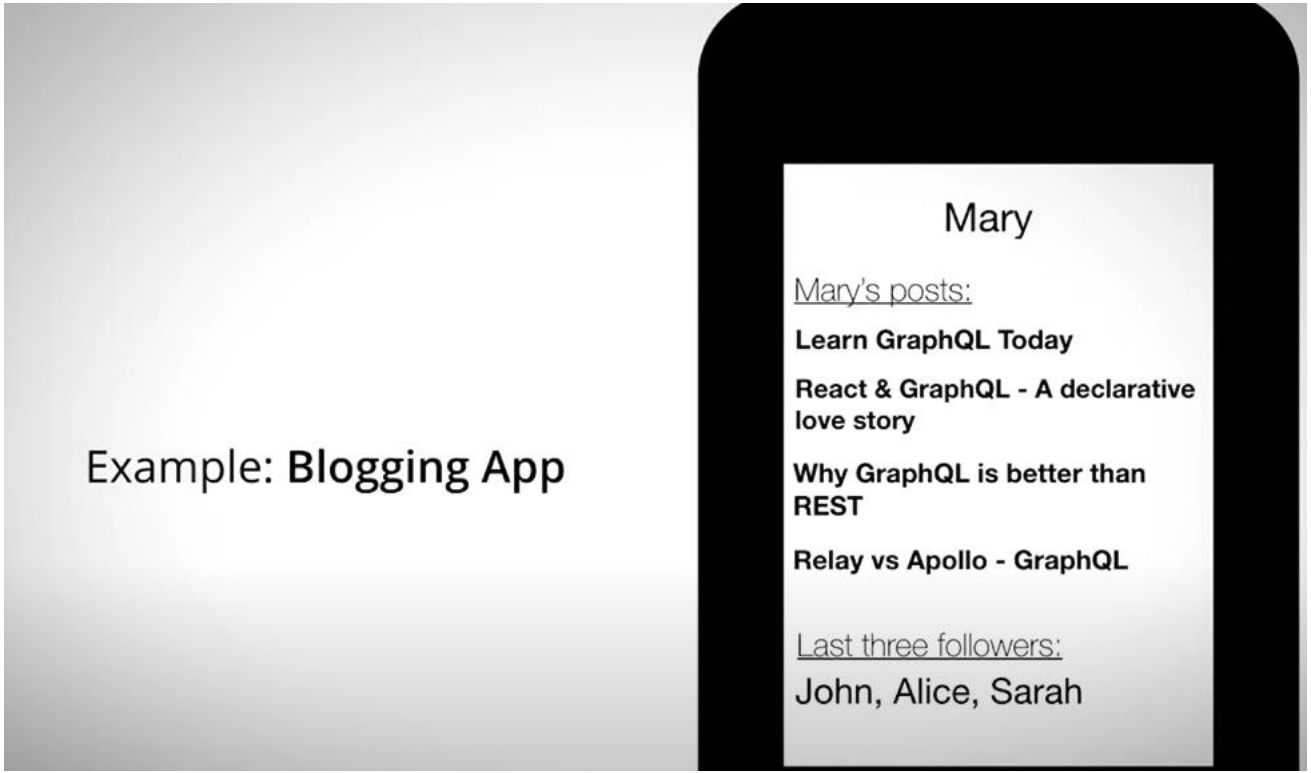
- ✓ **GraphQL** est un langage de requête et un environnement d'exécution côté serveur pour les APIs qui permet de fournir aux clients ***uniquement les données qu'ils ont demandées***.

- ✓ **GraphQL** est conçu pour mettre à la disposition des développeurs des API rapides, flexibles et faciles à utiliser.

- ✓ Utilisé à la place de **REST**, ***GraphQL*** permet aux développeurs de créer des requêtes qui extraient les données de plusieurs sources à l'aide d'un seul appel d'API

# GraphQL

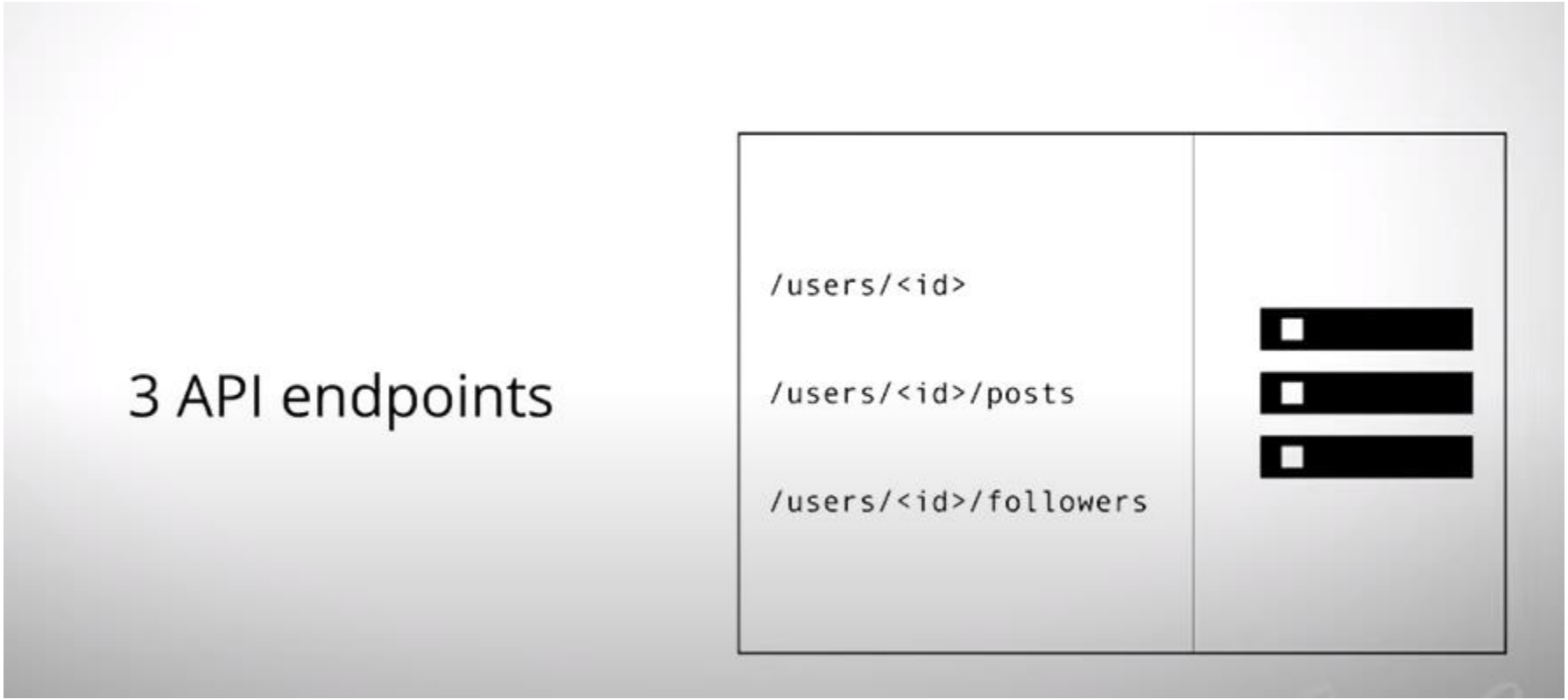
Exemple



# GraphQL

REST

Exemple



# GraphQL

REST

Exemple

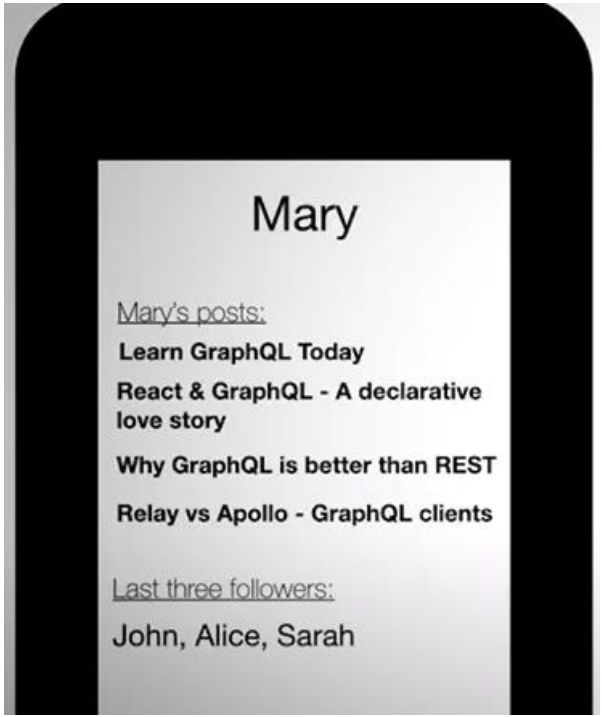
# GraphQL

REST

Exemple

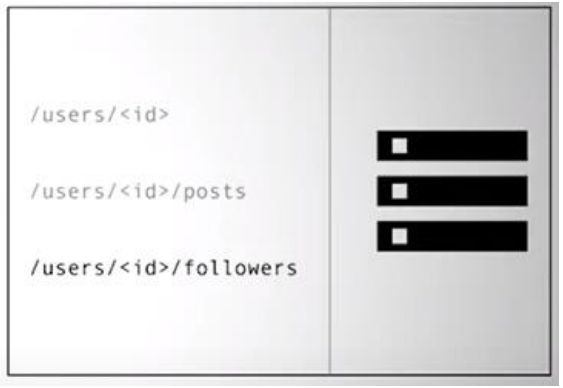


REST




# GraphQL

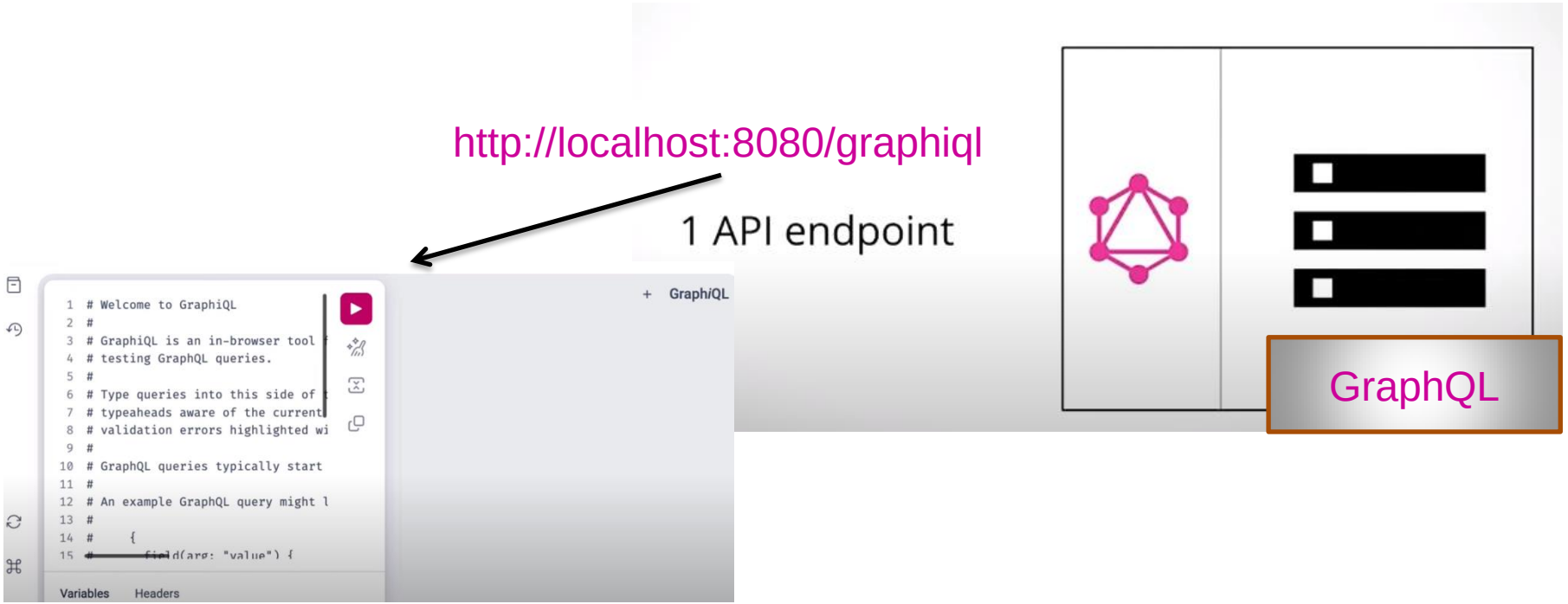
Exemple

# GraphQL

Exemple

GraphQL



# GraphQL

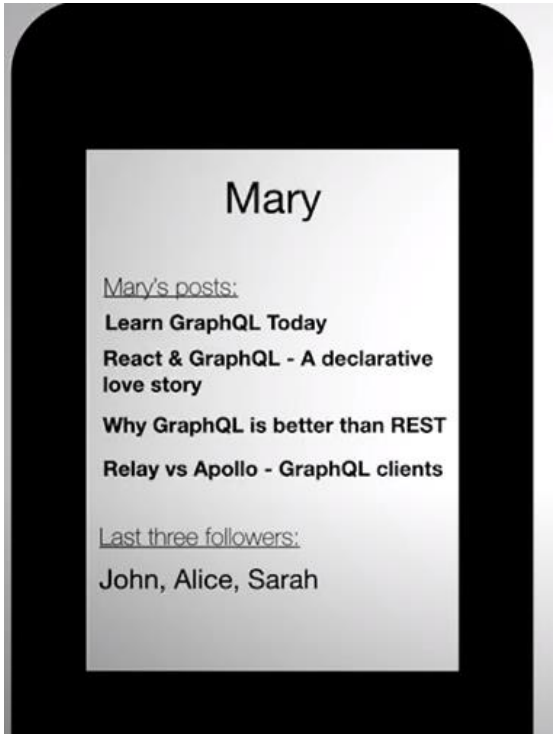
Exemple

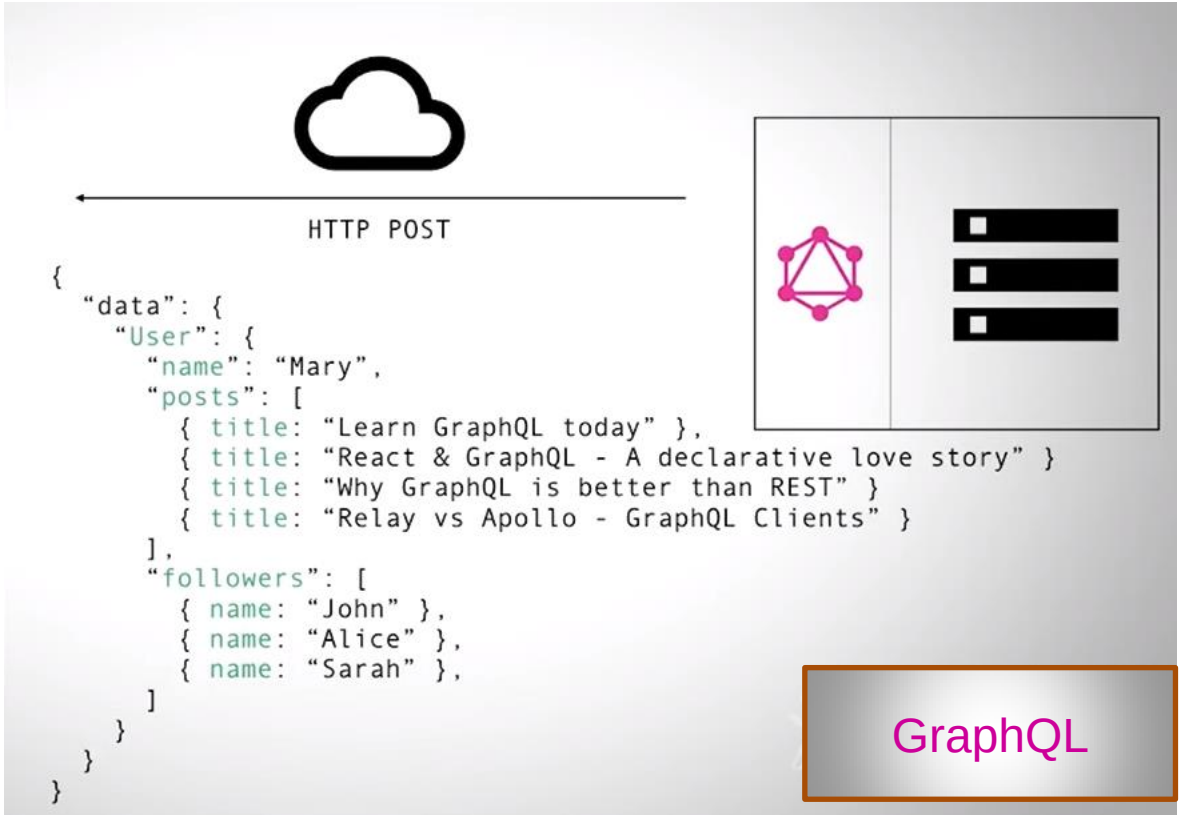
GraphQL

# GraphQL

Exemple

GraphQL



# GraphQL

## Schémas, résolveurs

- On parle aussi de **schéma** en **GraphQL** au lieu d'API
- **Il s'agit de définir la structure des éléments auxquels on peut accéder par l'API**
- Il y a deux types principaux dans un schéma
  - les "**queries**" qui définissent les requêtes possibles sur le serveur,
  - les "**mutations**" qui définissent les mises à jour, suppressions ou créations de données,

# GraphQL

## Schémas, résolveurs

```
type Query{
    customers:[CustomerDto]
}

type Mutation {
    createCustomer(dto:AddCustomerRequest):AddCustomerResponse
}

type CustomerDto {
    username : String,
    identityRef : String,
    firstname:String,
    lastname:String,
}


input AddCustomerRequest {
    username : String,
    identityRef : String,
    firstname:String,
    lastname:String
}

type AddCustomerResponse {
    message:String,
    username:String,
    identityRef:String,
    firstname:String,
    lastname:String
}
```

- customers est le nom du service pour consulter les clients.

- CustomerDto est le type des données qui seront retournés au clients.

- Pour les requêtes Select, le mot clé utilisé est Query.

- Toutes les requêtes autres que Select, le mot clé utilisé est Mutation.

- createCustomer est le nom du service pour créer un nouveau client.

- La requête createCustomer prends en paramètre un objet de type AddCustomerRequest.

- L'objet résultat de la requête createCustomer est AddCustomerResponse.

- Pour déclarer les objets de type paramètre, on utilise le mot clé "input".

- Pour déclarer les ojets de type response, on utilise le mot clé "type".

# GraphQL

## Schémas, résolveurs

```java
@Controller
public class CustomerGraphqlController {
    4 usages
    private final ICustomerService customerService;

    public CustomerGraphqlController(ICustomerService customerService) {...}

    @QueryMapping
    List<CustomerDto> customers() {
        return customerService.getAllCustomers();
    }

    @MutationMapping
    public AddCustomerResponse createCustomer(@Argument("dto") AddCustomerRequest dto) {
        return customerService.createCustomer(dto);
    }
```

- customers() est le nom du resolver qui sera exécuté par le serveur GraphQL lorsque la requête customers est envoyée.

- Pour les requêtes GraphQL de type Query, on annote les resolver avec l'annotation @QueryMapping.

- createCustomer() est le nom du resolver qui sera exécuté par le serveur GraphQL lorsque la requête createCustomer est envoyée.

-Le paramètre dto doit être annoté par @Argument afin de mapper le nom du paramètre en Input de la requête createCustomer avec le paramètre passé à la méthode createCustomer.

- Toutes les méthodes GraphQL (type Query ou bien Mutation) sont de type POST.

# ModelMapper

## Model Mapper

- **DTO (Data Transfer Object) :** Un **DTO** est un objet qui transporte des données entre les couches d'une application. Il est souv ent utilisé pour encapsuler un ensemble de données et le transférer entre différentes parties du système, généralement entre la couche de présentation et la couche de service. Les DTO sont généralement légers et ne contiennent que les données nécessa ires à une opération spécifique.

- **BO (Business Object) :** Un BO, ou Business Object, est un objet qui représente une entité métier dans l'application. Il contient généralement la logique métier et les règles associées à cette entité. **Les objets BO** sont souvent utilisés dans la couche de service de l'application pour effectuer des opérations métier.

L'objectif principal de **ModelMapper** dans ce contexte est de permettre une conversion transparente entre ces deux types d'objets .Plus précisément :

- **Conversion DTO vers BO :** Lorsqu'une requête arrive à la couche de service (par exemple, depuis la couche de présentation), elle est souvent accompagnée d'un DTO. **ModelMapper** est utilisé pour convertir ce **DTO** en un objet **BO**, de manière à ce que la couche de service puisse travailler avec l'objet **BO** pour effectuer les opérations métier nécessaires.

- **Conversion BO vers DTO :** Lorsque la couche de service a terminé son travail et doit renvoyer des données à la couche de pr ésentation, les résultats sont souvent encapsulés dans un objet **BO**. **ModelMapper** est utilisé pour convertir cet objet **BO** en un **DTO** approprié qui peut être renvoyé à la couche de présentation.

# Enum

## Enum

- **Enum**
-

En Java, **enum** est un mot-clé utilisé pour déclarer une énumération, une sorte de type de données spécialisé qui représente un ensemble fixe de constantes nommées. L'utilisation de public enum permet de déclarer une énumération avec une visibilité publique, ce qui signifie que l'énumération est accessible depuis d'autres classes.

```
public enum DayOfWeek {
    // Les constantes enum représentent les jours de la semaine
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}
```

# Enum

## Enum

- **Enum**

Dans cet exemple, DayOfWeek est une énumération publique représentant les jours de la semaine. Chaque jour est une constant e enum (SUNDAY, MONDAY, etc.). Vous pouvez utiliser cette énumération dans d'autres parties de votre code de la manière suiva nte :

```
public class Example {
    public static void main(String[] args) {
        // Utilisation de l'énumération
        DayOfWeek today = DayOfWeek.WEDNESDAY;

        // Comparaison avec une valeur enum
        if (today == DayOfWeek.WEDNESDAY) {
            System.out.println("C'est mercredi !");
        }

        // Utilisation d'une boucle avec une énumération
        for (DayOfWeek day : DayOfWeek.values()) {
            System.out.println(day);
        }
    }
}
```