MyDigitalSchool
2021/2022
B3 - Développeur Web

API

Données persistantes

Mr Bonnet Denis

Index

Problématique de la persistance

Etat des lieux :

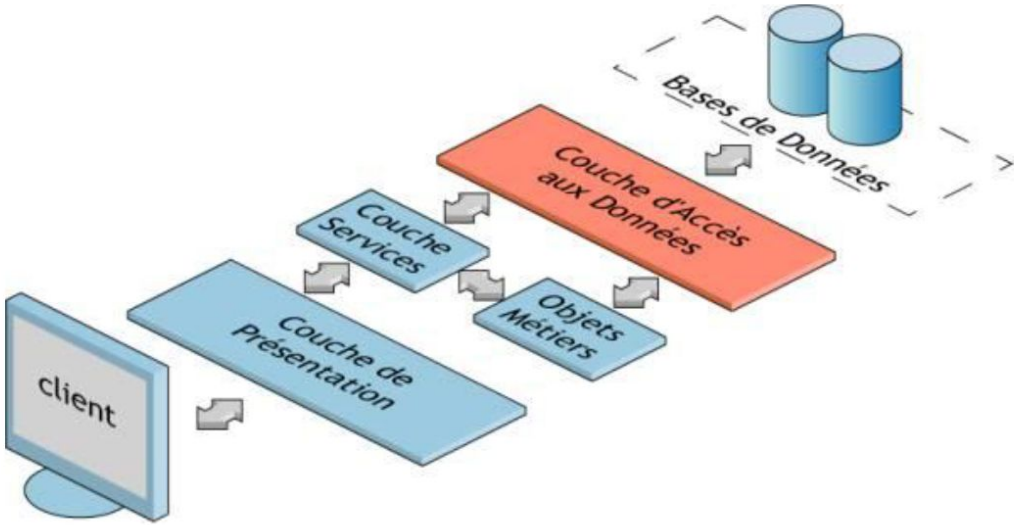
- Stockage données web
- Taille espace de stockage
- Majorité de langage orienté objet
- Schéma en UML
- Bases de donnée relationnelle majoritaires

Propriétés à conserver :

- Objets complexes
- Identification des objets
- Classes
- Hiérarchie de classes

Propriétés à ajouter :

- Persistance
- Interrogation
- Gestion de la concurrence
- Sécurité et reprise après panne
- Gestion de la mémoire secondaire


Bases de Données
Couche d'Accès aux Données
Couche Services
Objets Métiers
Couche de Présentation
client

Le mapping relationnel/objet

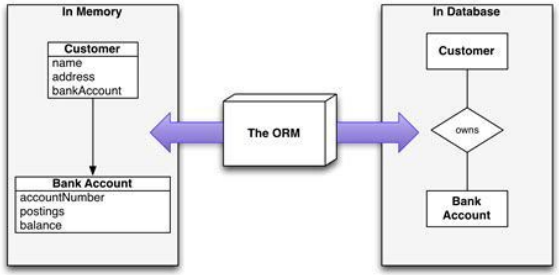
Développement Front

API

Mapping relationnel/objet

Le mapping relationnel/objet =>
Liaison entre un schéma Relationnel et un diagramme Objet

Problème :
SGBD sont conçu pour des tupples simples
Pas d'héritage de classe
Choisir entre un accès à la DB *public* ou *private*



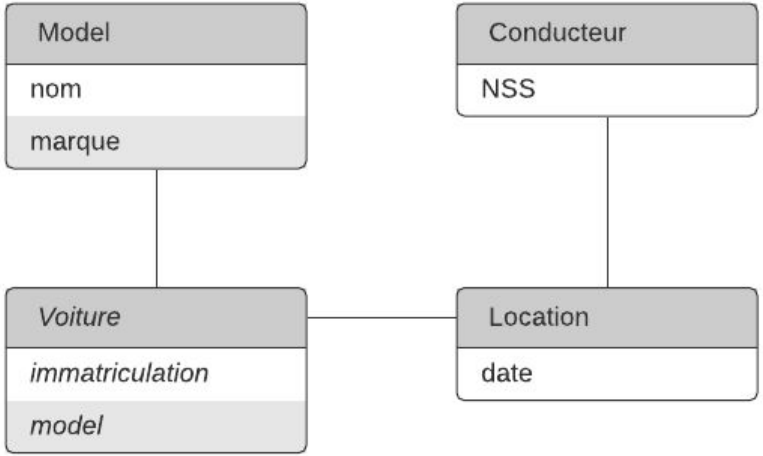
Diverses solutions :

Coder notre propre solution
Se baser sur un pattern existant
Se servir d'une API ORM
Utiliser une API REST

Coder notre propre solution (pattern active record):
~~*Serializer notre classe pour la stocker*~~ => cela demande trop de ressources. Un données est modifié, il faut charger TOUS l'objet.

Association Classe/Table :

- Classe scalaire => 1 table
- Classe avec association 1...n => 1 table scalaire avec FK pour l'association
- Classe avec association n...n => 1 table scalaire + une table de liaison



Développement Front

API

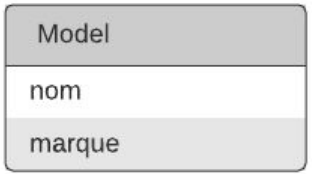
Mapping relationnel/objet

Class seule

```
public class Model{
// Fields
private String nom;
private String marque;

…
}
```

```
CREATE TABLE Model
( Model_ID SERIAL,
nom varchar(10) NOT NULL,
…
CONSTRAINT PK_Model PRIMARY KEY (Model_ID),
) ;
```



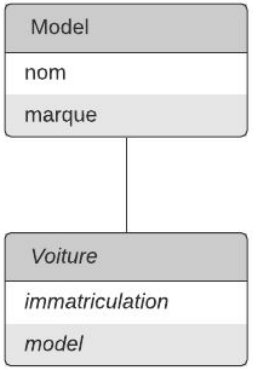
Développement Front

API

Mapping relationnel/objet

Liaison 1...n

```
public class Model{
// Fields
private String nom;
private Collection<Voiture> parc_automobile;
…
}
public class Voiture {
// Fields
private String immatriculation;
private Model model;
…
}
```




```
CREATE TABLE Model
( Model_ID SERIAL,
nom varchar(10) NOT NULL,
…
CONSTRAINT PK_Model PRIMARY KEY (Model_ID),
) ;

CREATE TABLE Voiture
( Voiture_ID SERIAL,
immatriculation varchar(10) NOT NULL,
…
model_ID int NOT NULL,
CONSTRAINT PK_Voiture PRIMARY KEY
(Voiture_ID),
CONSTRAINT FK_Model_Voiture
FOREIGN KEY (model_ID) REFERENCES Model
(Model_ID)
);
```

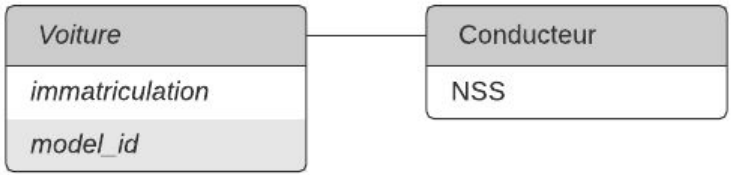
Développement Front

API

Mapping relationnel/objet

Liaison n...n

```
public class Conducteur{
// Fields
private String NSS;
private Collection<Voiture> location;
…
}
```



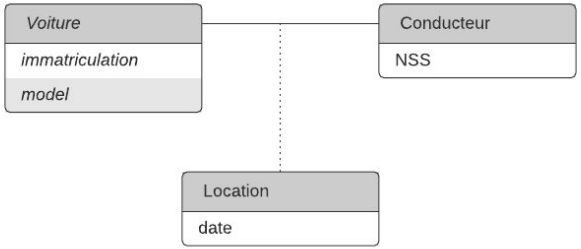

```
CREATE TABLE Conducteur
( Conducteur_ID SERIAL,
NSS varchar(10) NOT NULL,
…
CONSTRAINT PK_Conducteur PRIMARY KEY (Conducteur_ID),
) ;

CREATE TABLE Location
( …
  CONSTRAINT PK_Location PRIMARY KEY
     (Conducteur_ID,Voiture_ID),
  CONSTRAINT FK_Location_Voiture FOREIGN KEY
      (Voiture_ID) REFERENCES Voiture (Voiture_ID),
  CONSTRAINT FK_Location_Conducteur FOREIGN KEY
      (Conducteur_ID) REFERENCES Conducteur
      (Conducteur_ID),
);
```

Liaison n...n avec données

```
public class Location {
private Conducteur loueur;
private Voiture vehicule;
private Date date;
…
}
```

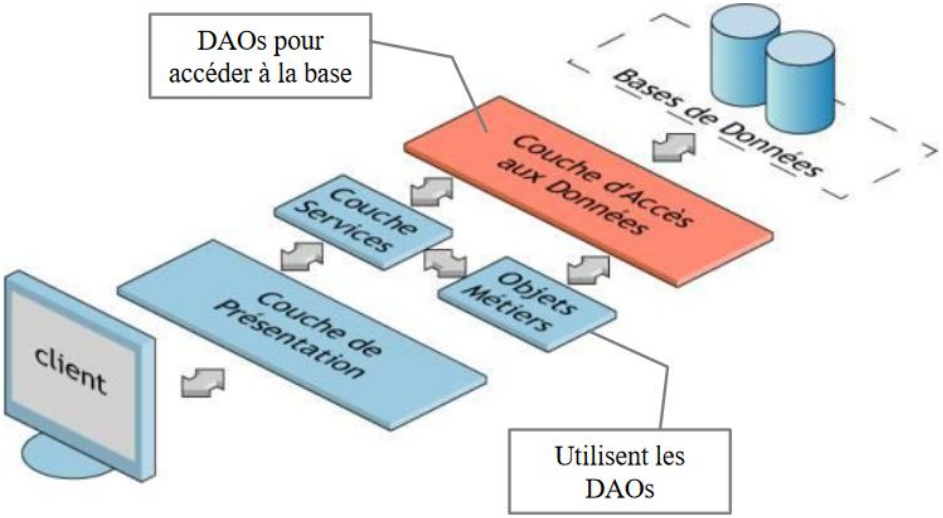
```
CREATE TABLE Location
( …
  Date DATE ,
  CONSTRAINT PK_Location PRIMARY KEY
     (Conducteur_ID,Voiture_ID),
  CONSTRAINT FK_Location_Voiture FOREIGN KEY
      (Voiture_ID) REFERENCES Voiture (Voiture_ID),
  CONSTRAINT FK_Location_Conducteur FOREIGN KEY
      (Conducteur_ID) REFERENCES Conducteur
      (Conducteur_ID),
 ;
```



Le patron DAO

Développement Front

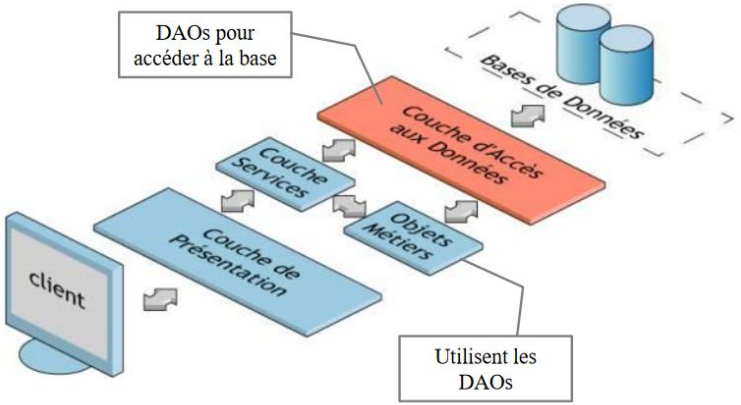
API
DAO



Design Pattern DAO => Data Access Object

Différence entre un DAO et un active record :
« Les objets manipulant les données n'ont pas accès au code permettant de sauvegarder ces données dans la base »

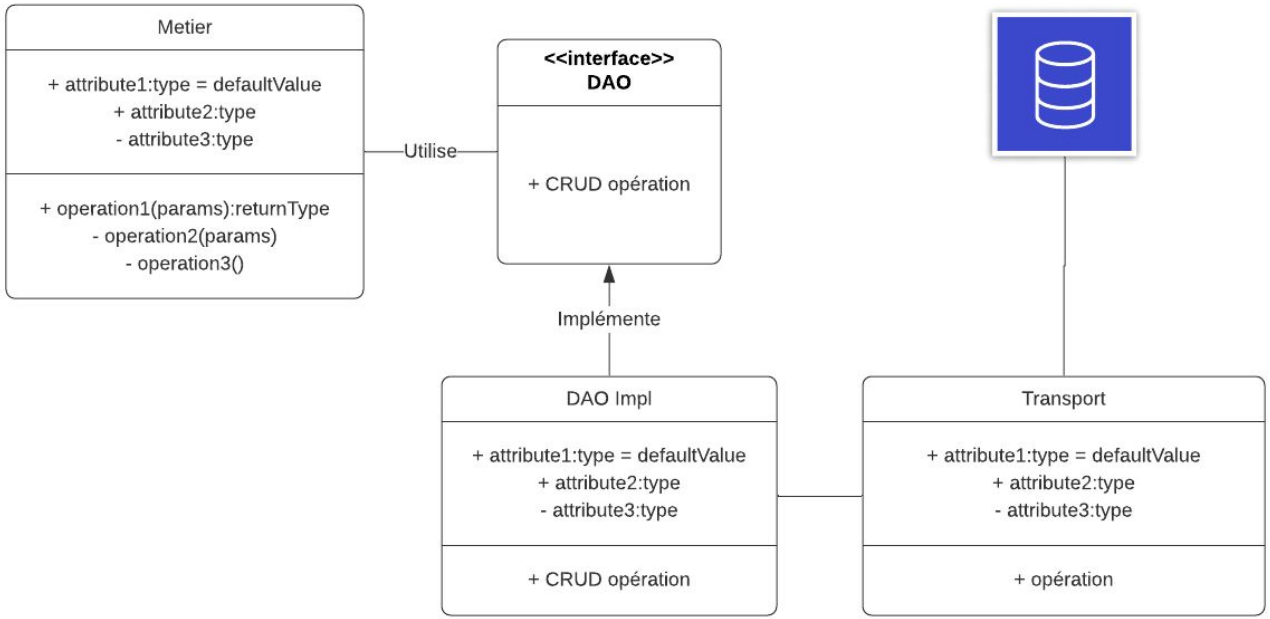
Utilité des DAO :
• Faciliter la modification du modèle de base de données
• Factoriser le code d'accès aux données
• Faciliter l'optimisation des accès à la base en les regroupant au sein d'objets particuliers



Développement Front

API

DAO



Développement Front

API

DAO

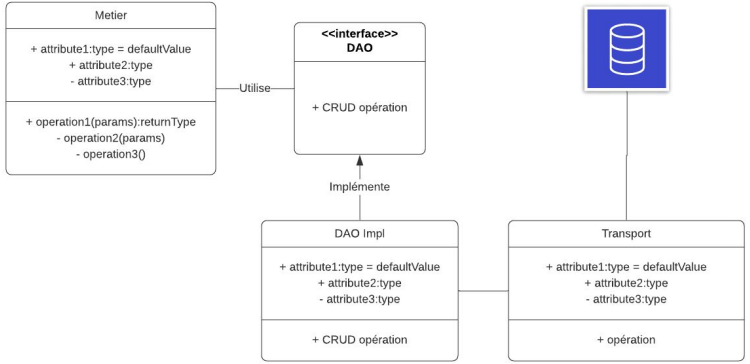
Class Metier : class que le logiciel manipule

Interface DAO : interface stocké par métier pour plus de polymorphisme

Class DAOImpl : Implémentation de l'interface pour chaque support de persistance

Class Transport : gestion de la connexion avec la DB, le fichier...

La zone de stockage : SGBD, fichier binaire, texte...



Développement Front

API

DAO

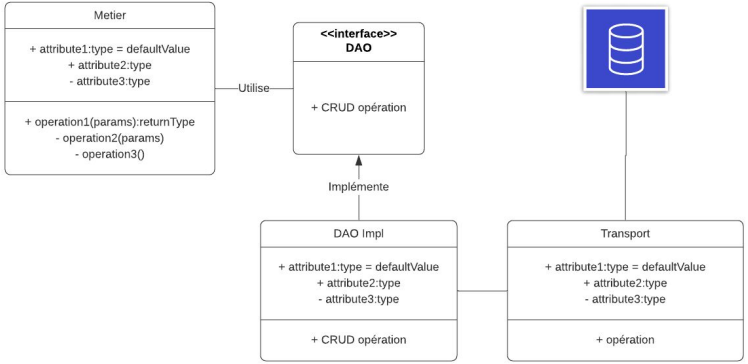
DAO :

Méthode CRUD : Create, Read, Update, Delete

Objet entier ou seulement une partie

2 stratégies :
DAO référencé par chaque objet métier pour sa propre persistance
• Aucune connaissance des DAO par les programmes
qui manipulent les objets métiers
• Nécessité d'une référence vers le DAO utilisé (ex.
obtenue par une méthode static de la classe DAO)

DAO directement manipulés par les programmes qui manipulent les objets métier
• Pas de référence aux DAO par les objets métier
• Stratégie la plus souvent utilisée
- Perte de la pureté de la programmation OO



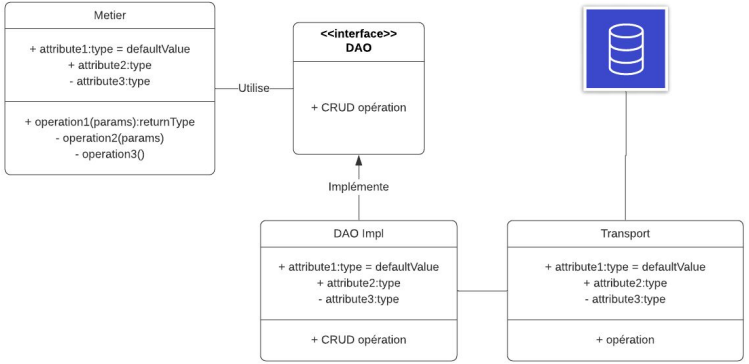
DAO et héritage:

Classe hérité => structure commune entre class mère et fille.

Plusieurs méthodes :
1. Faire correspondre toutes les classes de la hiérarchie à une seule relation de bases de données

2. Représenter chaque classe (abstraite ou concrète) par une relation

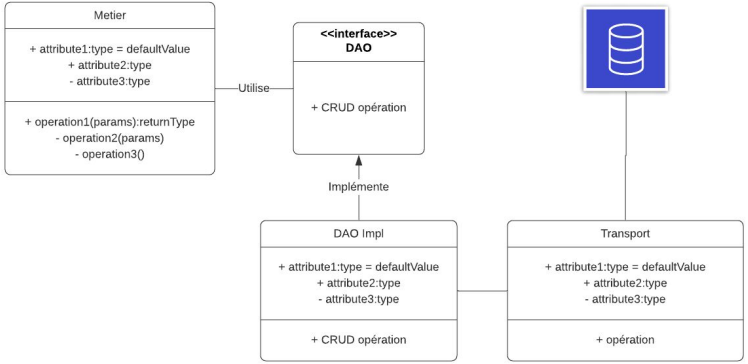
3. Représenter chaque classe concrète par une relation



DAO et héritage:

1. Faire correspondre toutes les classes de la hiérarchie à une seule relation de bases de données

- \+ Facile à mettre en place
- \- Obligation de gérer des valeurs NULL pour plusieurs colonnes
- \- Pas de possibilité de déclarer une contrainte NOT NULL sur une de ces colonnes même si la contrainte doit être vérifiée



DAO et héritage:

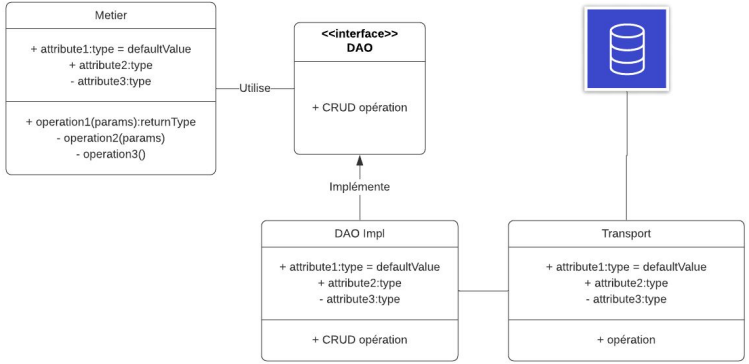
2. Représenter chaque classe (abstraite ou concrète) par une relation

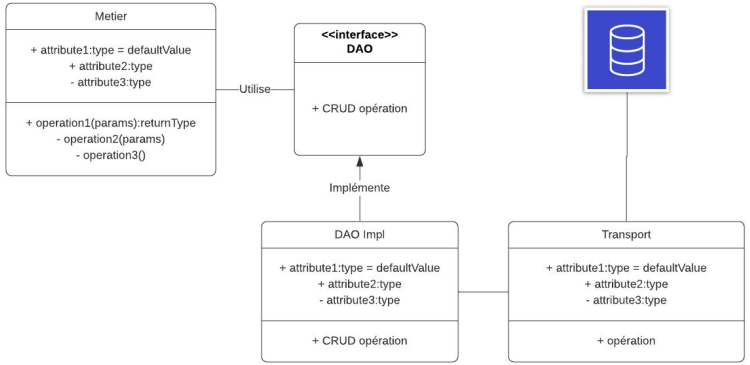
- Répartition des attributs d'un objet dans plusieurs relations
- Préservation de l'identité en donnant la même valeur de clé primaire à chaque nuplet correspondant à l'objet dans les différentes relations

\+ Simple bijection entre les classes et les relations
\- Nombreuses jointures à faire en cas de hiérarchie complexe

Possibilité de limiter certains problèmes en ajoutant des attributs dans les classes mères :

\- Problème de performances
\+ Vérification des contraintes d'intégrité





DAO et héritage:

3. Représenter chaque classe concrète par une relation

- Correspondance de chaque classe concrète avec une relation contenant tous les attributs (même les attributs hérités) de la classe
- En cas de classe concrète avec des classes filles :
  Clé primaire des relations correspondantes aux classes filles =clés étrangères faisant référence à la clé primaire de la relation correspondant à la classe mère

\+ Pas de jointure pour retrouver les informations

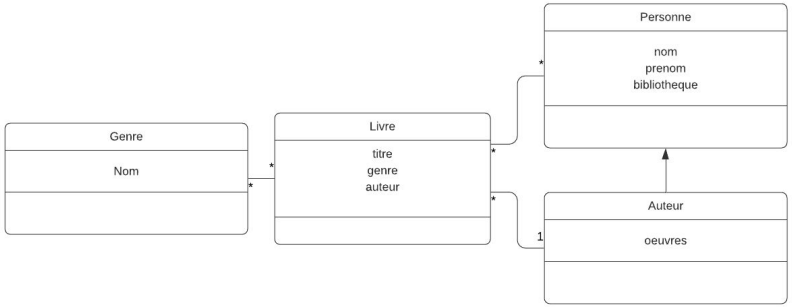
\- Problème pour les associations et requêtes polymorphes

\- Redondance d'information

EXERCICE

Créer une interface DAO.

- Gestion de livres suivant le schéma suivant ci-contre
- Opération CRUD sur chaque entité
- Persistance suivant modèle DAO
- Interface simple permettant d'enregistrer, consulter et supprimer les données



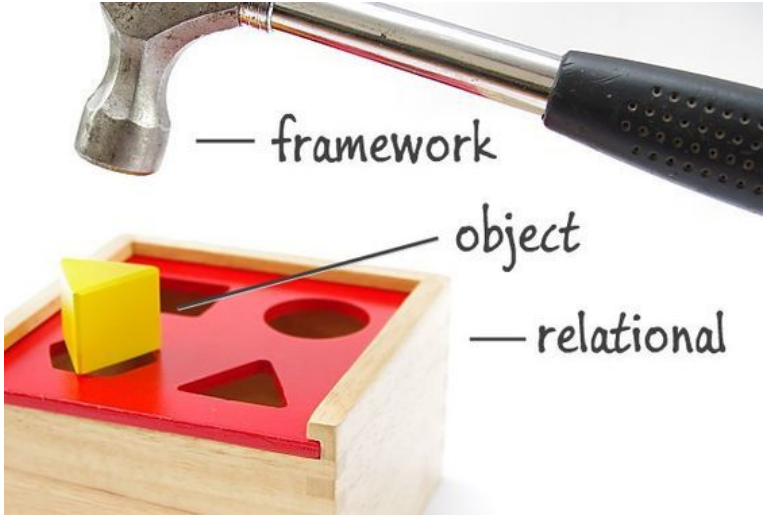
Les API ORM

Développement Front

API
ORM

API ORM => Bibliothèque qui s'interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet

+ Économie du temps de codage DAO
+ Aide de la communauté sur l'API
- Difficile de faire une requête "complexe" dans l'ORM
- Chaque ORM est différent => difficulté d'apprentissage/de changement



Développement Front

API
ORM

Chaque ORM est différent =>

- Java
  - Java Persistence API
    - TopLink
      - EclipseLink
    - Apache OpenJPA
  - Hibernate
  - Java Data Objects
  - Apache Cayenne
  - OJB - Object Relational Bridge
  - Apache Torque
  - SimpleORM
  - iBATIS
  - Avaje Ebean
  - Grails
  - AJO
- Python ...
- Node.js ...
- .Net
  - CodeFluent Entities
  - Entity Framework[1]
  - NHibernate
  - Linq To SQL (.Net Framework 3.5)
  - iBATIS
  - Euss (*Evaluant Universal Storage Services*)
  - MyGeneration/dOOdads
  - LayerCake Generator (.NET Framework 4.5) [archive]
- Ruby
  - Active record
  - RBatis, portage de iBATIS en Ruby pour Ruby on Rails
  - Sequel
  - DataMapper
- PHP 5
  - RedBean
  - Doctrine
  - Pdomap
  - Phpmyobject
  - CakePhp
  - Propel
  - FoxOrm
  - AgileToolKit
  - Syrius
  - Fuelphp
  - Eloquent

Développement Front

API
ORM

Exemple : RedBean / PHP

Library ORM pour PHP créer en 2009

+ Compatible avec les bases MySQL, MariaDB, PostgreSQL, SQLite ou CUBRID. D'autre bases sont disponibles avec des plugins tier
+ Rapide d'accès, pas de fichier XML de config
+ Création/Gestion de la DB à la volée, on ne gère que le code Objet PHP pas la base.
+ Tables, colonnes et type variable en fonction des besoins et des objets
+ Utilisation du SQL pur possible
- Pas/peu efficace sur les DB pré-existantes
- Gestion des ressources peut être gourmande à cause de la mutation "à la volée" de la base
- Pas de fichier de config => pas de nomenclature custom ni de mapping custom

Version PHP compatible : 5.3.4+ ou 7
Moteur PHP compatible : ZEND PHP et HHVM
Module pour différent framework PHP : Laravel, CodeIgniter, Kohana, Silex ou Zend Framework

lien : https://www.redbeanphp.com/index.php?p=/download

Développement Front

API
ORM

Prise en main rapide :

```php
require 'rb.php';//1
R::setup();//2
$photo = R::dispense( 'photo' );//3
$photo->title = 'Mes vacances';
$id = R::store( $photo );//4
$photo-copie = R::load( 'photo', $id );//5
R::trash( $photo-copie );//6
$photos = R::find(
   'photo', ' title LIKE ?', [ 'vacances' ] );//7
 R::close();//8
```

1. Télécharger l'API et inclure l'unique fichier PHP au projet
2. Créer une base SQLite temporaire
3. Créer un objet persistant "Photo"
4. Stocker l'objet en mémoire
5. Charger l'objet stocké précédemment
6. Supprimer l'objet de la DB
7. Rechercher les photos
8. Fermer la connexion

SIMPLE

Commande en détail

Connexion : `R::setup( 'sqlite:/tmp/dbfile.db' );`
Déconnexion : `R::close();`
Créer un “bean” : `$var = R::dispense( 'type' );`
Sauvegarder /Updater un bean : `$id = R::store( $var );`
Charger un bean : `$var = R::load( 'type', $id );`
Charger plusieurs beans : `$vars = R::loadAll( 'type', $ids );`
Delete un bean : `R::trash( $var );`
Delete plusieurs beans : `R::trashAll( $vars );`
SQL pour charger des beans : `$vars = R::findFromSQL('type',"SQL”);`
SQL pour charger des beans : `R::exec( 'UPDATE page SET title="test" WHERE id = 1' );`

Développement Front

API
ORM

Gestion des relations en détail

one-to-many:

```
$shop = R::dispense( 'shop' );
$shop->name = 'Antiques';

$vase = R::dispense( 'product' );
$vase->price = 25;
$shop->ownProductList[] = $vase
R::store( $shop );
```

```
$vase = R::dispense( 'product' );

$shop->ownProductList[] = $vase
```

many-to-many:

```
list($vase, $lamp) = R::dispense('product', 2);

$tag = R::dispense( 'tag' );
$tag->name = 'Art Deco';
$vase->sharedTagList[] = $tag;
$lamp->sharedTagList[] = $tag;
R::storeAll( [$vase, $lamp] );
```

```
$tag = R::dispense( 'tag' );

$vase->sharedTagList[] = $tag;
```

many-to-one :

```
$product->shop = $someShop;
R::store( $product );
```

La relation est défini par le “Owner” (shop).

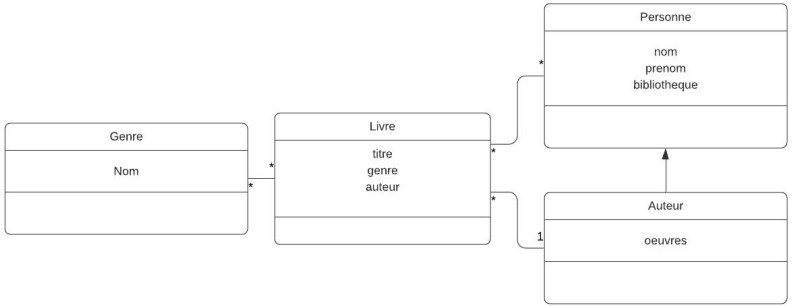
Création auto d’un champ shop_id dans product en DB

Création auto d’une table product_table en DB

EXERCICE

Créer un système de bibliothèque.

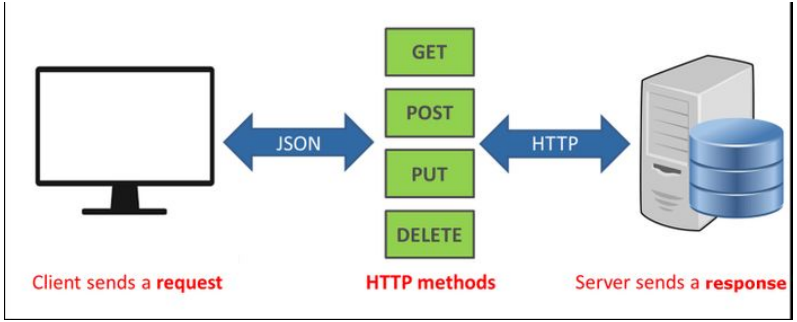
- Gestion de livres suivant le schéma suivant ci-contre
- Opération CRUD sur chaque entité
- Persistance suivant modèle DAO + API ORM
- Interface simple permettant d'enregistrer, consulter et supprimer les données



REST

Développement Front

API
REST



REST = REpresentational State Transfer

Protocole de WebServices

Basé sur HTTP

Utilisation des méthodes HTTP courantes : GET, POST, PUT, DELETE
Ou moins courantes : PATCH, HEAD

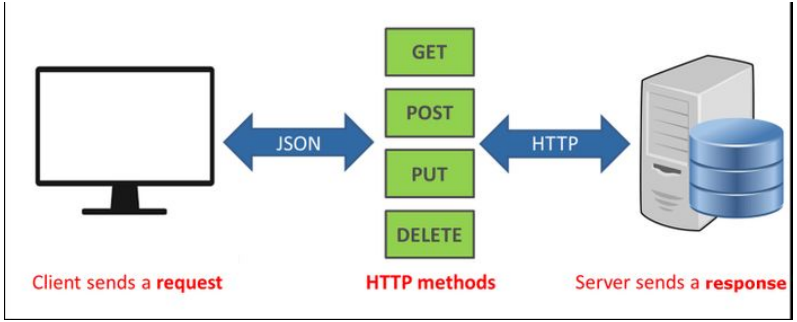
Réponses aux requêtes dans différents formats: CSV, JSON, RSS, etc

Modèle client/serveur (Node PHP Postman Python...)

Développement Front

API

REST



Méthodes HTTP

GET : Récupérer les ressources

POST : Insérer une ressources en DB

PUT : Modifier une ressource en DB

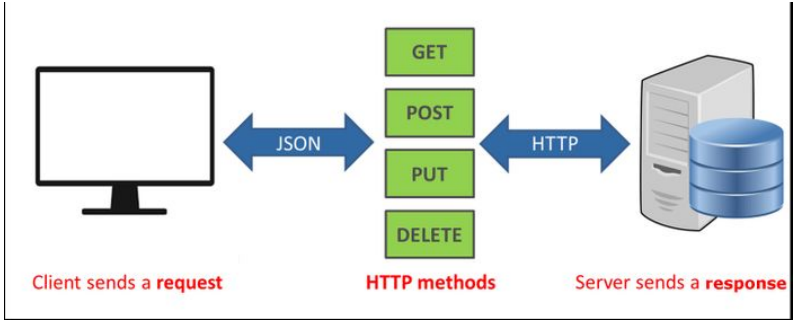
DELETE : Supprimer une ressource en DB

Routage :

Associer une URI à une ressource

exemple :

- http://myTestServeur.fr/shop
- http://myTestServeur.fr/shop/15
- http://myTestServeur.fr/product
- http://myTestServeur.fr/product/24



Exemple en PHP : API Bibliothèque

architecture

- **index.php**: Ce fichier est un fichier d'entrée(vide). Ce fichier empêche la navigation dans les fichiers de répertoire.
- **db_connect.php**: Ce fichier utilisera la chaîne de connexion mysql.
- **livres.php**: Ce fichier contient tous les méthode d'API REST.
- **.htaccess**: Ce fichier est utilisé pour le routage.

fichier produits.php

```php
<?php
    // Connect to database
    include("db_connect.php");
    $request_method = $_SERVER["REQUEST_METHOD"];

    function getProducts()
    {
        global $conn;
        $query = "SELECT * FROM produit";
        $response = array();
        $result = mysqli_query($conn, $query);
        while($row = mysqli_fetch_array($result))
        {
            $response[] = $row;
        }
        header('Content-Type: application/json');
        echo json_encode($response, JSON_PRETTY_PRINT);
    }

switch($request_method)
    {
        case 'GET':
            getProducts();
            break;
        default:
            // Invalid Request Method
            header("HTTP/1.0 405 Method Not Allowed");
            break;
    }
?>
```

fichier db_connect.php

```
<?php
    $server = "localhost";
    $username = "root";
    $password = "";
    $db = "stock";
    $conn = mysqli_connect($server, $username, $password, $db);
?>
```

fichier .htaccess

```
RewriteEngine On # Activer le module Rewrite
RewriteRule ^produits/?$ produits.php [NC,L]
RewriteRule ^[^/]+/(\d+)$ produits.php?id=$1
```

Ces règles de routage ne sont pas nécessaires mais facilitent la lecture des URIs

EXERCICE

Ajouter les services associés aux méthodes GET, PUT, POST et DELETE à votre application de Bibliothèque.