1. Persistance des données

1.1 généralités

- Les données sont une composante fondamentale de de tout projet informatique.
- Elles correspondent aux informations qui doivent être conservées d'une session à l'autre.

- Exemples :
  - Les données liées à l'utilisateur (données personnelles, sauvegardes,...)
  - Les données de l'entreprise:
    - données comptables
    - ventes
    - achats
    - employés
    - stocks
    - production
    - etc..
  - Bases d'information:
    - Documents en ligne
    - Services (trajets SNCF, articles commerciaux à vendre, données météo, articles de presse...)
  - Échanges et communication
    - index web (moteurs de recherche)
    - graphes de liens et messages (réseaux sociaux)
    - ...

Donnée informatique

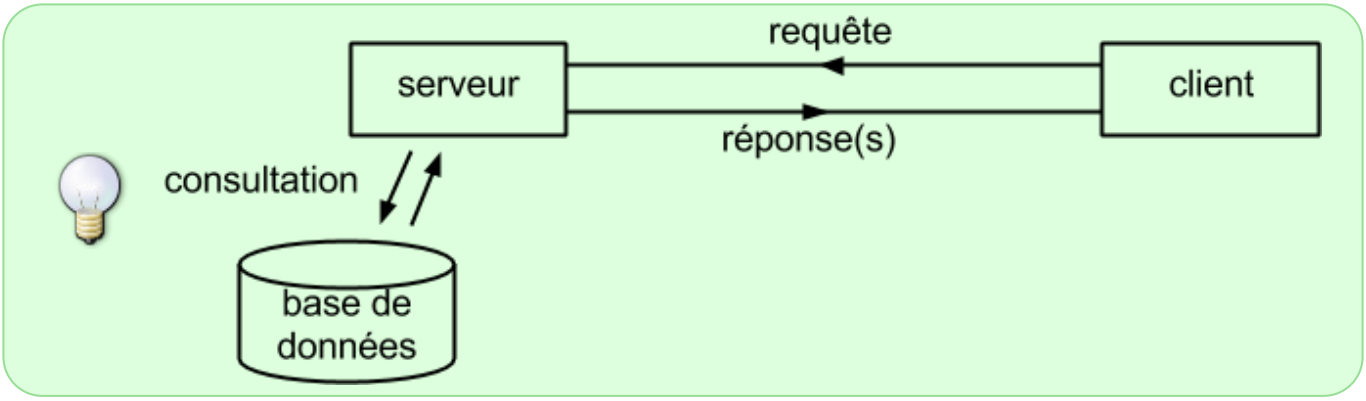
- Une données informatique est un élément d'information ayant subi un encodage numérique
  - Consultable/manipulable/échangeable par des programmes informatiques
  - Possibilité de la conserver sur un support de stockage numérique (CD-ROM, dique dur, SSD, ...)
- Les informations peuvent être stockés dans un fichier (ex : fichier csv).
- La plupart du temps, on utilisera des bases de données :
  - plus robuste
  - plus sécurisé
  - plus rapide

Serveur de bases des données

En informatique, une requête (en anglais query) est une demande de consultation, effectuée par un programme *client* à l'attention d'un programme *serveur*.

- Le programme **client** représente l'utilisateur, il s'agit du programme qui enregistre la demande

de l'utilisateur, la transmet au serveur, puis met en forme visuellement la réponse du serveur.
- Les données sont centralisées au niveau du **serveur**, chargé de la gestion, de la manipulation et du stockage des données.Il traite la requête, consulte les données et transmet le résultat au client.



La requête peut être une simple référence vers un fichier, ou être l'expression d'une recherche plus spécifique (consultation de certaines fiches d'un fichier, croisement d'information (entre plusieurs fichiers), etc...). Dans ce cas, il est nécessaire d'utiliser un langage de requête (le plus souvent SQL).

Lors d'une consultation de type lecture/recherche, il y a souvent plusieurs réponses qui correspondent à la demande. Le résultat d'une requête prend donc la forme d'un ensemble de réponses. Ces réponses sont éventuellement classées, selon la valeur d'un certain identifiant, ou selon le degré de pertinence.

Exemples :

- requêtes http : demande de consultation d'une page web (= référence vers un fichier)
- moteur de recherche : recherche de pages contenant les mots-clés spécifiés
- bases de données : utilisation d'un langage de requête :

```
SELECT *
FROM  Eleves
WHERE NOM = 'Dugenou'
```

1.2. La mémoire cache

Rappel sur les fichiers

- La mémoire secondaire est organisée sous forme de *pages* (ou *secteurs* : blocs de 512 octets environ)
- Les programmes ne peuvent pas directement écrire sur les secteurs. Le système d'exploitation assure la gestion la gestion de l'accès au disque via *les fichiers*
- Un fichier est une entité logique représentant un ensemble de pages de la mémoire secondaire
- Il est désigné par son *descripteur* (nom, chemin d'accès, droits,...)

Lecture

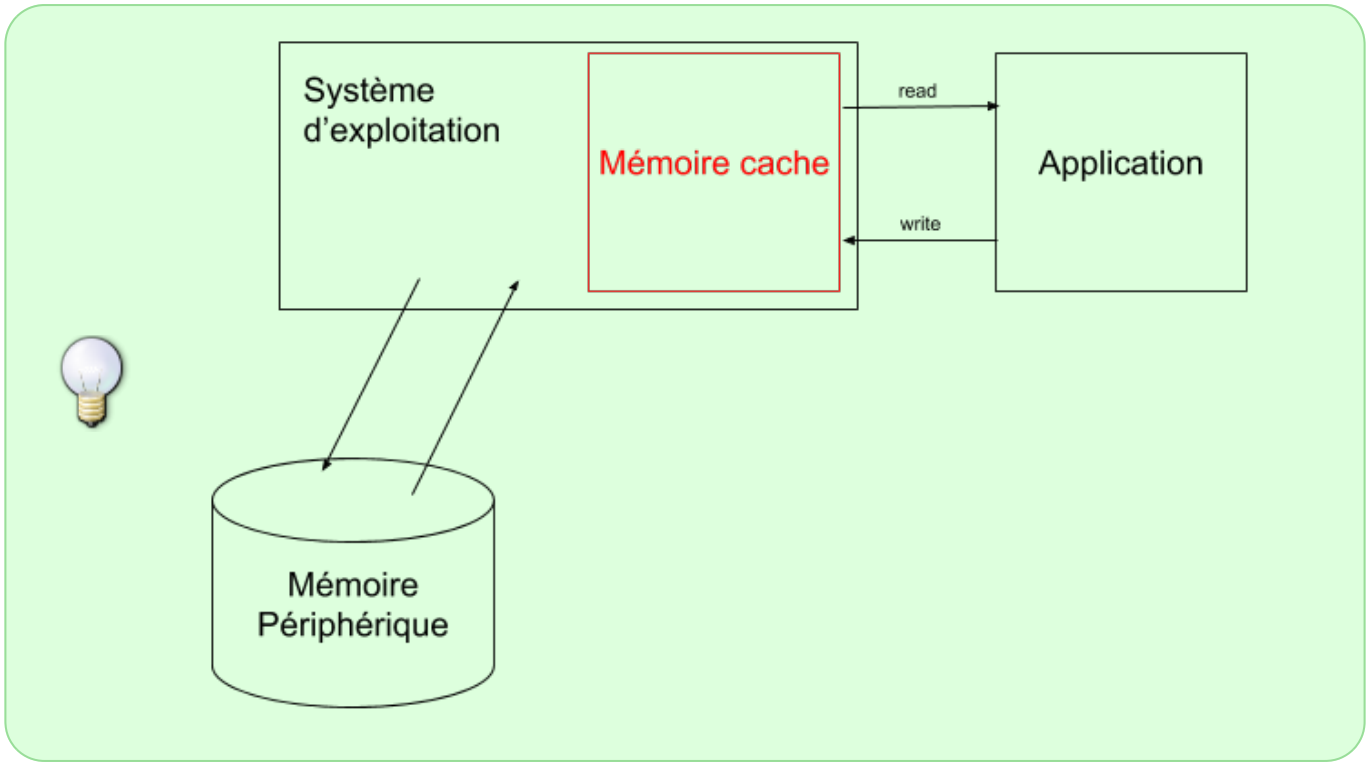
- ouverture d'un fichier = initialisation d'un *descripteur de fichier* `f`:

```
f = open("/chemin/vers/mon_fichier", "r")
```

* `f` est un objet qui implémente un *flux de données*. * Un flux de données est structure d'accès :

- sans adressage
- les données sont lues dans l'ordre dans lequel elles ont été écrites par le serveur

* On parle d'accès *séquentiel* aux données




```
s = f.readline()
```

- L'opération de lecture dépile l'élément situé en tête de flux (dans la mémoire cache) et le retourne à l'utilisateur

SCHEMA : TODO

* Le système d'exploitation se charge de gérer la mémoire cache :

- les donnée sont lues en fonction des demandes du programme
- typiquement le système charge plusieurs pages en avance dans la mémoire cache

Ecriture

```
g = open("/chemin/vers/mon_fichier", "w")
```

autres possibilités :

- Lecture/écriture : "rw"
- Ajout : "a"

```
g.write(s)
```

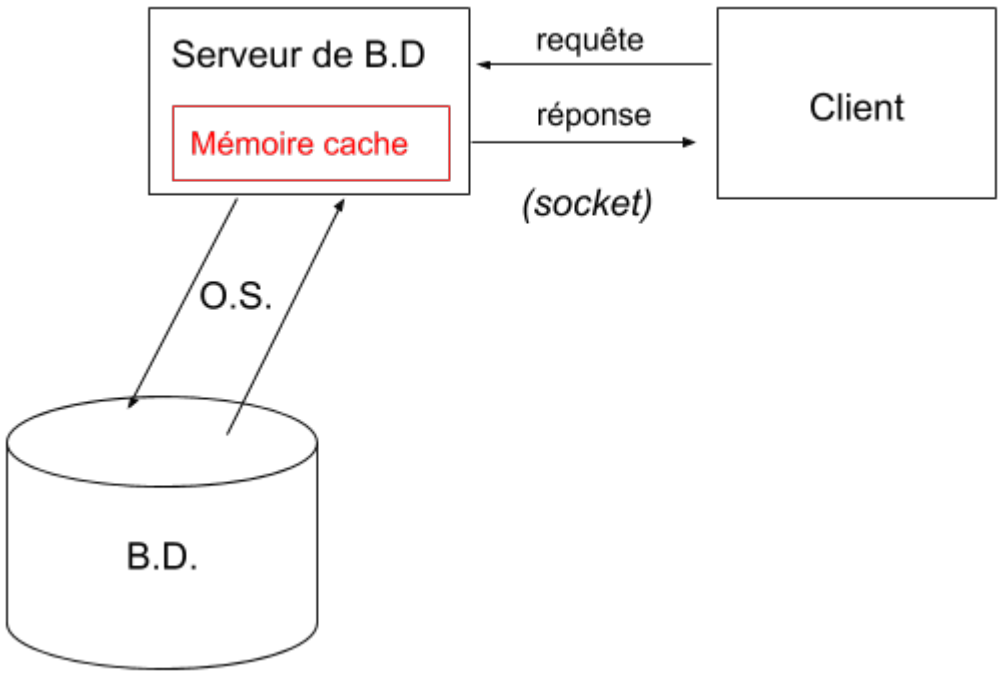
- Le système d'exploitation modifie le cache sans modifier directement/immédiatement le fichier
- L'écriture dans la mémoire secondaire est garanties lors de la fermeture :

```
g.close()
```

- Pour forcer l'écriture sans fermer le fichier, on on effectue un "vidage" du cache dans la mémoire secondaire

```
g.flush()
```

Accès aux bases de données




```
import sqlite3
db = sqlite3.connect("/chemin/vers/mabase.db")
```

- Ici l'objet db ouvre une communication (un "*socket*") avec un programme de gestion de bases de données (ici l'application `sqlite3`).
  - Le programme n'est donc plus en communication directe avec le système d'exploitation mais avec une application tierce
  - C'est maintenant sqlite3 qui gère le flux de données.
- Le curseur est l'objet qui gère l'accès séquentiel aux données:

```
c = db.cursor()
```

- Le programme sqlite3 exécute la requête et génère un flux de données:

```
c.execute("SELECT * FROM MaTable")
```

- Lecture de la première réponse (du premier tuple) du flux:

```
t = c.fetchone()
```

- Accès en écriture:

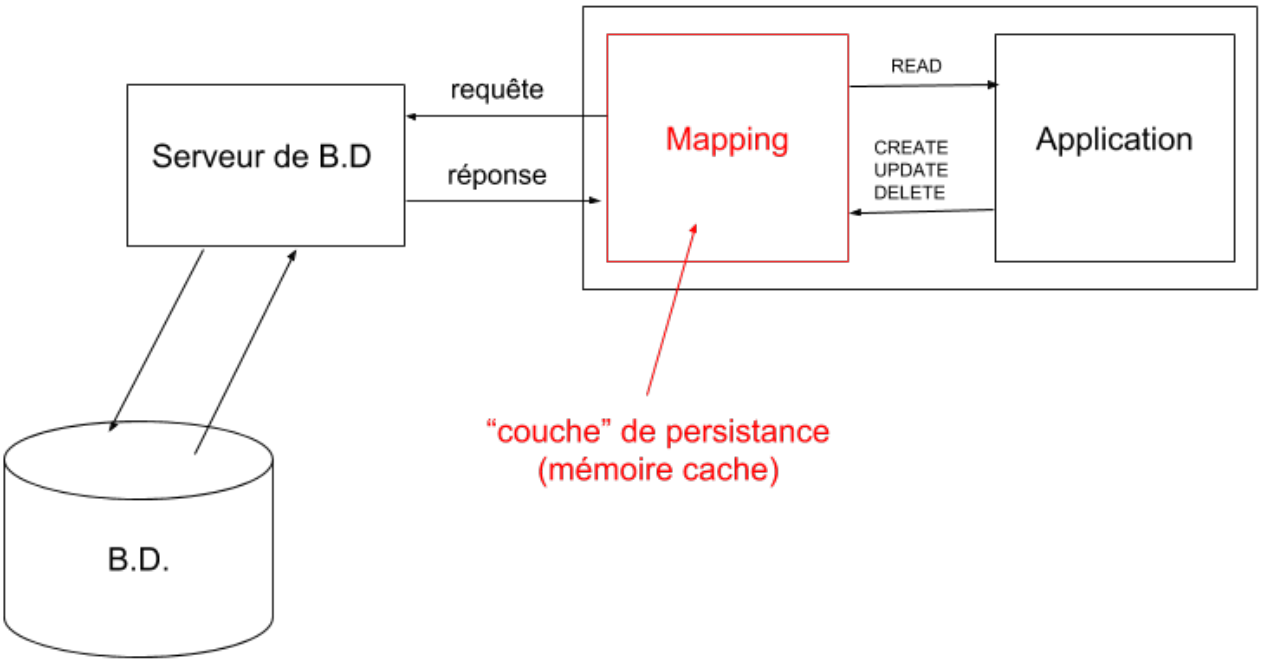
```
c.execute("INSERT INTO MaTable VALUES (v1, v2, v3)")
```

- Les modifications sont conservées dans la mémoire cache de sqlite. Néanmoins, pour s'assurer d'un enregistrement effectif en mémoire secondaire, il faut effectuer un `commit` (pour forcer l'écriture)

```
db.commit()
```

Appariement (Mapping) Objet/Données

- Un appariement (*Mapping*) est une couche d'abstraction logicielle qui permet :
  - de manipuler les données de la base données de manière plus conviviale et intuitive,
  - essentiellement en faisant correspondre les tuples (enregistrements) de la base de données avec des variables du programme
    - correspondance objets/données
    - On parle aussi d'interface objets/données
- En particulier, l'utilisateur n'a pas besoin d'écrire de requêtes SQL pour modifier le contenu de la base : le SQL est "caché"...
- Le schéma vu par l'utilisateur peut être différent du schéma de la base de données



Voir aussi : transparent_persistence.html

Exemples

- "Vues" d'une base de données : tables virtuelles correspondant à des requêtes pré-définies –> en RAM
- Mapping objet/Relationnel (ORM – *Object-Relational Mapping*) :
  - mise en correspondance table/classe
  - + ajout de getters/setters
  - patron de conception DAO (*Data Access Object*)
- Approche “CRUD”: les requêtes se réduisent à quatre grandes familles d'opérations:
  - **Création** (***C****reate*) : ajout de nouvelles données dans la base
  - **Lecture/recherche** (***R****ead*) : consulation du contenu de la base
  - **Mise à jour** (***U****pdate*) : changement du contenu existant
  - **Suppression** (***D****elete*) : suppression des données obsolètes
- Analyse de données :
  - Mise en forme *hiérarchique* des données :
    - Dimensions (indexation multiple hiérarchique)
    - Agrégation multi-critères (tableaux croisés dynamiques, cubes de données etc.)
  - Visualisation

Dans les deux premiers cas que nous avons vus, la gestion de la mémoire cache est déléguée à des programmes tiers:

- Les système d'exploitation (pour la gestion des fichiers)
- Le gestionnaire de BD (requêtes vers une BD)

Dans le cas du mapping Objet/Relationnel, la couche de persistance gère le chargement en mémoire des données de la base.

Elle joue le même rôle que la mémoire cache:

- en maintenant en mémoire l'état des objets modifiés par l'utilisateur
- en chargeant de manière parcimonieuse le contenu de la base (seuls les données réellement utilisées doivent être chargées en mémoire)

1.3 De la conception à la réalisation

Principe général : ***Retrouver le programme dans l'état dans lequel on l'a laissé lorsqu'on l'a précédemment quitté***:

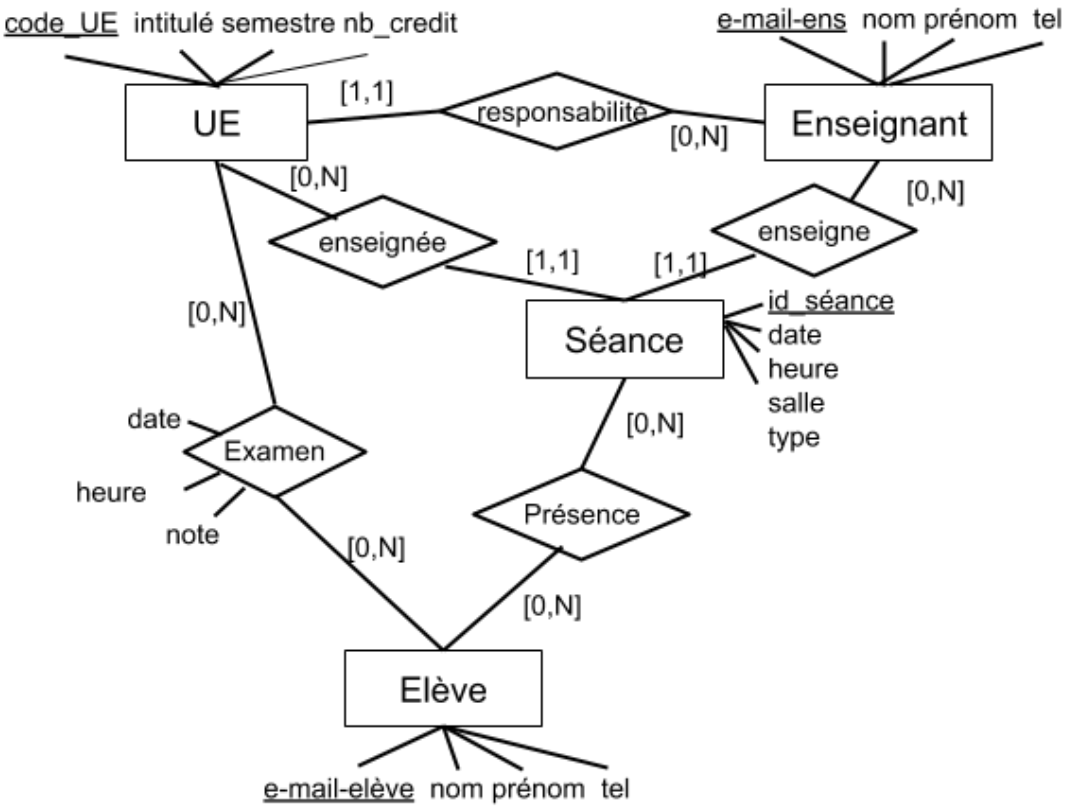
- Les variables et objets manipulé sont régulièrement sauvegardés
- Pour plus d'efficacité, le fichier de sauvegarde est une base de données
- Au niveau de la conception du programme, on doit distinguer les *données persistances* des *données non-persistantes*
- Mise en correspondance et synchronisation entre les donnée du programme et

la base de données
- Maintien de la cohésion assuré par la mémoire cache (interface entre application et SGBD)

Les données persistantes sont conservées dans une base de données relationnelle dont le schéma est établi au cours de l'étape de *modélisation*.

- Conception initiale sous forme d'un modèle Entité/Association (cours de première année)
- Passage au modèle relationnel (cours de première année)
- Modélisation UML

Modèle Entité/Association



Passage au modèle Relationnel

- **Enseignant**(e-mail-ens, nom, prénom, tel)
- **UE**(code_UE, intitulé, semestre, nb_crédits, e-mail-ens)
- **Séance**(id_séance, code_UE, e-mail-ens, type, salle, date, heure)
- **Présence**(e-mail-eleve, id_seance)
- **Elève**(e-mail-eleve, nom, prénom, tel)

- **Examen**(code_UE, e-mail-eleve, date, heure, note)

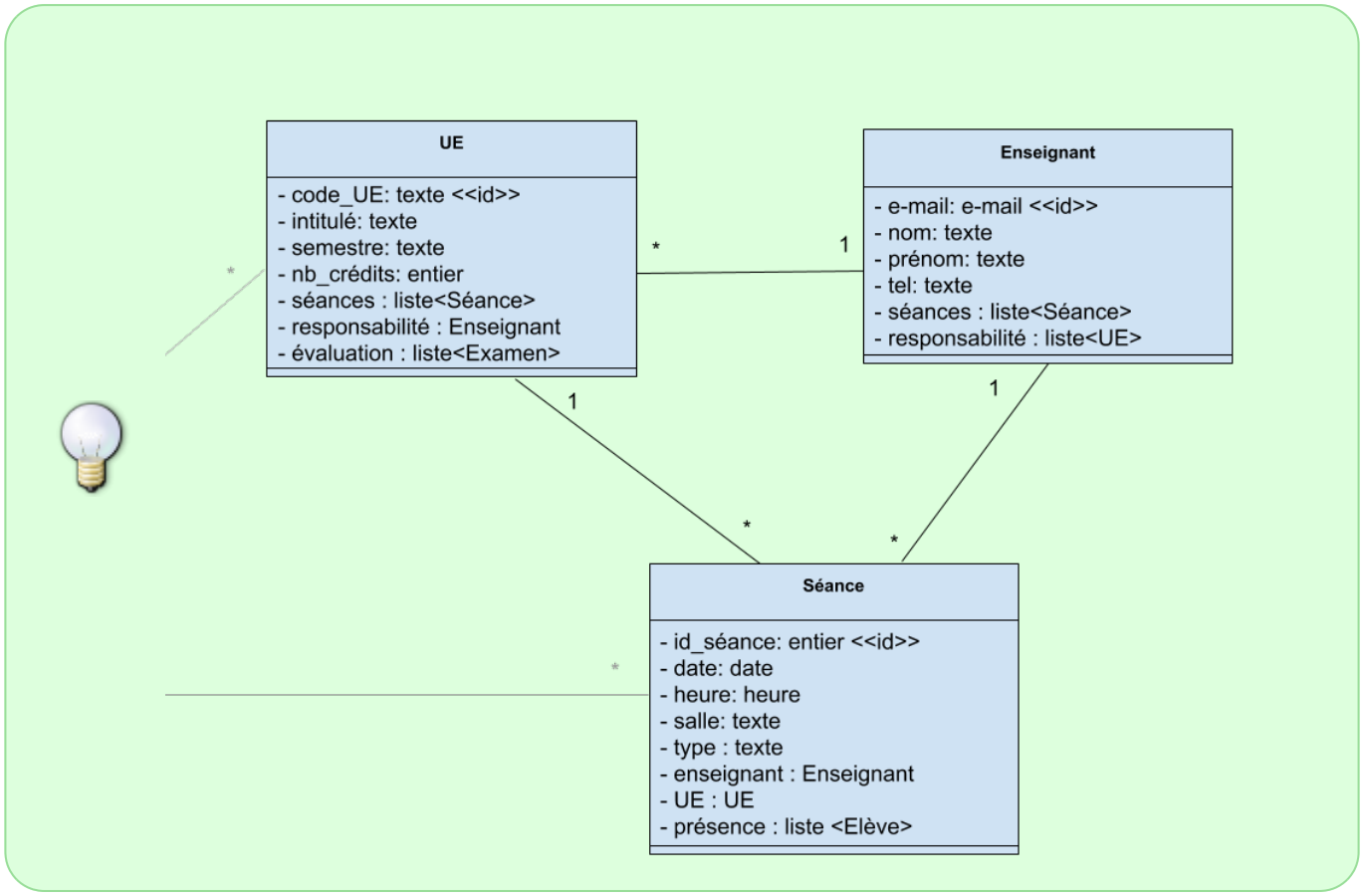
Création de tables SQL

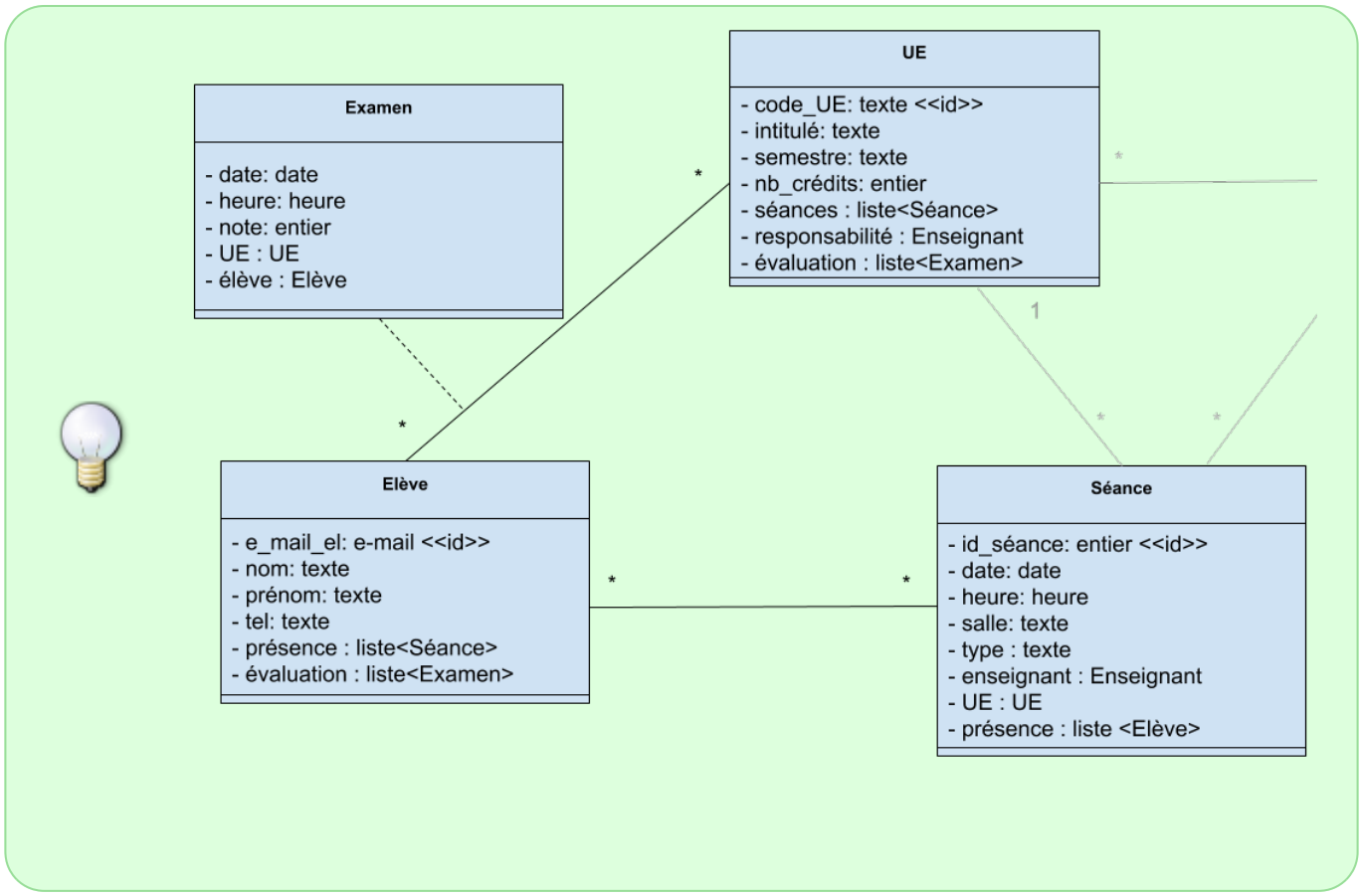
```
CREATE TABLE Enseignant (
  e_mail_ens VARCHAR(30) NOT NULL,
  nom VARCHAR(30) NOT NULL,
  prénom VARCHAR(30) NOT NULL,
  tel VARCHAR(12),
  PRIMARY KEY (e_mail_ens));
```

```
CREATE TABLE UE (
  code_UE VARCHAR(30) NOT NULL,
  intitulé VARCHAR(30) NOT NULL,
  semestre INTEGER NOT NULL,
  nb_crédits INTEGER NOT NULL,
  e_mail_ens VARCHAR(30) NOT NULL,
  PRIMARY KEY (code_UE),
  FOREIGN KEY (e_mail_ens) REFERENCES Enseignant);
```

etc...

Passage au modèle UML





Réalisation en Python

Pour chaque Entité une classe distincte.

```
class Enseignant:
    def __init__(self, e_mail_ens , nom, prénom, tel):
        self.e_mail_ens = e_mail_ens
        self.nom = nom
        self.prénom = prénom
        self.tel = tel
        self.responsabilités = set()
        self.séances = set()

    def ajoute_seance(self, séance):
        self.séances.add(séance)

    def ajoute_responsabilité(self, UE):
        self.responsabilités.add(UE)
```

etc...

Mise en œuvre de la persistance : patron DAO

Un DAO (Data Access Object) est une classe qui réalise l'interface entre une classe d'objets

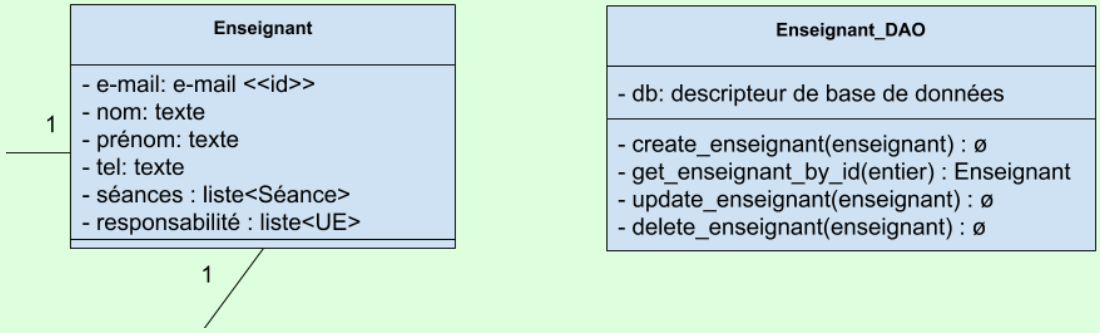
persistants et la base de données.

- Une classe DAO permet de mettre en œuvre les quatre opérations de base :
  - Create
  - Read
  - Update
  - Delete
- Il existe autant de classes DAO que de classes persistantes

Dans l'exemple considéré, on doit donc avoir les six classes suivantes:

- **Enseignant_DAO**
- **UE_DAO**
- **Séance_DAO**
- **Elève_DAO**
- **Examen_DAO**

Il est possible de mettre en œuvre un patron de conception *Factory* permettant de gérer les différentes interfaces au sein d'une même classe en évitant la répétition de code



En Python:

```
class Enseignant_DAO:
    def __init__(self, db_name):
        self.db = sqlite3.connect(db_name)
    def create_enseignant(self, enseignant):
        ...
    def get_enseignant_by_id(self, id_enseignant): # Read
        ...
    def update_enseignant(self, enseignant):
        ...
    def delete_enseignant(self, enseignant):
        ...
```

Remarque : l'opération de lecture `get_enseignant_by_id` effectue les opérations suivantes:

- extraction des données de l'enseignant dans la table `Enseignant`:

```
c.execute("SELECT * FROM Enseignant WHERE e_mail_ens = ?",(id_enseignant,))
```

- initialisation un objet `enseignant` (à l'aide du constructeur de la classe `Enseignant`)
- recherche des séances programmées dans la table Séances:

```
c.execute("SELECT id_séance FROM Séance WHERE e_mail_ens = ?",(id_enseignant,))
```

- pour chaque `id_séance` trouvé,
  - initialise un objet de type `Séance` avec la méthode `get_séance_by_id` de la classe `Séance_DAO`
  - ajoute la séance dans l'ensemble `séances` avec la méthode `ajoute_seance` de l'objet `enseignant`
- recherche des responsabilités d'UE dans la table UE:

```
c.execute("SELECT code_UE FROM UE WHERE e_mail_ens = ?",(id_enseignant,))
```

- pour chaque `code_UE` trouvé,
  - initialise un objet de type `UE` avec la méthode `get_UE_by_id` de la classe `UE_DAO`
  - ajoute l'UE dans l'ensemble `responsabilités` avec la méthode `ajoute_UE` de l'objet `enseignant`
- retourne l'objet `enseignant`

Problème : avec le patron DAO, l'existence de relations many-to-many a pour effet de charger l'intégralité des tables concernées en mémoire lors de la lecture d'un objet unique!!

Exemple : relation many-to-many entre Elève et Séance :

- La méthode `get_séance_by_id` fait appel à `get_eleve_by_id` pour établir la liste de présence
- La méthode `get_eleve_by_id` fait appel à `get_séance_by_id` pour établir la liste des séances auxquelles l'élève a assisté
- etc....

Il faut donc prévoir de ne charger qu'une partie des informations, celle qui est réellement utile au programme (inutile de charger l'emploi du temps de chaque élève lorsqu'on s'intéresse à la liste de présence d'une séance particulière).

- Avec le patron DAO, il faut gérer au cas par cas
- Les *Gestionnaires de persistance* ORM (django, Pony ORM) permettent de gérer le chargement des données "à la demande", à la manière de la mémoire cache.

Voir aussi : dao-et-orm-sont-ils-compatibles

Un gestionnaire de persistance : la librairie Pony ORM

- Pony ORM

voir notebook (nbviewer)

2. Le patron MVC (Modèle-Vue-Contrôleur)

Le patron de conception "Modèle - Vue - Contrôleur" est destiné à faciliter le développement d'interfaces graphiques.

Une Interface graphique est constituée essentiellement de deux modules :

- Le "*front-end*" (la "devanture"), qui est la partie du programme visible pour l'utilisateur et avec laquelle l'utilisateur peut interagir à l'aide de menus, de boutons et de formulaires...
- Le "*back-end*" (l'"arrière-boutique"), qui correspond aux rouages invisible à l'utilisateur, permettant au programme de réaliser la tâche pour laquelle il a été conçu.

Pour développer un tel programme, on le divise généralement en trois modules appelés respectivement:

- le Modèle
- la Vue
- le Contrôleur

Le Modèle

Le modèle est la partie du programme qui manipule et met à jour les informations qui doivent être conservées d'une session à l'autre. Il s'agit de l'ensemble des variables et objets qui sont créés et mis à jour par l'utilisateur lorsqu'il interagit avec le programme.

La Vue

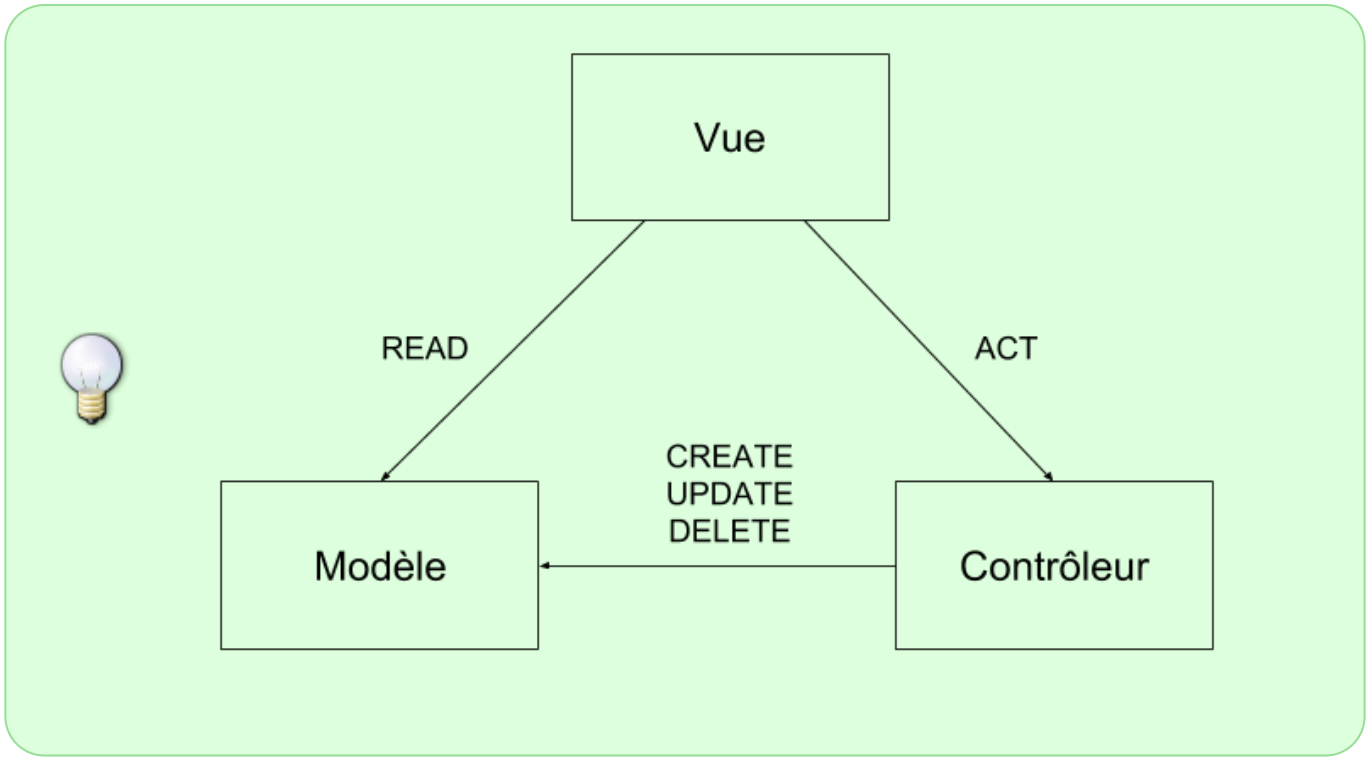
La Vue est la partie du programme qui gère la mise en page, la disposition des informations, des boutons et des formulaires, l'organisation et la visibilité des différentes fenêtres du programme s'il y en a.

- La Vue fait appel au Contrôleur à chaque fois que l'utilisateur effectue une action (saisie d'information, pointage de souris, sélection de menu, activation d'un bouton etc...)
- La Vue fait appel au Modèle pour afficher en permanence un contenu actualisé par les actions de l'utilisateur.

Le Contrôleur

Le contrôleur est la partie du programme qui gère les actions de l'utilisateur. Chacune des actions proposées dans la vue est implémentée dans le contrôleur sous la forme d'une fonction.

- Le contrôleur fait appel au Modèle lorsque l'action modifie les variables et objets manipulés par le programme.



Voir aussi

3. Développement Web

Le Web est basé sur trois piliers :

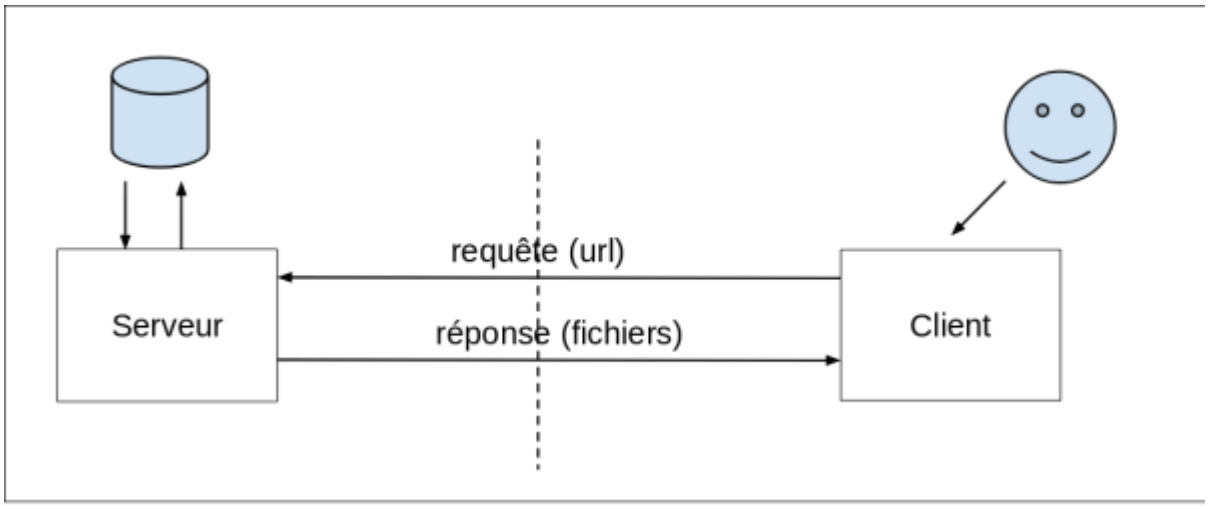
- Les liens hypertextes sur lesquels on peut cliquer et leur protocole (HTTP)
- le système d'adresses (URL)
- le langage de construction des pages (HTML)

- Le Web englobe les sites qui peuvent être consultés dans un navigateur, et n'est qu'une des briques de l'Internet
- Parmi les autres :
  - le courrier électronique
  - les applications mobiles
  - etc.

(source:Libération du 19/01/2011)

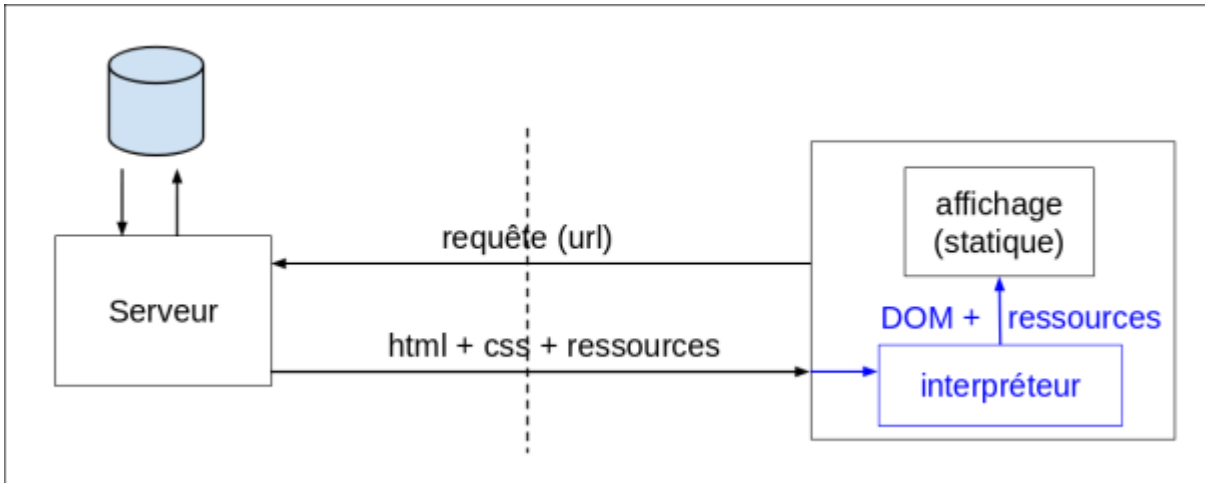
3.1 Généralités

Client/Serveur



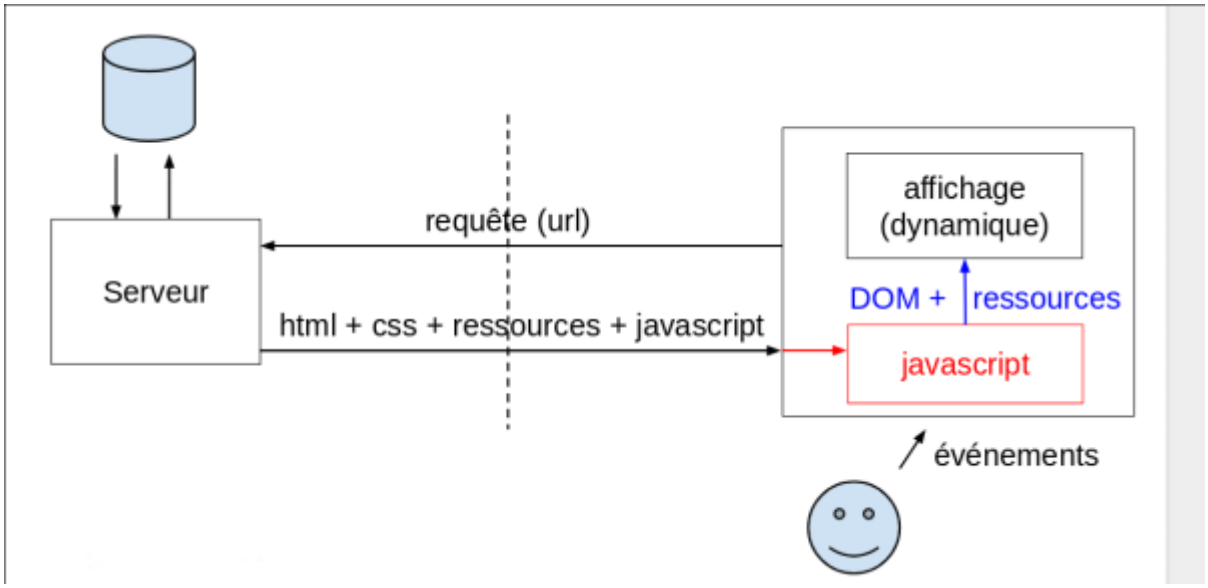
- url = adresse IP + /chemin/vers/fichier
- réponse = fichier (lu sur le DD du serveur)
- Le client gère la mise en page.

HTML + CSS



- côté client :
  - construction de l'arbre DOM ⇒ bloc = objet
  - affichage (flux)

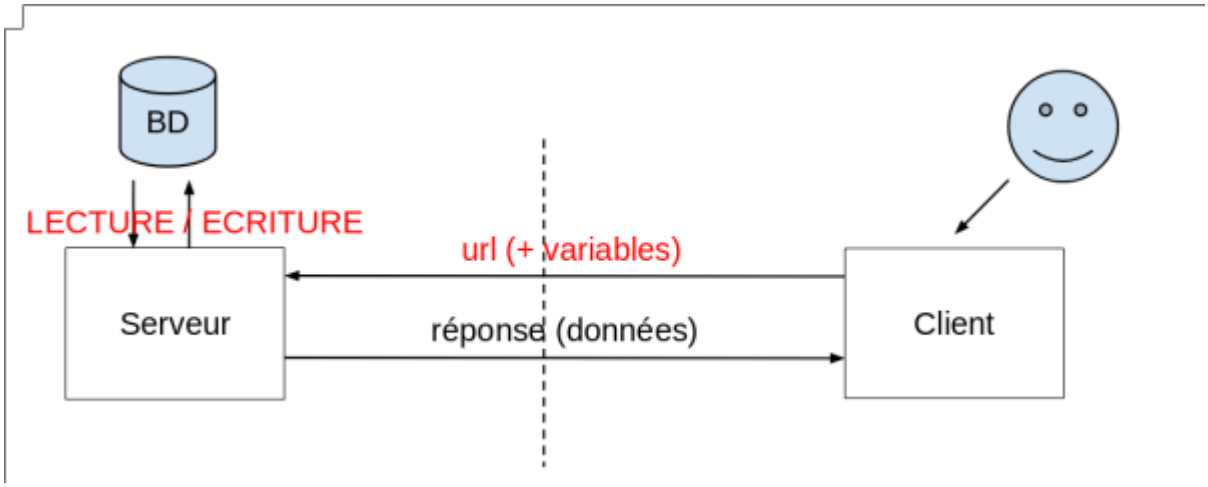
Pages dynamiques



- côté client :
  - modification du DOM selon les événements produits par l'utilisateur

Web dynamique

Principe général : consultation/mise à jour à distance d'une base de données.



- CRUD : Create / Read / Update / Delete
- réponse = données mises en forme au niveau du serveur

Remarque : 3 langages sont nécessaires pour réaliser ce schéma:

- Un langage d'édition de pages Web (interprété côté client):
  - en général langage HTML (ou HTML + CSS)
  - agrémenté de différentes librairies de mise en forme:
    - Bootstrap
    - Materialize
  - Les seuls langages compréhensibles pour le navigateur sont HTML et CSS et `Javascript`
- Un langage de développement (interprété côté serveur)
  - Le choix est vaste (n'importe quel langage de programmation)

- Les plus courants sont (par ordre de popularité):
  - PHP
  - Java (avec la librairie `Spring MVC`)
  - Python (avec la librairie `Django`)
  - Javascript (librairie `Node.js`)
  - etc.
- Un langage de requêtes (interprété côté serveur):
  - pour communiquer avec la base de données et enregistrer les mise à jour
  - Le SQL dans 90% des cas
  - Mais d'autres alternatives sont possibles (*NoSQL*): MongoDB, etc...

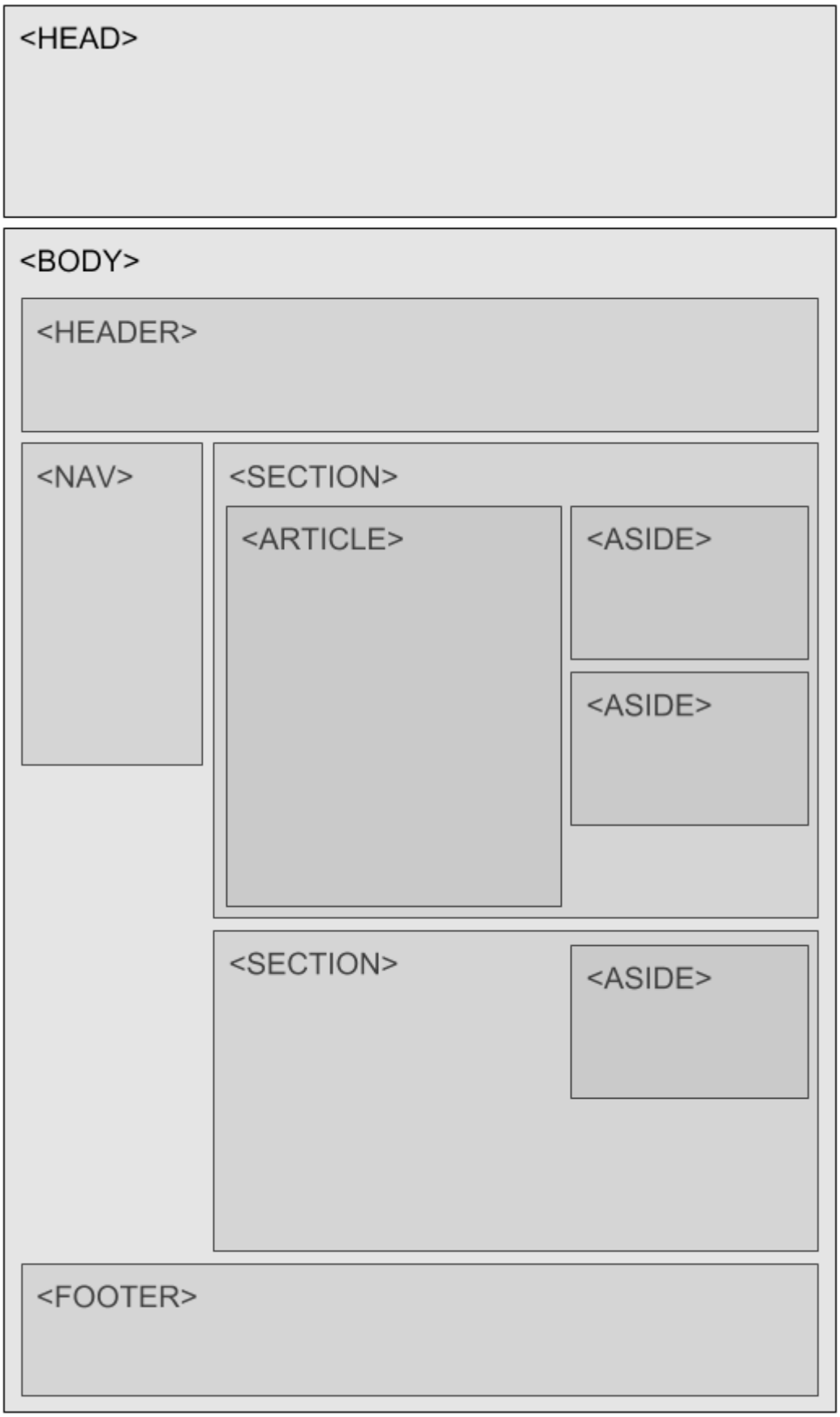
3.2 HTML

- HTML = "*Hypertext Markup Language*"
- But de ce langage : produire des documents consultables à distance par de navigateurs Web
- Un système de balises ("*Markups*") permet de décrire la mise en forme du document
- Possibilité de définir des *liens hypertexte* : chaque lien permet d'accéder à de nouveaux documents
- Possibilité de définir des *formulaires* qui permettent à l'utilisateur d'envoyer des informations vers le serveur

Un document HTML contient deux parties :

- L'**en-tête**, qui contient des contient des informations relatives à l'auteur et au contenu du document
  - Balise <HEAD>
- Le **corps**, qui contient le texte et les médias à afficher, avec des indications de mise en page
  - Balise <BODY>

Structure d'une page en HTML 5



De nombreuses ressources web sont disponibles pour apprendre le HTML, voir par exemple html

Exemple

emmanuel.dauce.free.fr

Aide-mémoire

- **En-tête**:
  - `<head>...</head>`
  - `<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />`
  - `<title>...</title>`
  - `<link rel="stylesheet" href="style.css" type="text/css" />`
- **Corps**:
  - *Bloc*:
    - `<body>...</body>`
    - `<p>...</p>`
    - `<h1>...</h1>`, `<h6>...</h6>`
    - `<div>...</div>`
  - *Flux*:
    - `<br />`
    - `<hr />`
    - `<em>...</em>`, `<strong>...</strong>`
    - `<span>...</span>`
  - *Listes*:
    - `<ol><li>...</li>...</ol>`
    - `<ul><li>...</li>...</ul>`
    - `<dl><dt>...</dt><dd>...</dd>...</dl>`
  - *Tables*:
    - `<table>...</table>`
    - `...`
    - `<tr>...</tr>`
    - `<th>...</th>`
    - `<td>...</td>`
  - *Images*:
    - `<img src="image.png" alt ="texte alternatif " />`
  - *Liens et ancres*:
    - `<a href="http://www.cnrs.fr/">...</a>`
    - `<baliseid="toto">...</balise>`
    - `<a href="#toto">...</a>`
  - *Attributs communs à toutes les balises*:
    - `style`
    - `class`
    - `id`

3.3 CSS

voir :

- Débuter avec les feuilles de style
- CSS Zen Garden:
  - html
  - css

3.4 Transfert de données

Transmission de données "en clair"

Variables GET :

- Les variables sont inscrites dans l'URL transmise au serveur:

```
http://mon.adresse.com/mon_site.php?nom=Pignon&prenom=Francois
     A D R E S S E      RESSOURCE       V A R I A B L E S
```

Le serveur exécute le script (php, python, java, ...), c'est à dire :

- traite les variables
- exécute des opérations de lecture/écriture
- transmet un contenu au client (en général html mais aussi xml ou json...)

Transmission de données par formulaire

Variables POST (n'apparaissent pas dans l'URL)

- utilisation de Formulaires HTML :
- le fichier cible est défini comme attribut du formulaire :

```
<FORM method="post" action="cible.php">
    ...
</FORM>
```

les balises INPUT définissent les variables à transmettre:

```
<INPUT type="text" name="var1" />
```

l'INPUT de type submit lance la requête : la page cible est chargée et remplace la page courante:

```
<INPUT type="submit" value="Envoyer" />
```

Exemple

- formulaire.html :

```
<form action = "bonjour.php" method ="post">
Votre nom : <input type="text" name="nom"/> <br/>
Votre prénom : <input type="text" name="prenom"/> <br/>
<input type="submit" value="Envoyer">
</form>
```

- bonjour.php :

```
echo "Bonjour, ".$_POST["prenom"]." ".$_POST["nom"]." !!!";
```

Aide-mémoire formulaires

- `<form method="post" action="script.php" enctype="multipart/form-data">`
  - ...
- `</form>`

(enctype peut être omis, il vaut alors application/x-www-form-urlencoded).

- `<fieldset>...</fieldset>`
- `<legend>...</legend>`
- `. . .`
- `<input type="text" name="nom" value="défaut" maxlength="42" />`
- `<input type="password" name="nom" value="défaut" maxlength="42" />`
- `<input type="checkbox" name="nom[]" value="valeur" checked="checked" />`
- `<input type="radio" name="nom" value="valeur" checked="checked" />`
- `<input type="file" name="nom" />`
- `<input type="hidden" name="nom" value="valeur" />`
- `<input type="reset" value="étiquette" />`
- `<input type="submit" value="étiquette" />`
- `<textarea name="nom" cols ="80" rows="5"> . . . </textarea>`
- `<select name="nom">`
  - `<option value="valeur1">...</option>`
  - `<option value="valeur2" selected="selected">...</option>`
- `</select>`

3.5 Conservation des données

- Les informations fournies par l'utilisateur peuvent être stockés dans un fichier (ex : fichier `csv`).
- La plupart du temps, on utilisera des bases de données :

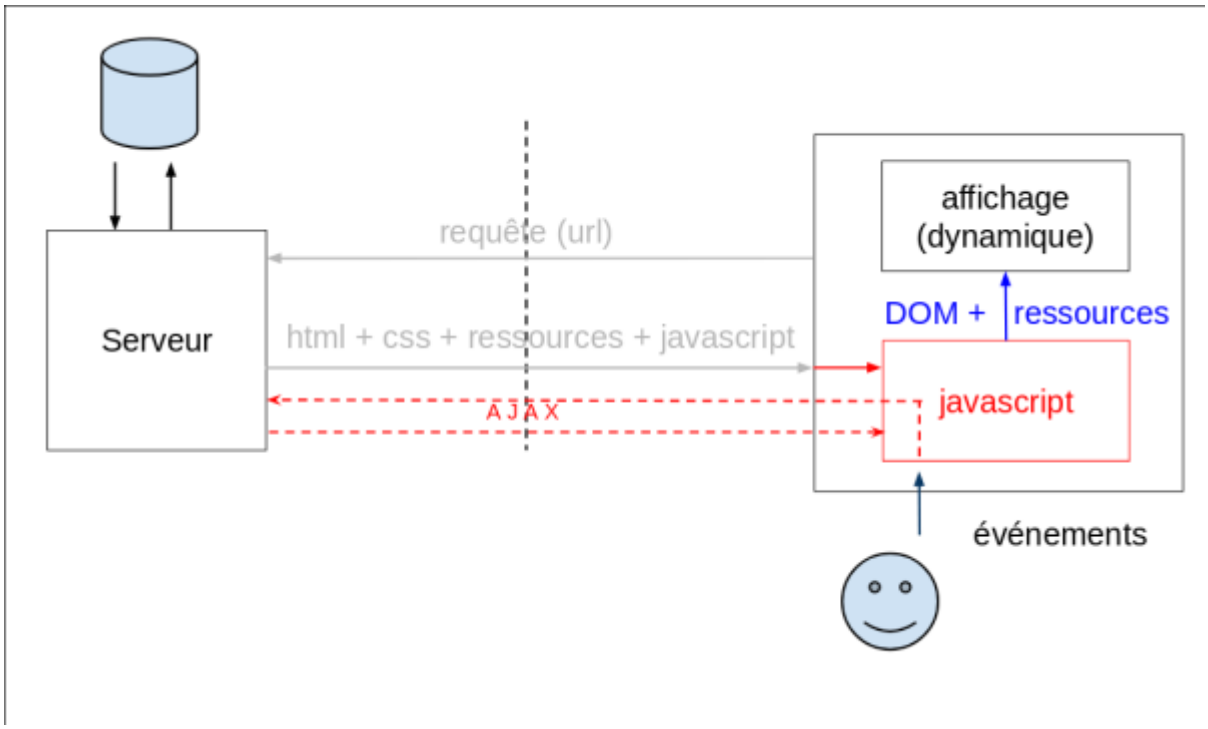
    - plus robuste
    - plus sécurisé
    - plus rapide
- Le web dynamique repose sur :
    - la mise à jour en continu d'informations
    - gérées par une base de données
    - tout se passe côté serveur
- Mise en œuvre :
    - PHP
    - `Python` + django
    - `Java` + Hibernate + Spring MVC

- Le patron MVC est devenu la norme pour le développement d'applications Web.
- On pourra prendre exemple sur l'environnement de développement Django pour la mise en œuvre : 1871426-le-fonctionnement-de-django

3.6 Interfaces Mixtes

- Seule une partie des contenus est mise à jour lors d'une action de l'utilisateur
- Évite de recharger les pages à chaque action de l'utilisateur

JQuery + AJAX



- apprendre-et-comprendre-jquery-3-3
- ajax

Exemples :

```
$.ajax({
    type: "POST",
    url: "test.html",
    success:
    function(retour){
        alert("Données retournées : " + retour );
    }
});
```

- Requête sur un lien:

```
$("a.test").click(function() {
    $.ajax({
        type: "POST",
        url: $(this).attr("href"),
        success: function(retour){
            $("#recipient").empty().append(retour);
        }
    });
    return false;
});
```

- Requête sur un formulaire:

```
$("form.test").submit(function() {
s = $(this).serialize();
$.ajax({
        type: "POST",
         data: s,
         url: $(this).attr("action"),
         success: function(retour){
            $("#recipient").empty().append(retour);
         }
     });
return false;
 });
```