

BDW : BASES DE DONNÉES & PROGRAMMATION WEB

PARTIE PROGRAMMATION WEB



Python
Avancé

MVC

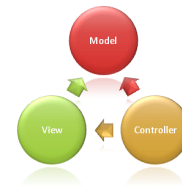
Nicolas Lumineau

nicolas.lumineau@univ-lyon1.fr

Remerciements à Fabien Duchateau

PYTHON AVANCÉ & MVC

- Module et Package
- Connexion à la BD
 - Module psycopg
- Templates et Moteur de templates
- Design pattern MVC
- Les Variables HTTP
- Organisation du code



LES MODULES ET PACKAGES

○ Définitions

- Un **module** est un fichier Python contenant des définitions de variables et de fonctions
- Une collection de modules est appelée **package**

○ Intérêt:

- Permet de structurer le code
- Facilite la réutilisation du code
- Évite les conflits de nommage
 - Possibilité de deux fonctions du même nom dans deux modules différents

LES MODULES ET LES PACKAGES

- Importation d'un module ou d'un package

- Pour l'importation complète du module

```
# importation de tout le module 'nom_module'  
import nom_module
```

- Pour l'importation spécifique de fonctions ou variables d'un module

```
# importation des fonctions 'fonct1' et 'fonct2' du module 'nom_module'  
from nom_module import fonct1, fonct2
```

- Par convention, les imports sont regroupés en début de fichier

OÙ TROUVER DES MODULES ET DES PACKAGES ?

- Python Standard Library (PSL):
 - Modules de bases (ex: fonction `print()`)
 - Pas d'import nécessaire
 - Modules optionnels
 - Modules fournis avec la distribution Python mais qu'il est nécessaire d'importer (cf. `import`)
- Python Package Index (PyPi)
 - Modules divers et variés selon les projets
 - Il est recommandé de vérifier l'état (date de dernière mise à jour et activité de la communauté) du module avant de l'utiliser.
- Développer vos propres modules

IMPORT DE MODULES ISSUS DE PSL

- Exemple : module random
 - pour générer un nombre aléatoirement

```
import random
# affectation d'un entier aléatoire entre 0 et 10 dans la variable alea
alea = random.randint(0,10)
print(alea)                # affiche : 3
# affectation d'un entier aléatoire entre 0 et 10 dans la variable alea
alea = random.randint(0,10)
print(alea)                # affiche : 1
# affectation d'un entier aléatoire entre 0 et 10 dans la variable alea
alea = random.randint(0,10)
print(alea)                # affiche : 7
```

IMPORT DE MODULES ISSUS DE PSL

- Exemple : module random
 - pour sélectionner une valeur aléatoire dans une séquence

```
import random

lst = [2,12,14,7,21,23,48]

# sélection d'une valeur dans la liste 'lst'
alea = random.choice(lst)
print(alea)                # affiche : 48
# sélection d'une valeur dans la liste 'lst'
alea = random.choice(lst)
print(alea)                # affiche : 21
# sélection d'une valeur dans la liste 'lst'
alea = random.choice(lst)
print(alea)                # affiche : 7
```

IMPORT DE MODULES ISSUS DE PSL

- Exemple : module `datetime`
 - pour récupérer des informations sur la date et l'heure courante

```
import datetime

dc = datetime.date.today()
print( dc )      # affiche : 2025-02-21
ydc = datetime.date.today().year
print( ydc )     # affiche : 2025
mdc = datetime.date.today().month
print( mdc )     # affiche : 2
ddc = datetime.date.today().day
print( ddc )     # affiche : 21
```

```
from datetime import datetime

dc = datetime.now()
print( dc )      # affiche : 2025-02-21 15:55:36.736681
```


IMPORT DE MODULES ISSUS DE PSL

- Exemple : module `os` et module `os.path`
 - pour effectuer des opérations en lien avec le système d'exploitation dont la gestion des fichiers (création, droits d'accès...) et des processus.

```
import os
from os import path

mon_rep = '/tmp'

# lister les fichiers
if path.isdir( mon_rep ):
    lst_rep = os.listdir( mon_rep )
    print( lst_rep )
else:
    print(mon_rep + " n'est pas un répertoire" )
```

IMPORT DE MES PROPRES MODULES

○ Créer son propre module

- Exemple : fichier `ma_boite_à_outils.py`

```
# Module : ma_boite_à_outils

# variables globales
motif = '*'

'''
set_motif_pour_pprint : affecte la valeur du paramètre
à la variable globale motif
'''

def set_motif_pour_pprint( s ):
    motif = s

'''
mypprint prend une string en paramètre et affiche
la valeur de la string encadré par le symbole
de la variable globale motif
'''

def mypprint( s ):
    l = len( s ) + 4    # 4 caractères supplémentaires à la longueur
                        # du mot pour un motif en début et fin de ligne
                        # et un espace avant et après le mot

    stars = ''
    for i in range(l):
        stars += motif
    print( stars )
    print( motif + ' ' + s + ' ' + motif )
    print( stars )
```

IMPORT DE MES PROPRES MODULES

- Utiliser un de ses modules

```
# importation du module
import ma_boite_à_outils

# appel de la fonction en précisant le nom du module
ma_boite_à_outils.mypprint( 'BDW' )

# Affiche : *****
#           * BDW *
#           *****
```

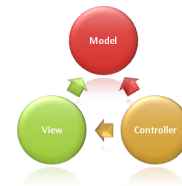
```
# importation du module
from ma_boite_à_outils import set_motif_pour_pprint, mypprint

# appel des fonctions direct
set_motif_pour_pprint( '-' )
mypprint( 'BDW' )

# Affiche : -----
#           - BDW -
#           -----
```

PYTHON AVANCÉ & MVC

- Module et Package
- Connexion à la BD
 - Module psycopg
- Templates et Moteur de templates
- Design pattern MVC
- Les Variables HTTP
- Organisation du code



CONNEXION À LA BASE DE DONNÉES

- On suppose que la table Personne a été créée sur la Base de Données et que des instances ont été insérées

personne

idp	nom	prénom
1	TARTINE	Kimberley
2	PELLE	Sarah
3	ONETTE	Marie
4	ONETTE	Camille
5	VERRE	Justin

PYTHON ET BD

- Pour interagir avec les bases de données, il existe différentes librairies Python adaptées à chaque SGBD ciblé
 - Pour PostgreSQL
 - psycopg
 - pygresql
 - podbc
 - ...

Ces différents modules Python offre un accès standardisé aux SGBD

Pour BDW,
nous allons utiliser **psycopg**
qui est la librairie la plus populaire



BDW

INTERACTIONS AVEC LE SGBD

- L'interaction entre un programme (quel qu'il soit, et pas seulement pour Python) se décompose en 4 étapes :
 1. Connexion au SGBD
 2. Exécution d'une requête
 3. Récupération des résultats
 4. Fermeture de la connexion

À noter qu'on peut itérer les étapes 2 à 3 autant de fois que nécessaire avant de fermer la connexion à l'étape 4

GESTION DE LA CONNEXION

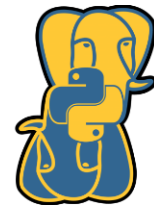
- Pour établir une connexion, il est nécessaire de spécifier les paramètres de connexion :
 - adresse du serveur de données qui héberge le SGBD
 - identifiant de l'utilisateur
 - mot de passe de l'utilisateur
 - nom de la Base de Données
 - nom du schéma



Ces informations sont sensibles et doivent être stockés dans un fichier à part avec des droits restreints

```
# définition des constantes de connexion
SERVER = 'bd-pedago.univ-lyon1.fr'
USER = 'p1234567'
PASSWORD = 'secretENclair'
DATABASE = 'p1234567'
SCHEMA = 'projet'
```


ÉTAPE 1 : CONNEXION AU SGBD



- Utilisation de `psycopg.connect()` pour établir la connexion

```
1  import psycopg
2
3  '''
4  get_connexion : fonction qui prend en argument les paramètres de connexion:
5  - le nom du serveur hôte de la BD
6  - l'identifiant de l'utilisateur
7  - le mot de passe de l'utilisateur
8  - le nom de la BD
9  - le nom du schéma
10  La fonction retourne un objet de connexion qui représente le lien de connexion vers la BD,
11  et lève une exception si la connexion n'a pas pu être établie
12  '''
13  def get_connexion(server_host, login, passwd, db, schema='public'):
14      try:
15          # création de la connexion à la BD
16          connexion = psycopg.connect(host=server_host,
17                                     user=login,
18                                     password=passwd,
19                                     dbname=db,
20                                     autocommit=True)
21
22          # Sélection du schéma courant
23          cursor = psycopg.Cursor( connexion )
24          cursor.execute( f"SET search_path TO {schema}" )
25      except Exception as e:
26          print( e )
27          return None
28      return connexion
```

ÉTAPE 2 : EXÉCUTION D'UNE REQUÊTE

○ Utilisation d'un curseur


- Objet qui permet d'exécuter une ou plusieurs requêtes et de récupérer le résultat sous la forme d'une liste de n-uplets (par défaut)

```
# Rappel de la création de la connexion
connexion = get_connexion(SERVER, USER, PASSWORD, DATABASE, SCHEMA)

# création du curseur à partir de la connexion
curseur = connexion.cursor()
# définition de la requête
requete = 'SELECT * FROM personne'
# exécution de la requête
curseur.execute( requete )
```

EXEMPLE D'UNE EXÉCUTION D'UNE REQUÊTE

```
1  '''
2  get_personnes : fonction qui prend en paramètre une connexion vers un SGBD
3  et qui retourne une liste de tuples représentant les n-uplets
4  stockés dans la table 'personne'
5  '''
6  def get_personnes( c ):
7      try:
8          curseur = c.cursor()
9          requête = "SELECT * FROM Personnes"
10         curseur.execute( requête )
11         return curseur
12     except psycopg.Error as e:
13         print(f"Error: {e}")
14         return None
15
16 # appel de la fonction
17 get_personnes(connexion)
18 # retourne : [ (1, 'TARTINE', 'Kimberley'),
19 #              (2, 'PELLE', 'Sarah'),
20 #              (3, 'ONETTE', 'Marie'),
21 #              (4, 'ONETTE', 'Camille'),
22 #              (5, 'VERRE', 'Justin')
23 #              ]
```



Type de retour :
liste de tuples

EXEMPLE D'UNE EXÉCUTION D'UNE REQUÊTE

```
1  '''
2  get_personnes : fonction qui prend en paramètre une connexion vers un SGBD
3  et qui retourne une liste de dictionnaires représentant les n-uplets
4  stockés dans la table 'personne'
5  '''
6  def get_personnes( c ):
7      try:
8          curseur = c.cursor()
9          curseur.row_factory = dict_row
10         requête = "SELECT * FROM Personnes"
11         curseur.execute( requête )
12         return curseur
13     except psycopg.Error as e:
14         print(f"Error: {e}")
15         return None
16
17 # appel de la fonction
18 get_personnes(connexion)
19 # retourne : [ ('idp':1, 'nom':'TARTINE', 'prénom':'Kimberley'),
20 #              ('idp':2, 'nom':'PELLE', 'prénom':'Sarah'),
21 #              ('idp':3, 'nom':'ONETTE', 'prénom':'Marie'),
22 #              ('idp':4, 'nom':'ONETTE', 'prénom':'Camille'),
23 #              ('idp':5, 'nom':'VERRE', 'prénom':'Justin')
24 #              ]
```

Type de retour :
liste de
dictionnaires



À PROPOS DE LA CONSTRUCTION DES REQUÊTES

○ Requête avec variables dans le WHERE

- Utilisation d'une liste de variables comme second paramètre de la fonction `execute()` du curseur
 - Chaque `%s` dans la requête SQL est remplacé par une valeur de la liste des variables (premier `%s` dans la `string` associée à la première valeur de la liste; ...)

```
1  '''
2  get_personnes_par_nom : fonction qui prend en paramètre une connexion vers un SGBD et
3  une string représentant un nom de personne et qui retourne une liste de tuples représentant
4  les n-uplets stockés dans la table 'personne' dont le nom est passé en paramètre
5  '''
6  def get_personnes_par_nom( c , nom ) :
7      try:
8          curseur = c.cursor()
9          requête = "SELECT * FROM Personnes WHERE nom = %s"
10         curseur.execute( requête , [nom])
11         return curseur
12     except psycopg.Error as e:
13         print(f"Error: {e}")
14         return None
15
16 # appel de la fonction
17 get_personnes_par_nom(connexion, 'ONETTE')
18 # retourne : [ (3, 'ONETTE', 'Marie'),
19 #              (4, 'ONETTE', 'Camille')
20 #              ]
```

À PROPOS DE LA CONSTRUCTION DES REQUÊTES

○ Requête avec variables dans le WHERE

- Autre exemple

```
1  '''
2  get_personnes_par_nom_et_prénoms : fonction qui prend en paramètre une connexion vers un SGBD,
3  une string représentant un nom de personne et une liste de 2 prénoms et qui retourne une liste de
4  tuples représentant les n-uplets stockés dans la table 'personne' dont le nom est passé en paramètre
5  et le prénoms est soit le premier élément de la liste soit le second
6  '''
7  def get_personnes_par_nom_et_prénoms( c , nom, prénoms ) :
8      try:
9          param = [nom].extend(prénoms)
10         curseur = c.cursor()
11         requête = "SELECT * FROM Personnes WHERE nom = %s AND (prénom = %s OR prénom = %s)"
12         curseur.execute( requête , param)
13         return curseur
14     except psycopg.Error as e:
15         print(f"Error: {e}")
16         return None
17
18 # appel de la fonction
19 get_personnes_par_nom_et_prénoms(connexion, 'ONETTE' , ['Marie', 'Camille'])
20 # retourne : [ (3, 'ONETTE', 'Marie'),
21 #              (4, 'ONETTE', 'Camille')
22 #              ]
```

À PROPOS DE LA CONSTRUCTION DES REQUÊTES

○ Requête avec variables dans le FROM

- Utilisation de la fonction `sql.SQL()` pour formater le nom de table dans la requête

```
1  from psycopg import sql
2
3  '''
4  get_instances : fonction qui prend en paramètre une connexion vers un SGBD,
5  une string représentant un nom de table et qui retourne une liste de
6  tuples représentant les n-uplets stockés dans la table passée en paramètre
7  '''
8  def get_instances( c , nom_table ):
9      try:
10         curseur = c.cursor()
11         requête = sql.SQL("SELECT * FROM {nomtable}").format(nomtable=sql.Identifier(nom_table))
12         curseur.execute( requête )
13         return curseur
14     except psycopg.Error as e:
15         print(f"Error: {e}")
16         return None
17
18 # appel de la fonction
19 get_instances(connexion, 'personnes')
20 # retourne : [ (1, 'TARTINE', 'Kimberley'),
21 #              (2, 'PELLE', 'Sarah'),
22 #              (3, 'ONETTE', 'Marie'),
23 #              (4, 'ONETTE', 'Camille'),
24 #              (5, 'VERRE', 'Justin')
25 #              ]
```

À PROPOS DE LA CONSTRUCTION DES REQUÊTES

- Pourquoi il NE faut PAS construire les requêtes SQL de cette manière:

```
1 '''
2 get_connecter : fonction qui prend en paramètre une connexion vers un SGBD,
3 deux string représentant respectivement un identifiant et un mot de passe
4 et qui retourne les informations de la personne identifiée
5 '''
6 def get_connecter( c , identifiant, passwd ) :
7     try:
8         curseur = c.cursor()
9         requête = "SELECT * FROM utilisateurs WHERE login = '" + identifiant + "' AND pwd = '" + passwd + "'"
10        curseur.execute( requête )
11        return curseur
12    except psycopg2.Error as e:
13        print(f"Error: {e}")
14        return None
```

Indice :

Que retourne l'appel suivant :

```
get_connecter( connexion, "admin' OR '1' = '1" , "pwdBidon")
```

Toutes les infos sur l'utilisateur admin !

ÉTAPE 3 : RÉCUPÉRATION DES RÉSULTATS

- Primitives du curseur adaptées au type de requêtes
 - Pour les requêtes de type SELECT
 - Récupération des instances qui filtrent la requête , *via* :
 - `curseur.fetchone()` : retourne un tuple contenant le prochain n-uplets dans le résultat de la requête et `None` quand tous les n-uplets ont été retourné
 - `curseur.fetchmany(nb)` : retourne une liste contenant au plus les nb prochains n-uplets dans le résultat de la requête et `None` quand tous les n-uplets ont été retourné
 - `curseur.fetchall()` : retourne la liste de tous les n-uplets contenus dans le résultat de la requête
 - Pour les requêtes de type UPDATE, INSERT, DELETE
 - Indication du nombre d'instances impactées par la requête, *via* `curseur.rowcount` :
 - Nombre de n-uplets modifiés par un UPDATE
 - Nombre de n-uplets ajoutés pour un INSERT
 - Nombre de n-uplets supprimés par un DELETE

EXEMPLE DE RÉCUPÉRATION DE RÉSULTAT

○ Récupération des instances

- Avec un `fetchone()`:

```
1  ...
2  curseur = c.cursor()
3  curseur.row_factory = dict_row
4  requête = "SELECT count(*) AS nb FROM personnes"
5  curseur.execute( requête )
6  nb_personnes = curseur.fetchone()['nb']
7  print(nb_personnes)
8  # Affiche : 5
```

- Avec un `fetchall()`:

```
1  ...
2  curseur = c.cursor()
3  requête = "SELECT * FROM personnes"
4  curseur.execute( requête )
5  les_personnes = curseur.fetchall()
6  for personne in les_personnes:
7      print(personne)
8  # Affiche : #   (1, 'TARTINE', 'Kimberley')
9  #               (2, 'PELLE', 'Sarah')
10 #               (3, 'ONETTE', 'Marie')
11 #               (4, 'ONETTE', 'Camille')
12 #               (5, 'VERRE', 'Justin')
```

EXEMPLE DE RÉCUPÉRATION DE RÉSULTAT

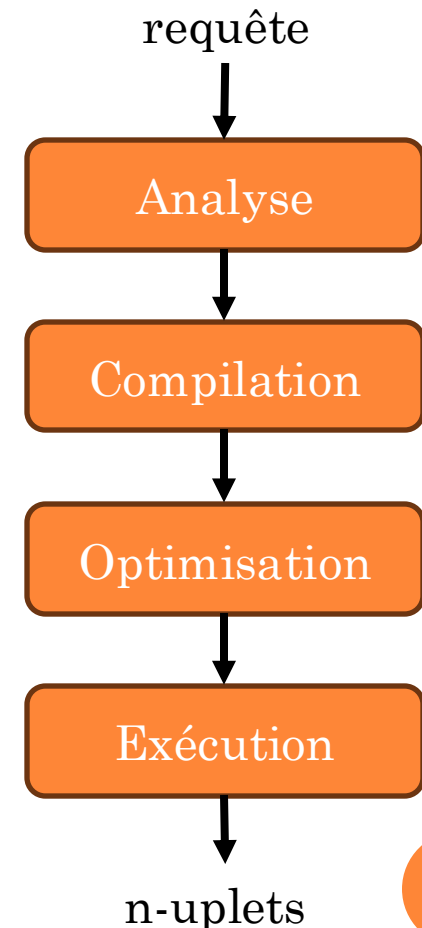
- Récupération du nombre de n-uplets affectés

```
1  ...
2  curseur = c.cursor()
3  requête = "UPDATE personnes SET nom = 'SURE' WHERE idp = 2"
4  curseur.execute( requête )
5  nb_maj = curseur.rowcount
6  print("Nombre de n-uplets modifiés : " + nb_maj)
7  # Affiche : Nombre de n-uplets modifiés : 1
```

LES REQUÊTES PRÉPARÉES

- Ces requêtes vont être analysées et compilées avec des variables
- Les phases d'optimisation et d'exécution se feront au moment de l'affectation des variables.
- Il est possible de rappeler la requête préparée plusieurs fois avec des affectations de variables différentes
- Avantage principal
 - Temps de traitement réduit dès la seconde exécution (pas de phase d'analyse et de compilation)

Traitement
classique
d'une requête



BDW

EXEMPLE DE REQUÊTES PRÉPARÉES

```
1  '''
2  get_personnes_par_liste_noms : fonction qui prend en paramètre une connexion vers un SGBD et
3  une liste de string représentant des nom de personne et qui retourne une liste de dictionnaires
4  stockant les noms et les liste de prénoms des personnes dont le nom est dans la liste en entrée.
5  Le traitement est fait via une requête préparée
6  '''
7  def get_personnes_par_liste_noms( c , lst_nom ) :
8      try:
9          res = list()
10         curseur = c.cursor()
11         requête = "SELECT DISTINCT prénom FROM Personnes WHERE nom = %s"
12         for nom in lst_nom: # on parcourt les noms oassé en entrée
13             curseur.execute( requête , [nom] , prepare=True)
14             lst_prénoms = curseur.fetchall() # on récupère les prénoms associés au nom
15             p = { 'nom' : nom, 'prénoms' : lst_prénoms } # on construit le dictionnaire pour le nom
16             res.append( p ) # on l'ajoute au résultat
17         return res
18     except psycopg.Error as e:
19         print(f"Error: {e}")
20         return None
21
22 # appel de la fonction
23 get_personnes_par_liste_noms(connexion, ['TARTINE','ONETTE'])
24 # retourne : [ {'nom':'TARTINE', 'prénoms':['Kimberley']},
25 #              ('nom':'ONETTE', 'prénoms': ['Marie','Camille'])
26 #              ]
```

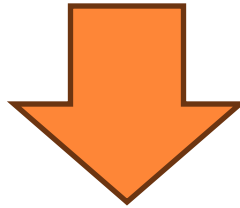
ÉTAPE 4 : FERMETURE DE LA CONNEXION

- Une fois l'ensemble des requêtes exécutées, il est recommandé de fermer la connexion, *via* la primitive `close()`.

```
1  '''
2  deconnexion : fonction qui prend en entrée une connexion et qui la ferme avant de retourner
3  un booléen à True si tout s'es bien passé.
4  '''
5  def deconnexion( c ):
6      c.close()
7      return True
8
9  # appel de la fonction
10 deconnexion( connexion )
```

REMARQUE

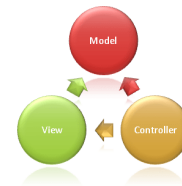
- Nous savons faire à présent les traitements grâce au langage de programmation Python
- Il nous reste à voir comment insérer dynamiquement des données dans un contenu HTML



Utilisation de « *templates* » et d'un « moteur de template »

PYTHON AVANCÉ & MVC

- Module et Package
- Connexion à la BD
 - Module psycopg
- Templates et Moteur de templates
- Design pattern MVC
- Les Variables HTTP
- Organisation du code

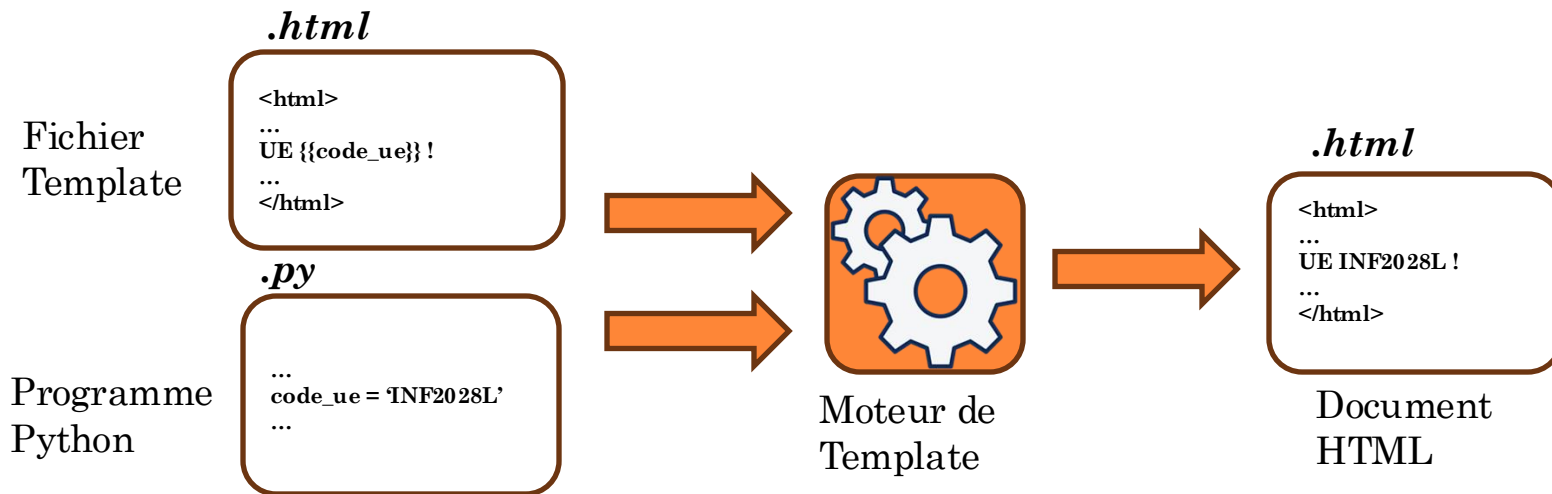


LES *TEMPLATES*

- Un *template* (ou gabarit) est un modèle de langage qui spécifie la façon dont les données issues de l'application sont insérées dans le document/page produit.
- Un template inclut des variables et des instructions particulières (ex: conditionnelle, boucle, affichage de variables).
- Un template est interprété par le « moteur de template »
- Intérêt
 - Facilite la factorisation du code et sa réutilisabilité
 - Permet la séparation des différentes parties du code

MOTEUR DE TEMPLATE

- Le moteur de template permet:
 - la substitution des variables du programme dans le template
 - des traitements de manipulation des variables
- Le moteur de template produit le document final (ici HTML)



MOTEUR DE TEMPLATE POUR PYTHON

- Il existe de nombreux moteur de template, e,n particulier pour Python :

<https://wiki.python.org/moin/Templating>

- Pour BDW, nous avons choisi :



- C'est un moteur de *template* couplé à Python qui offre un fonctionnement simplifié avec le serveur web Python
 - Pas d'environnement spécifique à gérer
 - Passage automatique des variables Python au *template*
 - Permet l'inclusion de templates dans des templates
 - Permet l'héritage entre templates

SYNTAXE DES TEMPLATES



- Un template Jinja contient

- du code HTML
- des commentaires

```
{# un commentaire dans le template #}
```

- des variables (à substituer)

```
{{ <nom_variable> }}
```

- des structures de contrôles

```
{% if <condition> %}  
  <!-- code HTML du bloc conséquence -->  
{% else %}  
  <!-- code HTML du bloc alternative -->  
{% endif %}
```

- des boucles

```
{% for <variable> in <liste> %}  
  <!-- code HTML de la boucle for incluant {{variable}} -->  
{% endfor %}
```

EXEMPLE DE SUBSTITUTION DE VARIABLES

- Extrait fichier Python

```
ue = "Bases de Données et programmation Web"  
info_ue = { 'sigle':'BDW', 'code_apogee':"INF2028L" }
```

- Extrait template Jinja

```
<h1>{{ue}}</h1>  
<p>Sigle : {{info_ue['sigle']}}</p>  
<p>Code APOGEE : {{info_ue.code_apogee}}</p>
```

Notation avec crochet

Notation pointée

- Rendu

Bases de Données et programmation Web

Sigle : BDW

Code APOGEE : INF2028L

BDW

EXEMPLE DE CONDITIONNELLE

○ Extrait fichier Python

```
ue = "Bases de Données et programmation Web"
info_ue = { 'sigle':'BDW', 'code_apogee':"INF2028L"}
étudiant = {'etu':"Yves HALIDE", 'note':16.5}
a_validé = (étudiant['note'] >= 10)
```

○ Extrait template Jinja

```
<h1>{{ue}}</h1>
<p>Sigle : {{info_ue['sigle']}}</p>
<p>Code APOGEE : {{info_ue.code_apogee}}</p>

{% if a_validé %}
    <p>{{étudiant.etu}} : <span class="adm">Admis</span></p>
{% else %}
    <p>{{étudiant.etu}} : <span class="aj">Ajourné</span></p>
{% endif %}
```

○ Rendu

Bases de Données et programmation Web

Sigle : BDW

Code APOGEE : INF2028L

Yves HALIDE : Admis

BDW

EXEMPLE DE BOUCLE

○ Extrait fichier Python

```
étudiants = [{'etu': "Yves HALIDE", 'note': 12},  
             {'etu': "Medhi DONC-SAPASPA", 'note': 9},  
             {'etu': "Ella TOURATET", 'note': 5},  
             ]
```

○ Extrait template Jinja

```
<ul>  
{% for e in étudiants %}  
    {% if e.note >= 10 %}  
    <li>{{e.etu}} : <span class="adm">Admis</span></li>  
    {% else %}  
    <li>{{e.etu}} : <span class="aj">Ajourné</span></li>  
    {% endif %}  
{% endfor %}  
</ul>
```

○ Rendu

- Yves HALIDE : Admis
- Medhi DONC-SAPASPA : Ajourné
- Ella TOURATET : Ajourné

INCLUSION DE TEMPLATES

- Il est possible de décomposer un template en plusieurs templates: `include`
- Intérêt
 - Permet de factoriser du code pour plusieurs templates

index.html

```
{% include 'header.html' %}  
<main>  
    <!-- template principal -->  
    <!-- incluant les templates -->  
    <!-- du header et du footer-->  
</main>  
{% include 'footer.html' %}
```

header.html

```
<header>  
    <!-- template du header -->  
    <h1>Bases de Données et programmation Web</h1>  
</header>
```

footer.html

```
<footer>  
    <!-- template du footer -->  
</footer>
```


HÉRITAGE ENTRE TEMPLATES

- Dans Jinja, un template peut hériter d'un autre template: `extends`
 - Le **template parent** définit des blocs nommées (éventuellement sans contenu)
 - Le **templates enfants** étendent le template parent et permet de redéfinir un contenu pour les blocs nommés
- L'héritage est utilisé pour avoir un agencement et un design cohérents sur toutes les pages du site, sans redondance de code :
 - Le template parent définit l'agencement général
 - Ce qui est commun
 - Le template enfant définit le contenu spécifique d'une page

HÉRITAGE

héritage

héritage

base.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <link type="text/css" href="style.css">
    <title>Mon site Web</title>
  </head>
  <body>
    {% include 'header.html' %}
    <main>
      {% block main %}{% endblock %}
    </main>
    {% include 'footer.html' %}
  </body>
</html>
```

afficher.html

```
{% extends "base.html" %}
{% block main %}
<h2>Affichage</h2>
  <table>
    ...
  </table>
{% endblock %}
```

ajouter.html

```
{% extends "base.html" %}
{% block main %}
<h2>Ajout</h2>
<form>
  ...
</form>
{% endblock %}
```

base.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <link type="text/css" href="style.css">
    <title>Mon site Web</title>
  </head>
  <body>
    {% include 'header.html' %}
    <main>
      {% block main %}{% endblock %}
    </main>
    {% include 'footer.html' %}
  </body>
</html>
```

afficher.html

```
{% extends "base.html" %}
{% block main %}
  <h2>Affichage</h2>
  <table>
    ...
  </table>
{% endblock %}
```

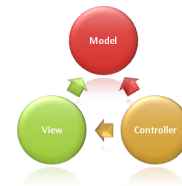
Redéfinition du contenu

ajouter.html





```
{% extends "base.html" %}
{% block main %}
  <h2>Ajout</h2>
  <form>
    ...
  </form>
{% endblock %}
```

PYTHON AVANCÉ & MVC

- Module et Package
- Connexion à la BD
 - Module psycopg
- Templates et Moteur de templates
- Design pattern MVC
- Les Variables HTTP
- Organisation du code








POUR RAPPEL

- Pour la structure et le contenu des pages :
→ **HTML** 
- Pour la mise en forme des pages :
→ **CSS** 
- Pour les traitements de génération de contenu (côté serveur)
→ **Python**  **psycopg** → interrogation de la BD postgresql
Jinja → moteur de templates pour produire des pages HTML
- Pour interroger les données
→ **SQL** 

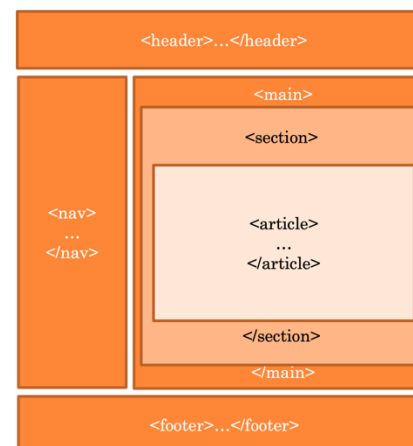
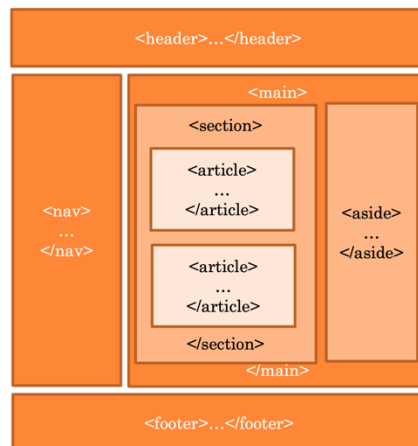
UNE APPLICATION WEB

- Des fonctionnalités & des interfaces Web

- Aller sur l'accueil  accueil.html
- Afficher les données  afficher.html
- Ajouter une donnée  ajouter.html
- Supprimer une donnée  supprimer.html
- xyz  xyz.html

UNE APPLICATION WEB

- Généralement, deux pages d'un même site, on :
 - Le même entête
 - Le même menu
 - Le même pied de page
 - La même charte graphique
 - La même mise en forme
 - La même mise en page



EXEMPLE DE PROGRAMMATION D'UNE PAGE

afficher.py

```
1  import psycopg
2  from jinja import Environment, PackageLoader
3
4  connexion = psycopg.connect(...)
5  try:
6      cursor = connexion.cursor()
7      cursor.execute("SELECT * FROM personne")
8      instances = cursor.fetchall()
9  except psycopg.Error as e:
10     print(f"Error : {e}")
11     return None
12
13  nb_instances = len(instances)
14  env = Environment(
15      loader = PackageLoader("mon_site"),
16  )
17  template = env.get_template("afficher.html")
18  print(template.render(titre='Liste des personnes',
19                      instances=instances,
20                      nb=nb_instances))
```

afficher.html

```
<h1>{{ titre }}</h1>
<p>Il y a {{ nb }} personnes.</p>
<ul>
    {% for i in instances %}
        <li>{{ i }}</li>
    {% endfor %}
</ul>
```


QUESTION

- Comment structurer le code ?
- Comment organiser les fichiers et les répertoires ?
- Comment gérer les différents types de pages ?

**Il est donc préférable
de passer par un *design pattern* !**

INTRODUCTION À MVC

- Architecture Modèle Vue Contrôleur
 - Patron de conception / Design pattern (origine en 1978)
 - C'est une façon d'organiser le code ➔ bonne pratique
- Architecture reposant sur 3 types de composant:
 - Le **modèle** : composant qui gère l'accès aux données
 - La **vue** : composant qui gère l'affichage des données
 - Le **contrôleur** : composant qui orchestre la construction de la page en interagissant avec le modèle et la vue

INTRODUCTION À MVC

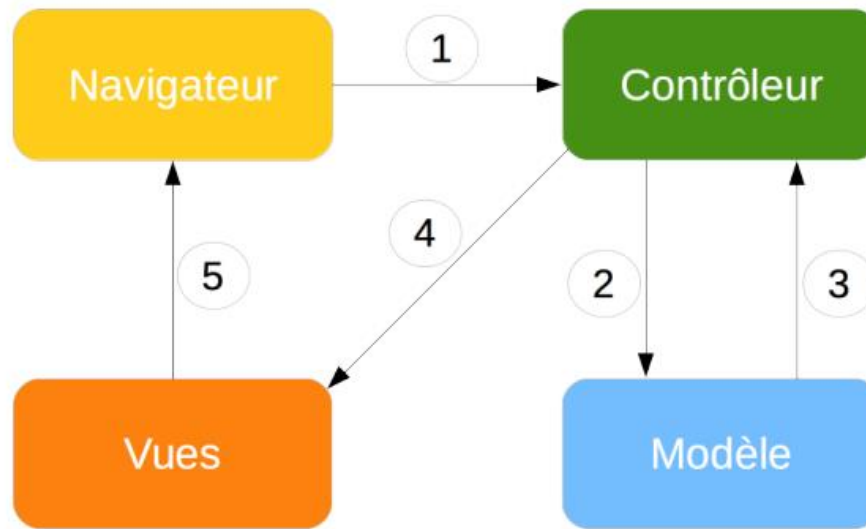
○ Avantages:

- **Indépendance du modèle par rapport à la Vue et au Contrôleur** (mais la réciproque est fausse)
- Approche très populaire et utilisée par de nombreux sites WEB et de *frameworks*
- Structuration du code facilitant sa maintenance
- Réutilisabilité du code simplifiée
- Développement en parallèle facilité
- Détection d'erreurs

○ Inconvénients:

- Peut sembler plus complexe à gérer
- Induit un léger « éparpillement » des fonctionnalités

LES INTERACTIONS ENTRE COMPOSANTS MVC



1. L'utilisateur envoie sa requête (demande d'URL + paramètres)
2. Le contrôleur vérifie les paramètres et appelle le modèle
3. Le modèle interroge la base de données et retourne un résultat au contrôleur
4. Le contrôleur sélectionne la vue à afficher et transmet le résultat à afficher
5. La vue retourne le code HTML au navigateur pour être visualisé par l'utilisateur

EXEMPLE MVC

controleur/afficher.py

```
from model.model_pers import get_personnes  
  
les_personnes = get_personnes(connexion)
```

model/model_pers.py

```
import psycopg  
  
def get_personnes(connexion):  
    try:  
        cursor = connexion.cursor()  
        cursor.execute("SELECT * FROM personne")  
        return cursor.fetchall()  
    except psycopg.Error as e:  
        print(f"Error : {e}")  
        return None
```

templates/afficher.html

```
{% extends "base.html" %}  
  
{% block main_content %}  
  
<h1>Liste des personnes</h1>  
  
<ul>  
    {% for p in les_personnes %}  
    <li>  
        {% for e, v in p.items() %}  
            <span>{{v}} &nbsp;</span>  
        {% endfor %}  
    </li>  
    {% endfor %}  
</ul>
```



Moteur de
Templates

Liste des personnes

- 10 CAMAIS Léon
- 20 CROQUE Odile
- 30 CATTE Suri

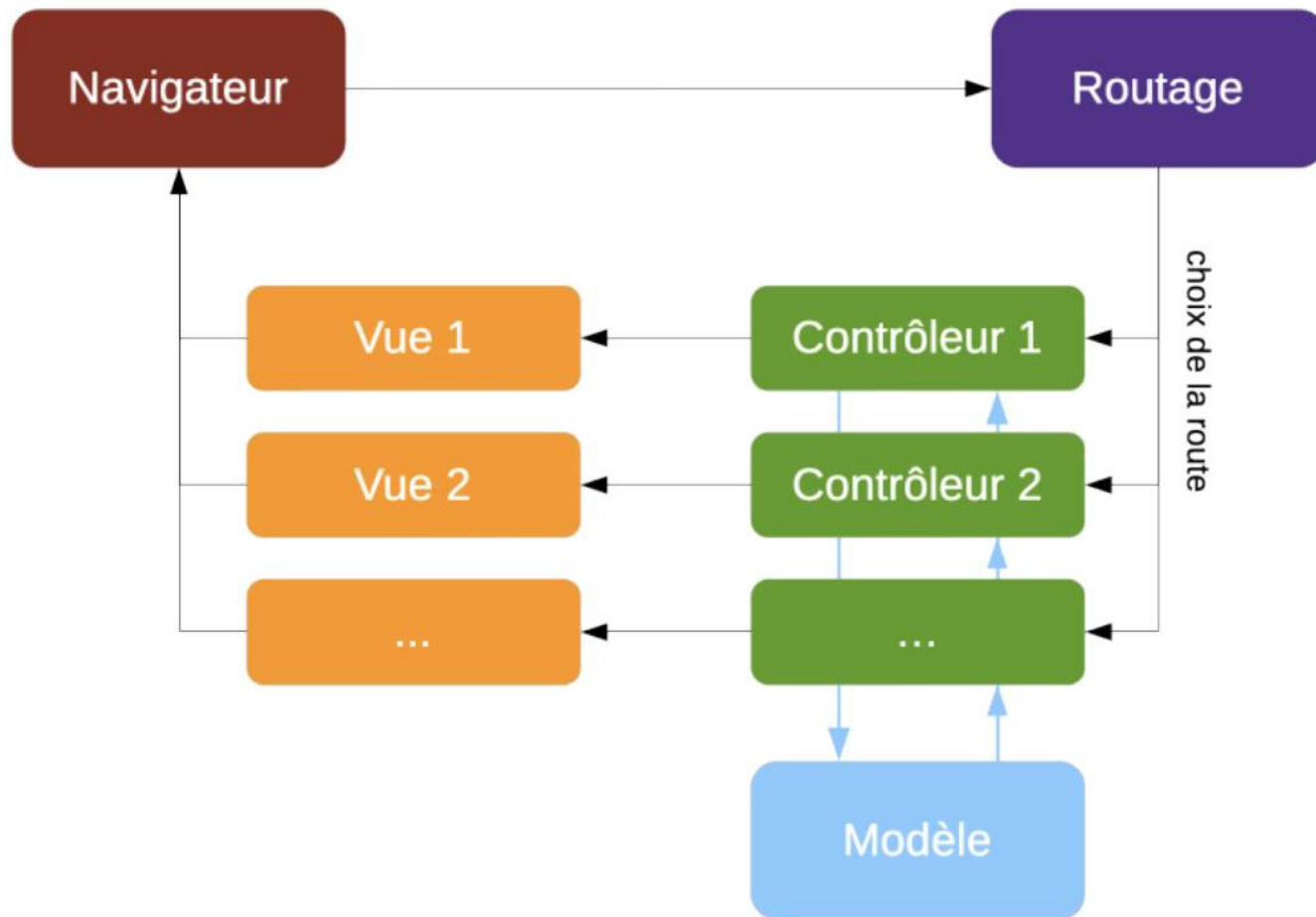
DÉFINITION DES FONCTIONNALITÉS

- Chaque fonctionnalité du site est associée :
 - Une page du site
 - Un **contrôleur**
 - Une **vue**
 - Des méthodes dans le **modèle**
- Comment lier la fonctionnalité au bon contrôleur et à la bonne vue ?



Principe de routage

MVC AVEC ROUTAGE



ROUTAGE

○ Principe d'une route

- Associer à une URL une fonctionnalité et à la fonctionnalité un couple contrôleur et vue

`https://www.mon_site.fr/afficher`

URL



`afficher`

Fonctionnalité



`afficher.py`

Contrôleur

`afficher.html`

Vue

○ Remarque

- Chaque framework ou serveur a son propre mécanisme de routage

POUR LES TPs

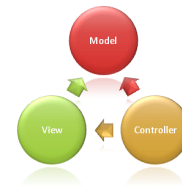
- Dans *bdw-server*,
 - Le routage par URL s'effectue avec un fichier de routes défini en TOML :
 - le serveur exécute le contrôleur et appelle la vue/template à partir des routes

routes.toml

```
[[routes]]
url = "afficher"
contrôleur = "contrôleurs/afficher.py" # chemin vers contrôleur
template = "afficher.html" # chemin vers vue/template
```

PYTHON AVANCÉ & MVC

- Module et Package
- Connexion à la BD
 - Module psycopg
- Templates et Moteur de templates
- Design pattern MVC
- Les Variables HTTP
- Organisation du code



GÉNÉRALITÉS

- Lorsqu'un serveur web reçoit une requête, il extrait des informations
 - Document demandé
 - URL, chemin dans l'arborescence de fichiers
 - Données soumises par un formulaire
 - Données liées à la réponse
 - Données stockées en session
 - ...



Dans la suite, la syntaxe des variables HTTP est orientée par rapport à la solution proposée en TP :
bdw-server

VARIABLES HTTP POUR BDW-SERVER

- GET : structure qui contient les données soumises par un formulaire avec méthode **get**
- POST : structure qui contient les données soumises par un formulaire avec méthode **post**
- REQUEST_VARS : structure qui contient les données uniquement accessibles pour la requête en cours
- SESSION : structure qui contient les données de session, accessibles tout au long de la navigation (conservées entre plusieurs pages)



Ces variables sont implémentées comme des dictionnaires et elles sont accessibles dans les contrôleurs et dans les templates !

FOCUS SUR LA VARIABLE REQUEST_VARS

- Le contrôleur peut avoir besoin de passer des données au template pour la requête courante :
 - Données récupérées après interrogation du modèle
 - Ex: liste des personnes
 - Message informatif ou d'erreur
- Principe :
 - La variable REQUEST_VARS est réinitialisée à vide lors de la réception d'une nouvelle requête HTTP et permet de stocker des informations pour générer la page HTML de réponse



Remarque importante

- Comme bdw-server simplifie les échanges, vous ne pouvez pas passer vous-même vos variables au template, d'où l'intérêt de cette variable.

FOCUS SUR LA VARIABLE SESSION

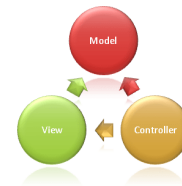
- Cette variable peut être utile de conserver des informations d'une page sur l'autre :
 - Login de l'internaute pour les sites avec authentification
 - Pages visitées par l'internaute (recommandations)
 - Panier (e-commerce), gestion d'une partie de jeu, etc.
 - ...
- Une session peut être vue comme un ensemble d'informations concernant un internaute (i.e., un navigateur sur une machine)

BILAN

- Un serveur web fournit des informations via des variables
 - GET et POST pour récupérer les valeurs d'un formulaire
 - REQUEST_VARS pour échanger des données temporaires entre contrôleur et template
 - SESSION pour stocker des informations persistantes même en changeant de page

PYTHON AVANCÉ & MVC

- Module et Package
- Connexion à la BD
 - Module psycopg
- Templates et Moteur de templates
- Design pattern MVC
- Les Variables HTTP
- Organisation du code



ORGANISATION D'UN SITE WEB

- Il peut y avoir différentes manières d'organiser un site, mais dans tous les cas, il est nécessaire de stocker :
 - Le code des pages HTML
 - Les routes
 - Les images
 - Les feuilles de style CSS
 - Les fichiers de configuration (accès BD...)

ORGANISATION DU SITE WEB : BDW-SERVER

Sur bdw-server, organisation des fichiers :

mon_site/	répertoire de votre site web
├─ init.py	obligatoire
├─ routes.toml	obligatoire
├─ controleurs/	
├─ model/	
├─ templates/	obligatoire
├─ static/	
│ └─ img/	
│ └─ css/	
└─ sql/	optionnel, pour vos scripts SQL

BILAN

- Focus sur la structuration d'un site Web
 - Organisation des fichiers dans l'arborescence du répertoire côté serveur
 - Deux approches d'organisation des fichiers selon les fonctionnalités
 - « un fichier par fonctionnalité »
 - « un fichier unique »
 - Introduction à l'architecture MVC
 - Bonne pratique
- Application de tout ça avec le TP4-5 et le projet !