BDW - Programmation web - structuration

Fabien Duchateau

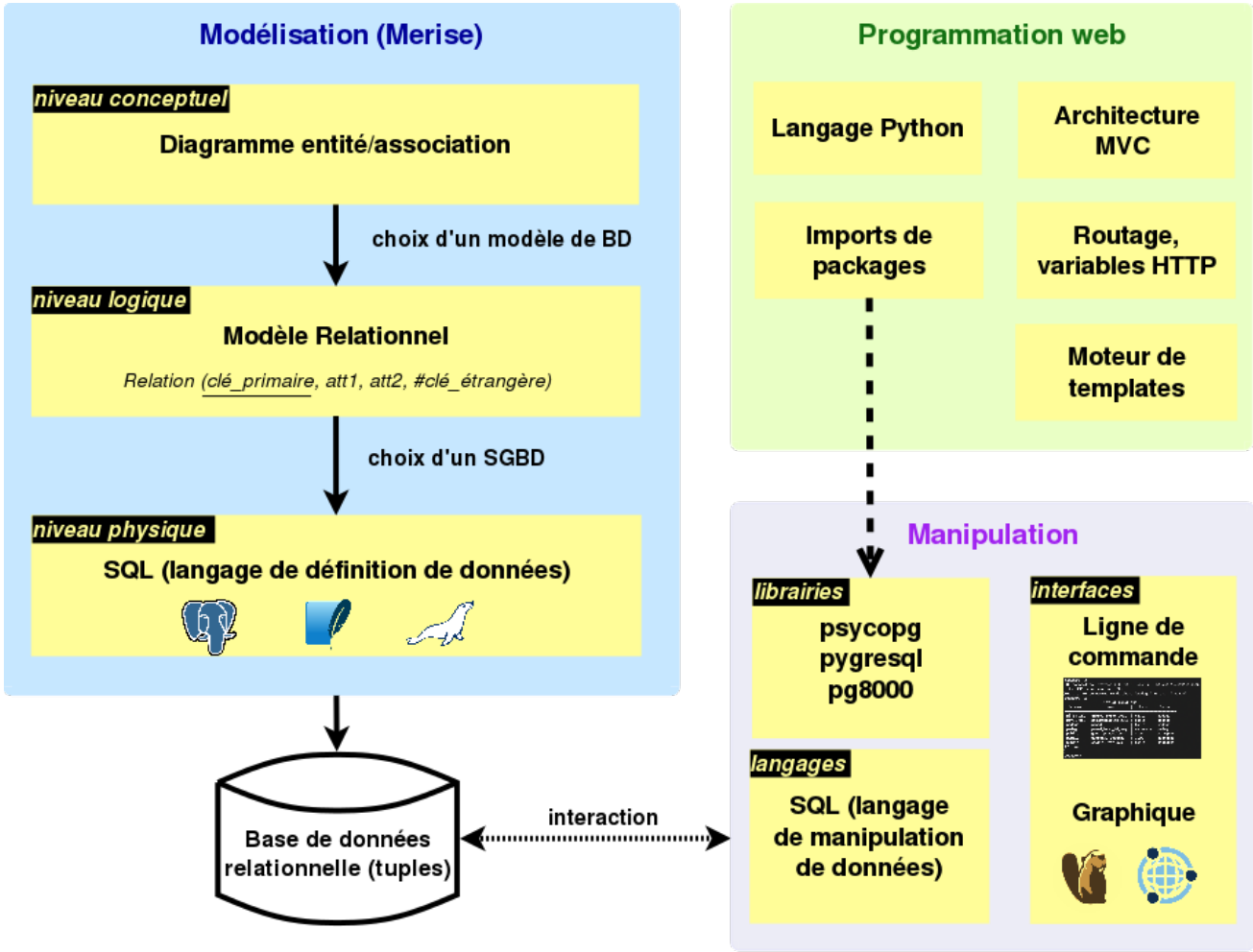
fabien.duchateau [at] univ-lyon1.fr

Université Claude Bernard Lyon 1

2024 - 2025

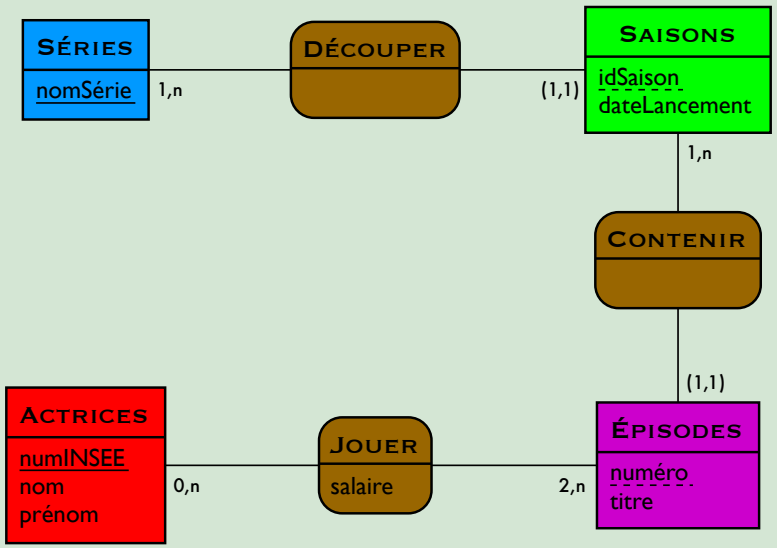
https://perso.liris.cnrs.fr/fabien.duchateau/BDW/

Positionnement dans BDW

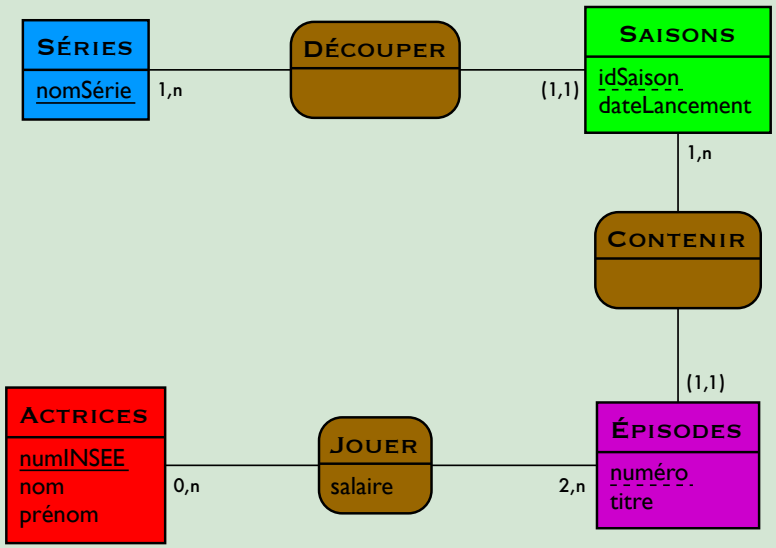


Ces diapositives utilisent le genre féminin (e.g., chercheuse, développeuses) plutôt que l'écriture inclusive (moins accessible, moins concise, et pas totalement inclusive)

Motivation - site sur les séries



Motivation - site sur les séries



Séries (nomSérie)
Saisons (idSaison, dateLancement, #nomSérie)
Épisodes (numéro, #idSaison, titre)
Actrices (numINSEE, nom, prénom)
Jouer (#numéro, #idSaison, #numINSEE, salaire)

Motivation - site sur les séries (2)

Réalisation d'un site web :

- Afficher les séries
- Ajouter une série
- Rechercher une série ou une actrice
- ...

Rappels

- Langage Python : programmation côté serveur
- Psycopg : interrogation d'une BD PostgreSQL
- Jinja : moteur de templates pour les pages HTML

Mais...

- comment organiser les fichiers et répertoires ?
- comment structurer le code ?
- comment traiter les différents types de page ?

Exemple de programmation d'une page web

afficher.py

```python
import psycopg
from jinja import Environment, PackageLoader

connexion = psycopg.connect(...)
try:
    cursor = connexion.cursor()
    cursor.execute("SELECT * FROM Series")
    instances = cursor.fetchall()
except psycopg.Error as e:
    print(f"Error : {e}")
return None

nb_instances = len(instances)

env = Environment(
    loader = PackageLoader("mon_site"),
)
template = env.get_template("afficher.html")
print(template.render(titre='Liste des séries',
    instances=instances, nb=nb_instances))
```

afficher.html

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

Quels problèmes ?

Exemple de programmation d'une page web

afficher.py

```python
import psycopg
from jinja import Environment, PackageLoader

connexion = psycopg.connect(...)
try:
    cursor = connexion.cursor()
    cursor.execute("SELECT * FROM Series")
    instances = cursor.fetchall()
except psycopg.Error as e:
    print(f"Error : {e}")
return None

nb_instances = len(instances)

env = Environment(
    loader = PackageLoader("mon_site"),
)
template = env.get_template("afficher.html")
print(template.render(titre='Liste des séries',
    instances=instances, nb=nb_instances))
```

afficher.html

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

Quels problèmes ?

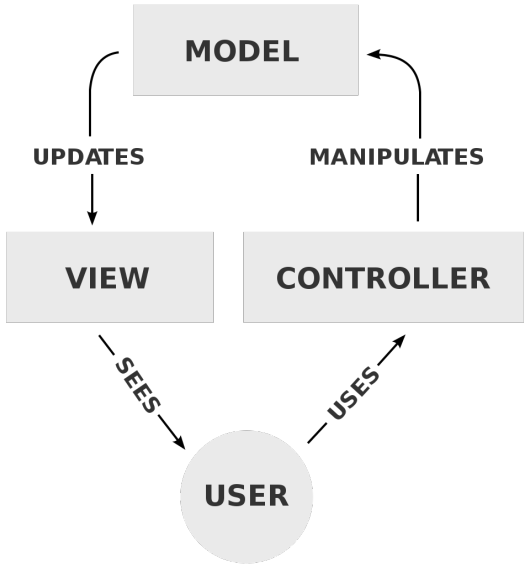
⇒ Besoin d'une architecture pour le code

Plan

Introduction à MVC

Architecture Modèle Vue Contrôleur (MVC) :

- Design pattern (origine en 1978)
- Utilisé par de nombreux sites web et frameworks (bonne pratique)
- Structuration et réutilisation du code, développement en parallèle, couplage faible
- Mais plus complexe et "fonctionnalités éparpillées"



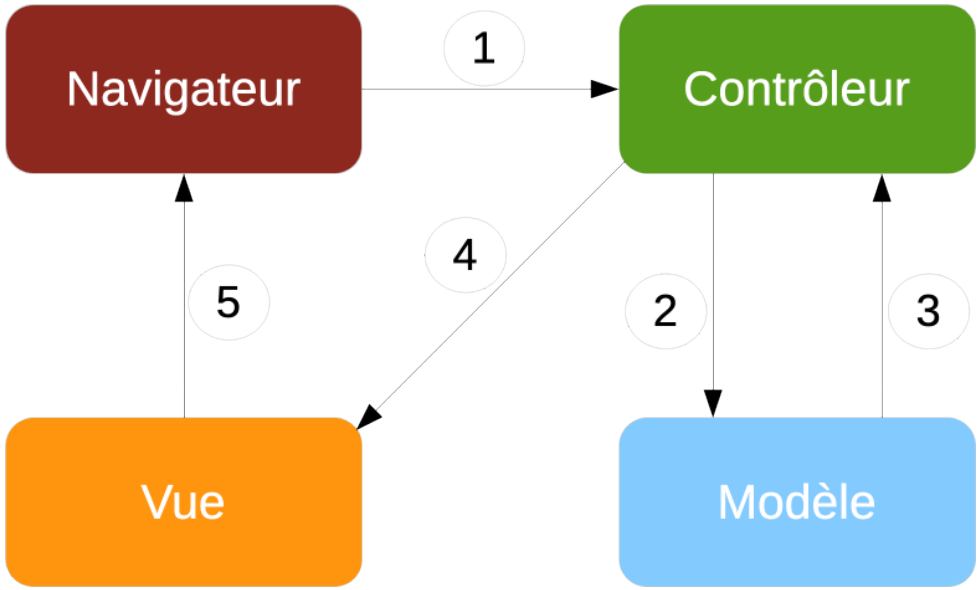
https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur

https://openclassrooms.com/fr/courses/7160741-ecrivez-du-code-python-maintenable/

Rôle des composants

- Modèle (structure dynamique des données) :
  - en lien avec la base de données
  - opérations de lecture, mise à jour, ajout, etc.
  - indépendant de la vue et du contrôleur
- Vue (interface graphique) :
  - éléments visuels et de présentation des données
  - éventuellement logique d'affichage des données
- Contrôleur :
  - lien entre modèle et vue : utilise le modèle, fournit des informations à la vue
  - traite les interactions avec l'internaute

Workflow entre composants MVC



1. *L'internaute envoie une requête (paramètres optionnels)*
2. *Le contrôleur vérifie les paramètres, et appelle le modèle*
3. *Le modèle interroge la base de données et retourne un résultat au contrôleur (e.g., des tuples, un booléen)*
4. *Le contrôleur sélectionne la vue à afficher et lui passe le résultat*
5. *La vue (code HTML) est envoyée au navigateur de l'internaute*

Exemple MVC - page afficher (simplifiée)

templates/afficher.html

```
{% extends "base.html" %}

{% block main_content %}
<h1>Liste des séries</h1>

<p>Il y a {{ nb_series }} séries.</p>
{% endblock %}
```

model/model.py

```
import psycopg

def count_series(connexion):
    try:
        cursor = connexion.cursor()
        cursor.execute("SELECT COUNT(*) AS nb
        ↪   FROM Series")
        return cursor.fetchone()  # un tuple
    except psycopg.Error as e:
        print(f"Error : {e}")
    return None
```

controleurs/afficher.py

```
from model.model import count_series

nb_series = count_series(connexion)[0]
```

Le contrôleur appelle la fonction `count_series` définie dans le modèle, et dont le résultat est stocké dans la variable `nb_series`. La vue associée (template) est chargée, et le contrôleur lui passe la variable `nb_series` pour affichage.

Routage

Chaque fonctionnalité du site (ou page) aura généralement :

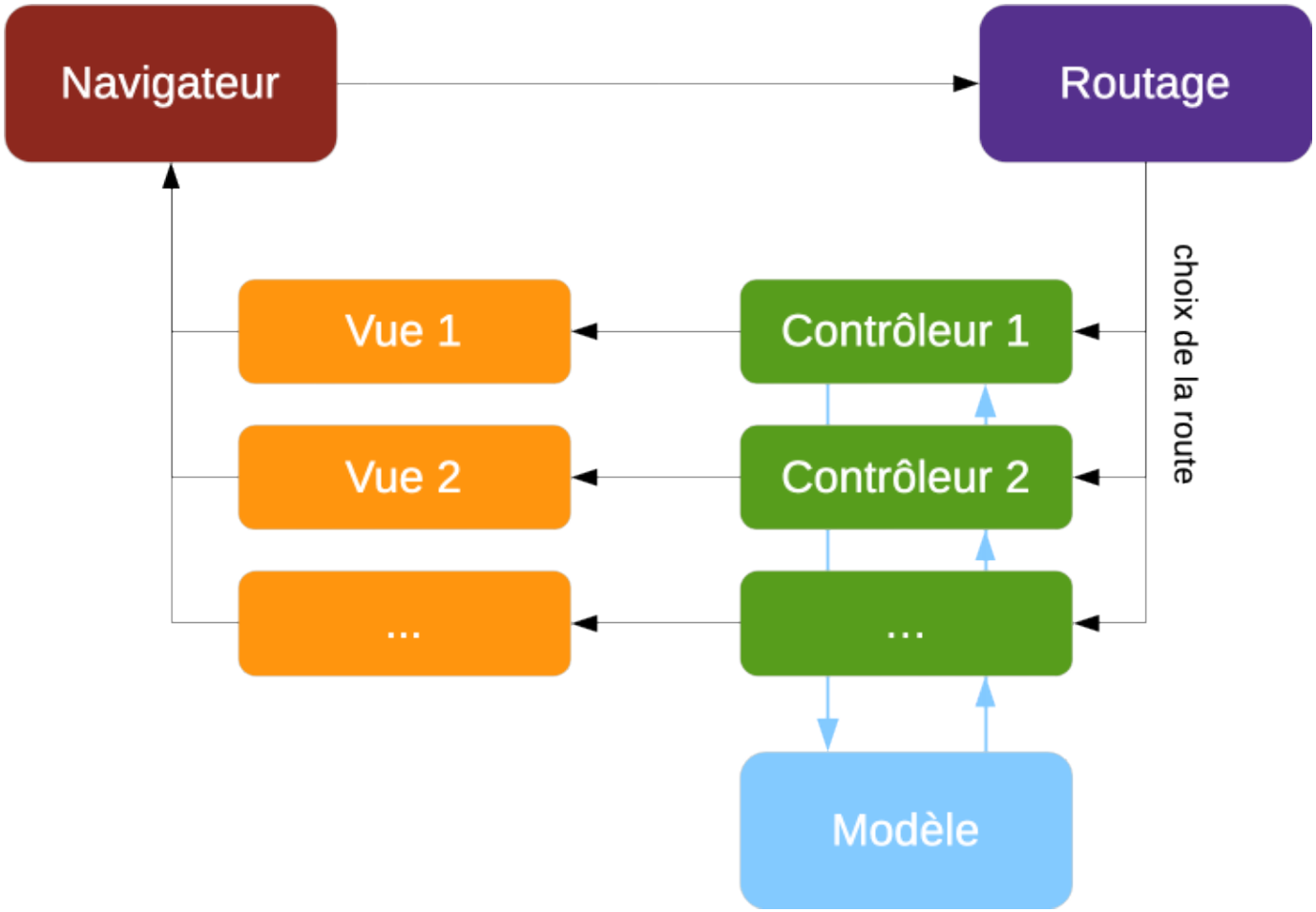
- Un contrôleur
- Une vue
- Éventuellement des méthodes dans le modèle

Comment lier la requête de l'internaute au bon contrôleur/vue ?

- Principe du **routage**, qui définit des routes (contrôleur + vue)
- Chaque route est associée à une (ou plusieurs) URL (e.g., `/afficher`) ou valeur soit abstraite (e.g., `page=1`) soit concrète (e.g., `page=afficher`)

Un contrôleur peut être vide (page statique) ou partagé par plusieurs pages.

Workflow MVC + routage



En réalité, un contrôleur peut effectuer différentes actions (e.g., afficher ou modifier une série). Chaque action peut ensuite avoir sa propre vue.

Routage par valeur

La valeur du contenu à charger est passée par un paramètre dans l'URL (`http://www.mon_site.fr/index.html?page=afficher`) :

- La valeur est récupérée et vérifiée (dans une BD, dans un fichier de routes)
- Si la valeur est valide, la route associée (contrôleur + vue) est chargée
- Sinon une page par défaut est chargée (généralement la page d'accueil)

Attention : ne pas directement charger des noms de fichier passés par valeur (e.g., `page=afficher.html`) $\Rightarrow$ problème de sécurité !

Routage par URL

La valeur du contenu à charger correspond à (un fragment de) l'URL (`http://www.mon_site.fr/afficher`) :

- Même traitement que le routage par valeur
- Permet d'avoir des URL plus compréhensibles (e.g., `mon_site.fr/series/123` plutôt que `mon_site.fr/afficher.py?type=series&id=123`)

Chaque framework ou serveur a son propre mécanisme de routage

Avec `bdw-server`, routage par URL avec un fichier de routes défini en TOML : le serveur exécute directement le contrôleur et appelle la vue/template à partir des routes

---

https://toml.io/

Exemple de routage - bdw-server

routes.toml

```toml
[[routes]]
url = ""  # pas de chemin, eg, mon_site.fr ou  mon_site.fr/
controleur = "controleurs/accueil.py"  # chemin vers controleur
template = "accueil.html"  # chemin vers vue/template

[[routes]]
url = "afficher" # matche les URL mon_site.fr/afficher ou
↪  mon_site.fr/afficher?id=1
controleur = "controleurs/afficher.py"
template = "afficher.html"
```

Une route comprend 3 éléments (en plus du `[[routes]]` qui permet de définir une nouvelle route) : `url` spécifie le chemin (fragment de l'URL à matcher), `controleur` et `template` spécifient le chemin vers les fichiers correspondants

En résumé

Nombreuses variantes du MVC :

- Modèle Vue Adaptateur (MVA)
- Modèle Vue Présentation (MVP)
- Modèle Vue Template (MVT)
- Modèle Vue Vue-Modèle (MVVM)
- ...

Dans notre contexte :

- Modèle = interrogation de la BD (`psycopg`)
- Vue = présentation de pages HTML (templates `Jinja`)
- Contrôleur = appel au modèle, traitement complexe (Python)

Plan

Généralités

Lors qu'un serveur web reçoit une requête, il en extrait des informations :

- Document demandé (URL, chemin, etc.)
- Données soumises par un formulaire
- Données liées à la réponse
- Données stockées en session
- ...

La disponibilité des variables HTTP et leur implémentation dépend du serveur web ! Dans la suite, syntaxe pour `bdw-server`

---

https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
Exemples de variables HTTP avec PHP : `$_GET`, `$_POST`, `$_SESSION`, etc.

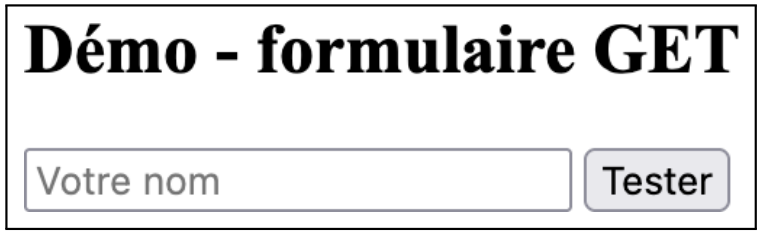
Variables HTTP - `bdw-server`

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

Toutes ces variables sont implémentées comme des **dictionnaires** et accessibles dans les contrôleurs et dans les templates

Sur `bdw-server`, la session est actuellement émulée (pas de cookie).

Exemple variable GET



routes.toml

```
[[routes]]
url = "exemple_get"
controleur = "controleurs/get.py"
template = "get.html"
```

model/model.py

```
# pas de méthode dans le modèle
```

controleurs/get.py

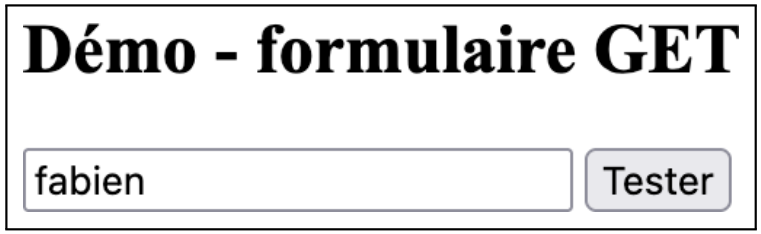
```
# controleur vide
```

templates/get.html

```
1  {% extends "base.html" %}
2
3  {% block main_content %}
4  <h2>Démo - formulaire GET</h2>
5
6  <form method="get">
7      <input name="field_name" type="text"
       ↪  placeholder="Votre nom" />
8      <input type="submit" value="Tester">
9  </form>
10
11 {% if GET['field_name'] %}
12     <p>Valeur reçue par GET : {{
       ↪  GET['field_name'][0] }}</p>
13 {% endif %}
14 {% endblock %}
```

Un formulaire pour saisir un prénom et afficher la valeur saisie quand le formulaire est soumis

Exemple variable GET



routes.toml

```
[[routes]]
url = "exemple_get"
controleur = "controleurs/get.py"
template = "get.html"
```

model/model.py

```
# pas de méthode dans le modèle
```

controleurs/get.py

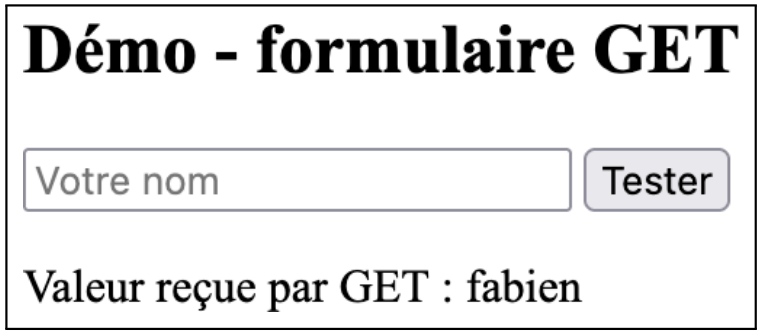
```
# controleur vide
```

templates/get.html

```
1  {% extends "base.html" %}
2
3  {% block main_content %}
4  <h2>Démo - formulaire GET</h2>
5
6  <form method="get">
7      <input name="field_name" type="text"
       ↪  placeholder="Votre nom" />
8      <input type="submit" value="Tester">
9  </form>
10
11 {% if GET['field_name'] %}
12     <p>Valeur reçue par GET : {{
       ↪  GET['field_name'][0] }}</p>
13 {% endif %}
14 {% endblock %}
```

Un formulaire pour saisir un prénom et afficher la valeur saisie quand le formulaire est soumis

Exemple variable GET



routes.toml

```
[[routes]]
url = "exemple_get"
controleur = "controleurs/get.py"
template = "get.html"
```

model/model.py

```
# pas de méthode dans le modèle
```

controleurs/get.py

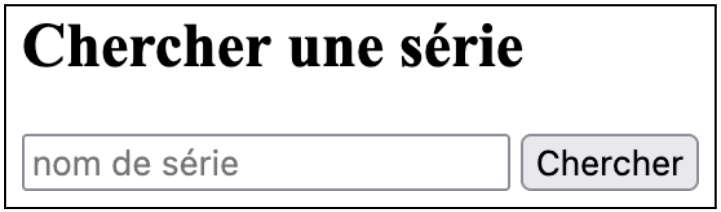
```
# controleur vide
```

templates/get.html

```
1  {% extends "base.html" %}
2
3  {% block main_content %}
4  <h2>Démo - formulaire GET</h2>
5
6  <form method="get">
7      <input name="field_name" type="text"
       ↪  placeholder="Votre nom" />
8      <input type="submit" value="Tester">
9  </form>
10
11 {% if GET['field_name'] %}
12     <p>Valeur reçue par GET : {{
       ↪  GET['field_name'][0] }}</p>
13 {% endif %}
14 {% endblock %}
```

Un formulaire pour saisir un prénom et afficher la valeur saisie quand le formulaire est soumis

Exemple variable POST



model/model.py

```python
def get_series_like(connexion, motif):
    like_pattern = '%' + motif + '%'
    try:
        cursor = connexion.cursor()
        query = sql.SQL("SELECT * FROM series
        ↪   WHERE nomsérie ILIKE
        ↪   {}").format(sql.Placeholder())
        cursor.execute(query, [like_pattern])
        return cursor.fetchall()
    except psycopg.Error as e:
        print(f"Error : {e}")
    return None
```

controleurs/find_serie.py

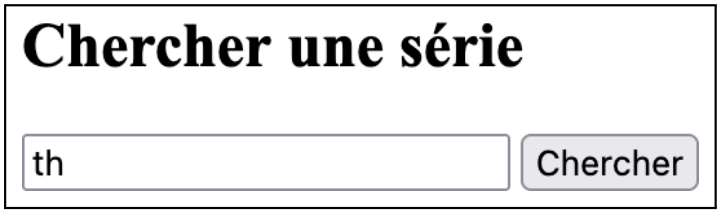
```python
from model.model import get_series_like

if 'nom_serie' in POST:  # formulaire soumis
    q = POST['nom_serie'][0]
    REQUEST_VARS['series'] =
    ↪   get_series_like(SESSION['CONNEXION'], q)
```

templates/find_serie.html

```
{% extends "base.html" %}

{% block main_content %}
<h2>Chercher une série</h2>

<form method="post">
    <input name="nom_serie" type="text"
    ↪   placeholder="nom de série" />
    <input type="submit" value="Chercher">
</form>

{% if POST['nom_serie'] %}
    <p>Recherche des séries contenant : {{
    ↪   POST['nom_serie'][0] }}</p>
    <ul>
    {% for instance in
    ↪   REQUEST_VARS['series'] %}
        <li>{{ instance[0] }}</li>
    {% endfor %}
    </ul>
{% endif %}

{% endblock %}
```

Un formulaire pour rechercher des séries et afficher le résultat

Exemple variable POST



model/model.py

```python
def get_series_like(connexion, motif):
    like_pattern = '%' + motif + '%'
    try:
        cursor = connexion.cursor()
        query = sql.SQL("SELECT * FROM series
        ↪   WHERE nomsérie ILIKE
        ↪   {}").format(sql.Placeholder())
        cursor.execute(query, [like_pattern])
        return cursor.fetchall()
    except psycopg.Error as e:
        print(f"Error : {e}")
    return None
```

controleurs/find_serie.py

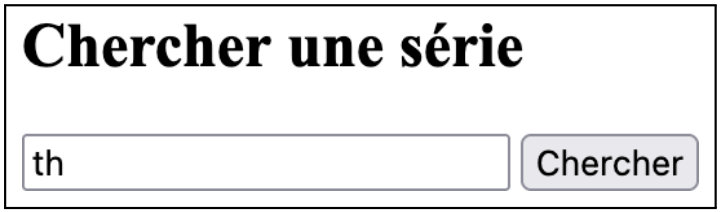
```python
from model.model import get_series_like

if 'nom_serie' in POST:  # formulaire soumis
    q = POST['nom_serie'][0]
    REQUEST_VARS['series'] =
    ↪   get_series_like(SESSION['CONNEXION'], q)
```

templates/find_serie.html

```
{% extends "base.html" %}

{% block main_content %}
<h2>Chercher une série</h2>

<form method="post">
    <input name="nom_serie" type="text"
    ↪   placeholder="nom de série" />
    <input type="submit" value="Chercher">
</form>

{% if POST['nom_serie'] %}
    <p>Recherche des séries contenant : {{
    ↪   POST['nom_serie'][0] }}</p>
    <ul>
    {% for instance in
    ↪   REQUEST_VARS['series'] %}
        <li>{{ instance[0] }}</li>
    {% endfor %}
    </ul>
{% endif %}

{% endblock %}
```

Un formulaire pour rechercher des séries et afficher le résultat

Exemple variable POST



model/model.py

```python
def get_series_like(connexion, motif):
    like_pattern = '%' + motif + '%'
    try:
        cursor = connexion.cursor()
        query = sql.SQL("SELECT * FROM series
        ↪   WHERE nomsérie ILIKE
        ↪   {}").format(sql.Placeholder())
        cursor.execute(query, [like_pattern])
        return cursor.fetchall()
    except psycopg.Error as e:
        print(f"Error : {e}")
    return None
```

controleurs/find_serie.py

```python
from model.model import get_series_like

if 'nom_serie' in POST:  # formulaire soumis
    q = POST['nom_serie'][0]
    REQUEST_VARS['series'] =
    ↪   get_series_like(SESSION['CONNEXION'], q)
```

templates/find_serie.html

```
{% extends "base.html" %}

{% block main_content %}
<h2>Chercher une série</h2>

<form method="post">
    <input name="nom_serie" type="text"
    ↪   placeholder="nom de série" />
    <input type="submit" value="Chercher">
</form>

{% if POST['nom_serie'] %}
    <p>Recherche des séries contenant : {{
    ↪   POST['nom_serie'][0] }}</p>
    <ul>
    {% for instance in
    ↪   REQUEST_VARS['series'] %}
        <li>{{ instance[0] }}</li>
    {% endfor %}
    </ul>
{% endif %}

{% endblock %}
```

Un formulaire pour rechercher des séries et afficher le résultat

Un mot sur la sécurité

Les données saisies par un internaute peuvent être "douteuses" : ne pas les utiliser directement !

```
https://duckduckgo.com/?q=chocolat
https://duckduckgo.com/?q='%3B%20DROP%20TABLE%
20uneTable%3B%20--
```

Votre code doit donc vérifier et valider ces variables :

- Côté client (HTML, éventuellement JavaScript)
- Côté serveur (Python, Psycopg)

Variable requête `REQUEST_VARS`

Le contrôleur peut avoir besoin de passer des données au template pour la requête courante :

- Données récupérées après interrogation du modèle (e.g., liste de séries)
- Message (informatif, d'erreur, etc.)
- ...

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

---

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

Exemple variable REQUEST_VARS

controleurs/check_int.py

```python
if 'nombre' in GET:  # formulaire soumis
    try:
        nombre = int(GET['nombre'][0])
        REQUEST_VARS['message'] = f"La valeur {nombre} est bien un nombre."
        REQUEST_VARS['message_class'] = "alert-success"
    except ValueError:
        REQUEST_VARS['message'] = f"Erreur : la valeur {GET['nombre'][0]} n'est pas un nombre entier."
        REQUEST_VARS['message_class'] = "alert-error"
```

templates/message.html

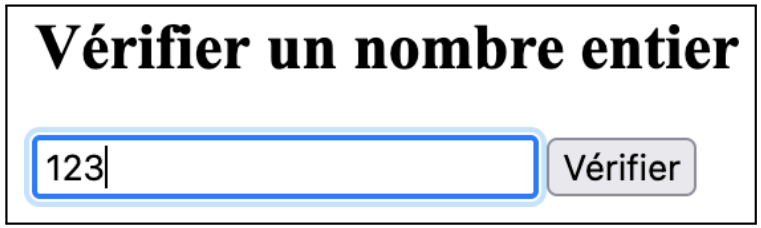
```
{% if REQUEST_VARS['message'] %}
    <div class="{{ REQUEST_VARS['message_class'] }}"> {{ REQUEST_VARS['message'] }}</div>
{% endif %}
```

templates/check_int.html

```
{% extends "base.html" %}

{% block main_content %}
<h2>Vérifier un nombre entier</h2>

<form method="get">
    <input name="nombre" type="text" placeholder="Entrez un entier" />
    <input type="submit" value="Vérifier">
</form>

{% if GET['nombre'] %}
    <p>Valeur : {{ GET['nombre'][0] }}</p>
    {% include 'message.html' %}
{% endif %}
{% endblock %}
```

Un formulaire pour saisir une valeur et vérifier qu'elle soit un entier

Exemple variable `REQUEST_VARS`



controleurs/check_int.py

```python
if 'nombre' in GET:  # formulaire soumis
    try:
        nombre = int(GET['nombre'][0])
        REQUEST_VARS['message'] = f"La valeur {nombre} est bien un nombre."
        REQUEST_VARS['message_class'] = "alert-success"
    except ValueError:
        REQUEST_VARS['message'] = f"Erreur : la valeur {GET['nombre'][0]} n'est pas un nombre entier."
        REQUEST_VARS['message_class'] = "alert-error"
```

templates/message.html

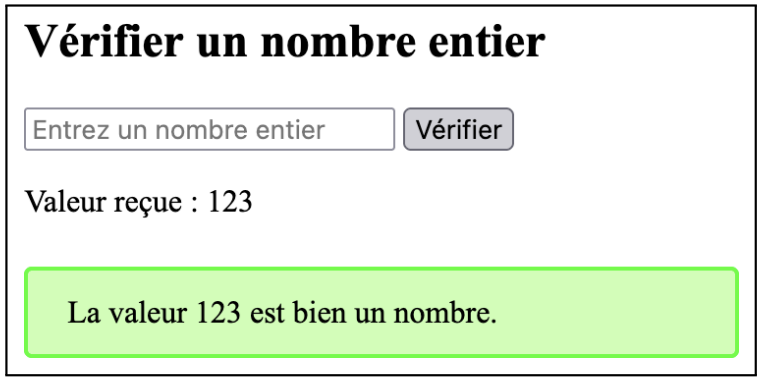
```jinja
{% if REQUEST_VARS['message'] %}
    <div class="{{ REQUEST_VARS['message_class'] }}"> {{ REQUEST_VARS['message'] }}</div>
{% endif %}
```

templates/check_int.html

```jinja
{% extends "base.html" %}

{% block main_content %}
<h2>Vérifier un nombre entier</h2>

<form method="get">
    <input name="nombre" type="text" placeholder="Entrez un entier" />
    <input type="submit" value="Vérifier">
</form>

{% if GET['nombre'] %}
    <p>Valeur : {{ GET['nombre'][0] }}</p>
    {% include 'message.html' %}
{% endif %}
{% endblock %}
```

Un formulaire pour saisir une valeur et vérifier qu'elle soit un entier

Exemple variable REQUEST_VARS



controleurs/check_int.py

```python
if 'nombre' in GET:  # formulaire soumis
    try:
        nombre = int(GET['nombre'][0])
        REQUEST_VARS['message'] = f"La valeur {nombre} est bien un nombre."
        REQUEST_VARS['message_class'] = "alert-success"
    except ValueError:
        REQUEST_VARS['message'] = f"Erreur : la valeur {GET['nombre'][0]} n'est pas un nombre entier."
        REQUEST_VARS['message_class'] = "alert-error"
```

templates/message.html

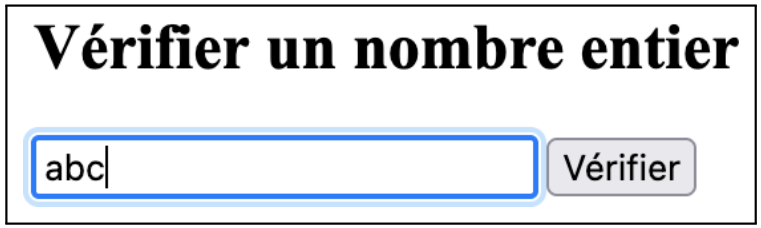
```
{% if REQUEST_VARS['message'] %}
    <div class="{{ REQUEST_VARS['message_class'] }}"> {{ REQUEST_VARS['message'] }}</div>
{% endif %}
```

templates/check_int.html

```
{% extends "base.html" %}

{% block main_content %}
<h2>Vérifier un nombre entier</h2>

<form method="get">
    <input name="nombre" type="text" placeholder="Entrez un entier" />
    <input type="submit" value="Vérifier">
</form>

{% if GET['nombre'] %}
    <p>Valeur : {{ GET['nombre'][0] }}</p>
    {% include 'message.html' %}
{% endif %}
{% endblock %}
```

Un formulaire pour saisir une valeur et vérifier qu'elle soit un entier

Exemple variable REQUEST_VARS



controleurs/check_int.py

```
1  if 'nombre' in GET:  # formulaire soumis
2      try:
3          nombre = int(GET['nombre'][0])
4          REQUEST_VARS['message'] = f"La valeur
               ↪ {nombre} est bien un nombre."
5          REQUEST_VARS['message_class'] =
               ↪ "alert-success"
6      except ValueError:
7          REQUEST_VARS['message'] = f"Erreur :
               ↪ la valeur {GET['nombre'][0]}
               ↪ n'est pas un nombre entier."
8          REQUEST_VARS['message_class'] =
               ↪ "alert-error"
```

templates/message.html

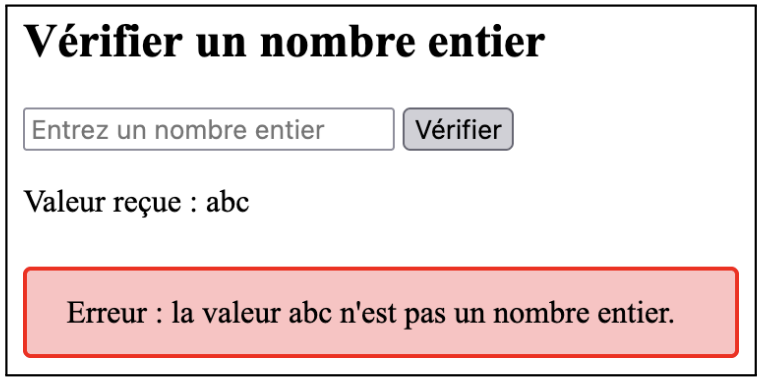
```
1  {% if REQUEST_VARS['message'] %}
2      <div class="{{
           ↪ REQUEST_VARS['message_class'] }}"> {{
           ↪ REQUEST_VARS['message'] }}</div>
3  {% endif %}
```

templates/check_int.html

```
1  {% extends "base.html" %}
2
3  {% block main_content %}
4  <h2>Vérifier un nombre entier</h2>
5
6  <form method="get">
7      <input name="nombre" type="text"
           ↪ placeholder="Entrez un entier" />
8      <input type="submit" value="Vérifier">
9  </form>
10
11 {% if GET['nombre'] %}
12     <p>Valeur : {{ GET['nombre'][0] }}</p>
13     {% include 'message.html' %}
14 {% endif %}
15 {% endblock %}
```

Un formulaire pour saisir une valeur et vérifier qu'elle soit un entier

Exemple variable REQUEST_VARS



controleurs/check_int.py

```python
if 'nombre' in GET:  # formulaire soumis
    try:
        nombre = int(GET['nombre'][0])
        REQUEST_VARS['message'] = f"La valeur {nombre} est bien un nombre."
        REQUEST_VARS['message_class'] = "alert-success"
    except ValueError:
        REQUEST_VARS['message'] = f"Erreur : la valeur {GET['nombre'][0]} n'est pas un nombre entier."
        REQUEST_VARS['message_class'] = "alert-error"
```

templates/message.html

```jinja
{% if REQUEST_VARS['message'] %}
    <div class="{{ REQUEST_VARS['message_class'] }}"> {{ REQUEST_VARS['message'] }}</div>
{% endif %}
```

templates/check_int.html

```jinja
{% extends "base.html" %}

{% block main_content %}
<h2>Vérifier un nombre entier</h2>

<form method="get">
    <input name="nombre" type="text" placeholder="Entrez un entier" />
    <input type="submit" value="Vérifier">
</form>

{% if GET['nombre'] %}
    <p>Valeur : {{ GET['nombre'][0] }}</p>
    {% include 'message.html' %}
{% endif %}
{% endblock %}
```

Un formulaire pour saisir une valeur et vérifier qu'elle soit un entier

Variable de session

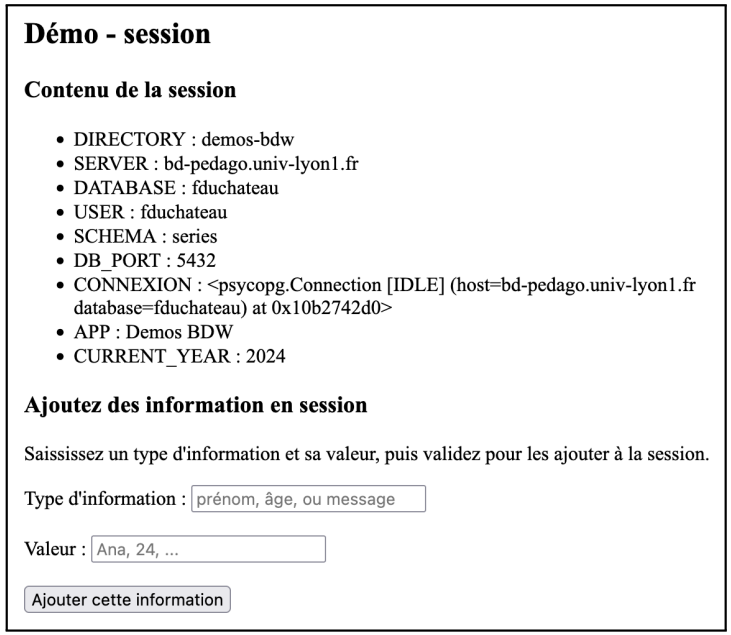
Il peut être utile de conserver des informations d'une page sur l'autre :

- 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·e internaute (i.e., un navigateur sur une machine)

Si l'internaute est authentifié, il est recommandé de permettre une déconnexion afin d'effacer le contenu de la session, la connexion à la BD, etc.

Exemple variable SESSION



controleurs/session.py

```python
if 'field_info' in GET and 'field_value' in GET:
    SESSION[GET['field_info'][0]] =
    ↪ GET['field_value'][0]
```

templates/session.html

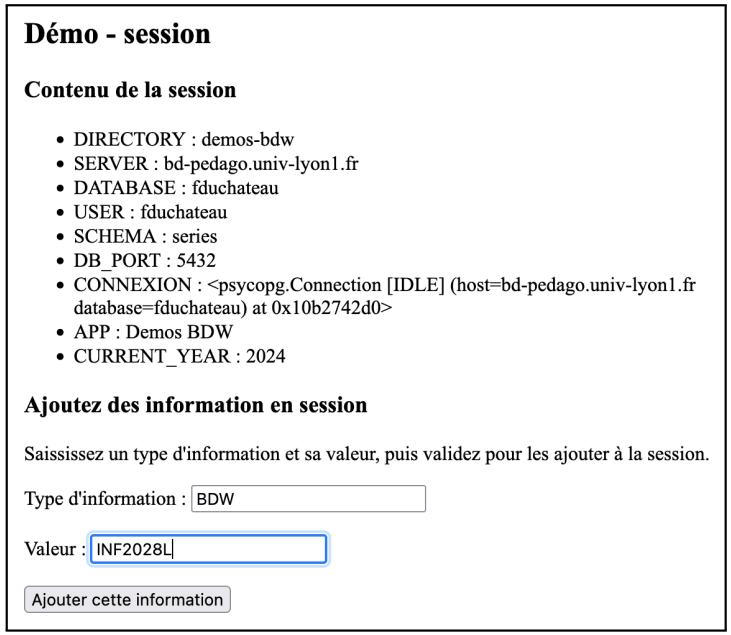
```jinja
{% extends "base.html" %}

{% block main_content %}
<h2>Démo - session</h2>

<h3>Contenu de la session</h3>
{% include 'afficher_session.html' %}

<h3>Ajoutez une information en session</h3>
<form method="get">
    <p>Saisissez ...</p>
    <label for="field_info">Type
    ↪ d'information</label> : <input
    ↪ id="field_info" name="field_info"
    ↪ type="text" />
    <label for="field_value">Valeur</label> :
    ↪ <input id="field_value"
    ↪ name="field_value" type="text" />
    <input type="submit" value="Ajouter">
</form>
{% endblock %}
```

templates/afficher_session.html

```jinja
<ul>
{% for key, value in SESSION.items() %}
    <li>{{ key }} : {{ value }}</li>
{% endfor %}
</ul>
```

Un formulaire pour saisir une information persistante en session

Exemple variable SESSION



controleurs/session.py

```python
if 'field_info' in GET and 'field_value' in GET:
    SESSION[GET['field_info'][0]] =
    ↪ GET['field_value'][0]
```

templates/session.html

```jinja
{% extends "base.html" %}

{% block main_content %}
<h2>Démo - session</h2>

<h3>Contenu de la session</h3>
{% include 'afficher_session.html' %}

<h3>Ajoutez une information en session</h3>
<form method="get">
    <p>Saisissez ...</p>
    <label for="field_info">Type
    ↪ d'information</label> : <input
    ↪ id="field_info" name="field_info"
    ↪ type="text" />
    <label for="field_value">Valeur</label> :
    ↪ <input id="field_value"
    ↪ name="field_value" type="text" />
    <input type="submit" value="Ajouter">
</form>
{% endblock %}
```

templates/afficher_session.html

```jinja
<ul>
{% for key, value in SESSION.items() %}
    <li>{{ key }} : {{ value }}</li>
{% endfor %}
</ul>
```

Un formulaire pour saisir une information persistante en session

Exemple variable SESSION

templates/afficher_session.html

```jinja
<ul>
{% for key, value in SESSION.items() %}
    <li>{{ key }} : {{ value }}</li>
{% endfor %}
</ul>
```

controleurs/session.py

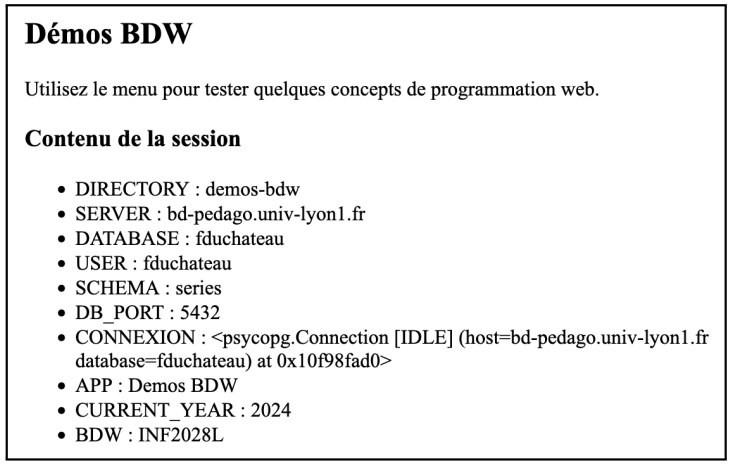
```python
if 'field_info' in GET and 'field_value' in GET:
    SESSION[GET['field_info'][0]] =
    ↪ GET['field_value'][0]
```

templates/session.html

```jinja
{% extends "base.html" %}

{% block main_content %}
<h2>Démo - session</h2>

<h3>Contenu de la session</h3>
{% include 'afficher_session.html' %}

<h3>Ajoutez une information en session</h3>
<form method="get">
    <p>Saisissez ...</p>
    <label for="field_info">Type
    ↪ d'information</label> : <input
    ↪ id="field_info" name="field_info"
    ↪ type="text" />
    <label for="field_value">Valeur</label> :
    ↪ <input id="field_value"
    ↪ name="field_value" type="text" />
    <input type="submit" value="Ajouter">
</form>
{% endblock %}
```

Un formulaire pour saisir une information persistante en session

Exemple variable SESSION



Démos BDW

Utilisez le menu pour tester quelques concepts de programmation web.

Contenu de la session

- DIRECTORY : demos-bdw
- SERVER : bd-pedago.univ-lyon1.fr
- DATABASE : fduchateau
- USER : fduchateau
- SCHEMA : series
- DB_PORT : 5432
- CONNEXION : <psycopg.Connection [IDLE] (host=bd-pedago.univ-lyon1.fr database=fduchateau) at 0x10f98fad0>
- APP : Demos BDW
- CURRENT_YEAR : 2024
- BDW : INF2028L

Sur la page d'accueil, l'information ajoutée "BDW : INF2028L" est toujours présente

templates/afficher_session.html

```
1 <ul>
2 {% for key, value in SESSION.items() %}
3     <li>{{ key }} : {{ value }}</li>
4 {% endfor %}
5 </ul>
```

controleurs/session.py

```
1 if 'field_info' in GET and 'field_value' in GET:
2     SESSION[GET['field_info'][0]] =
      ↪ GET['field_value'][0]
```

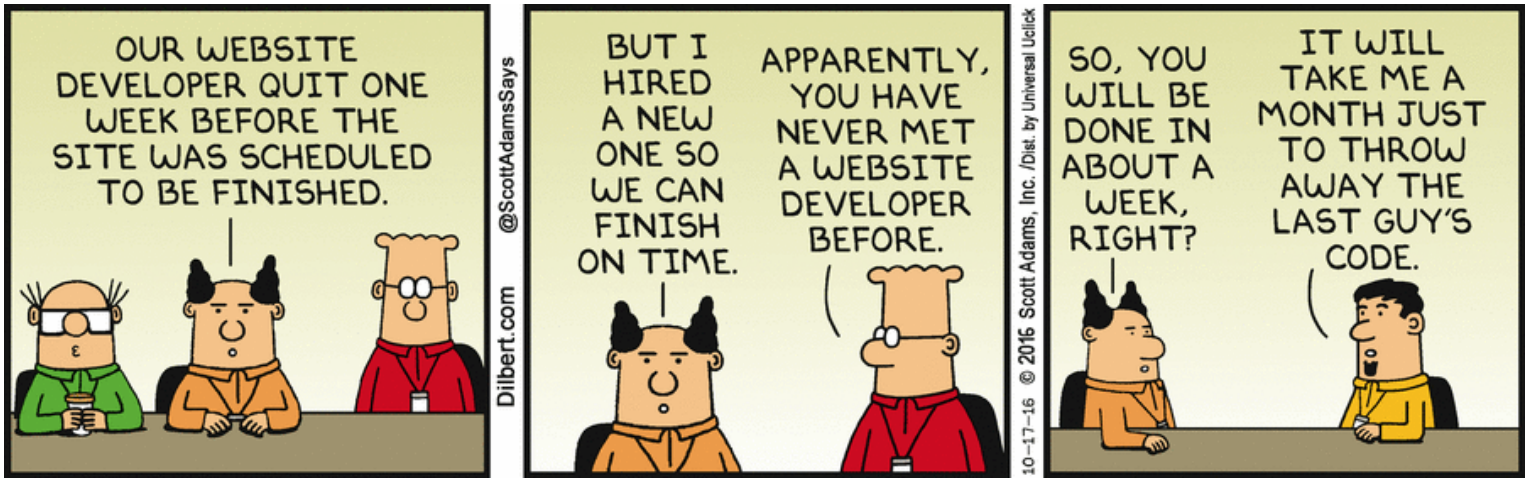
templates/session.html

```
1  {% extends "base.html" %}
2
3  {% block main_content %}
4  <h2>Démo - session</h2>
5
6  <h3>Contenu de la session</h3>
7  {% include 'afficher_session.html' %}
8
9  <h3>Ajoutez une information en session</h3>
10 <form method="get">
11     <p>Saisissez ...</p>
12     <label for="field_info">Type
       ↪ d'information</label> : <input
       ↪ id="field_info" name="field_info"
       ↪ type="text" />
13     <label for="field_value">Valeur</label> :
       ↪ <input id="field_value"
       ↪ name="field_value" type="text" />
14     <input type="submit" value="Ajouter">
15 </form>
16 {% endblock %}
```

Un formulaire pour saisir une information persistante en session

En résumé

- Un serveur web fournit des informations via des variables
- `GET` et `POST` pour récupérer les valeurs d'un formulaire (pensez aux vérifications)
- `REQUEST_VARS` pour échanger des données temporaires entre contrôleur et template
- `SESSION` pour stocker des informations persistantes



Plan

Organisation d'un site web

Différentes manières d'organiser un site, mais besoin de stocker :

- Les routes
- Le code des pages (souvent réparti dans différents fichiers)
- Les images
- Le ou les fichiers de styles CSS
- Les fichiers spéciaux (e.g., configuration, constantes)

Organisation d'un 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
```

Fichier `init.py`

Ce ficher permet d'initialiser des variables ou constantes utilisées sur différentes pages du site. Elles sont chargées au démarrage du serveur et stockées en session

init.py

```
from datetime import datetime

SESSION['APP'] = "Serial Critique"
SESSION['BASELINE'] = "Critiquez vos séries !"
SESSION['HISTORIQUE'] = dict()
SESSION['CURRENT_YEAR'] = datetime.now().year
```

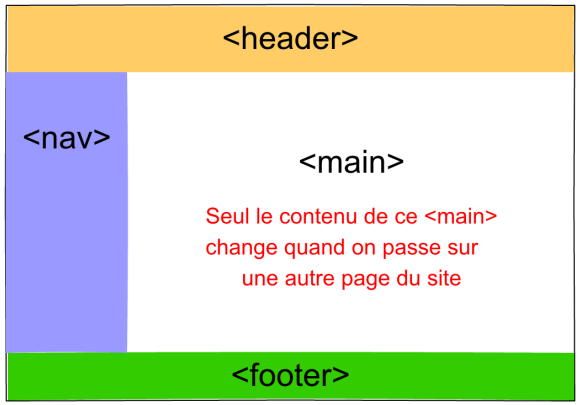
Exemple de fichier `init.py` pour serial-critique : on stocke le nom du site et son slogan, un dictionnaire vide pour l'historique des actions et l'année courante (avec le package `datetime`)

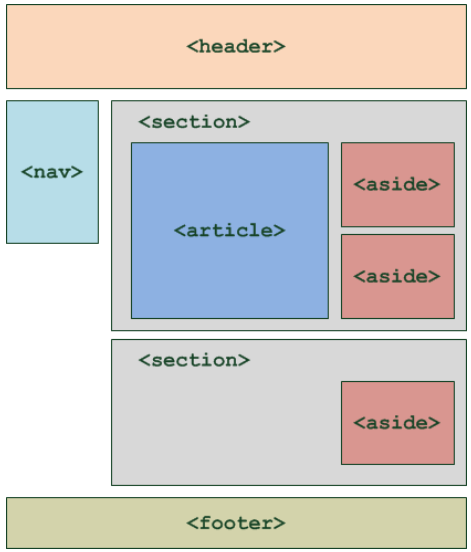
Si vous modifiez votre fichier `init.py`, vous devez arrêter et relancer `bdw-server` (pas seulement le redémarrer).

Comment structurer une page ?

Toutes les pages d'un site respectent en général un même *design*
On distingue donc :

- Les parties communes (identiques sur toutes les pages)
- La (ou les) parties spécifiques à une page (i.e., fonctionnalité)





Exemples de design : serial-critique (gauche) et un plus complexe

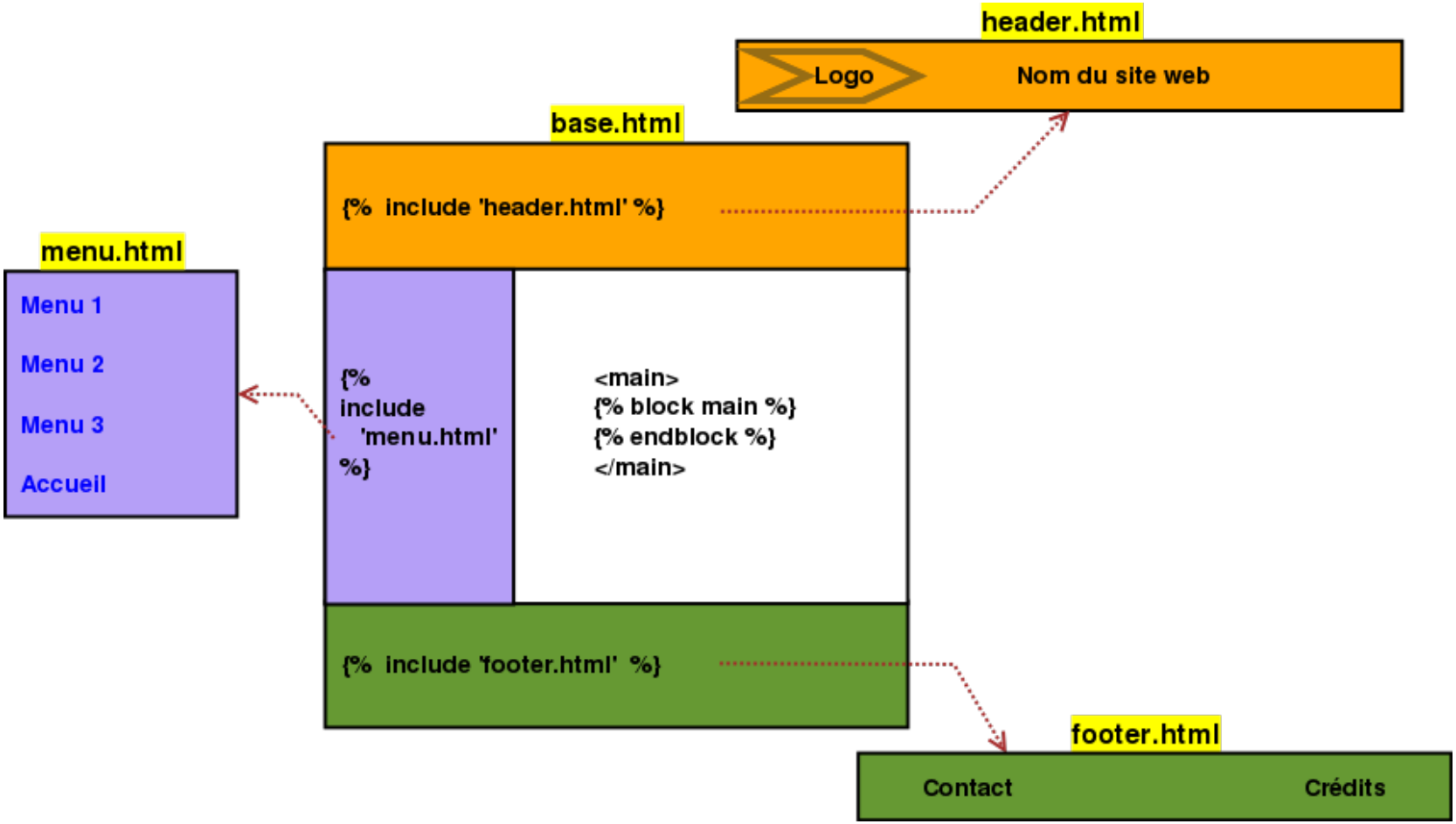
Distribution du code

- Parties communes : essentiellement du code HTML, inclus par le template de base (réutilisation, factorisation)
- Parties spécifiques (fonctionnalité, balise `<main>`) réparties selon l'architecture MVC avec 3 cas :
  - contenu statique (i.e., que du code HTML)
  - contenu dynamique sans interaction (contenu évolutif sans action humaine)
  - contenu dynamique avec interaction (contenu évolutif par une action de l'internaute)

Dans la suite, description des différentes situations

Les parties communes peuvent parfois dépendre de la page demandée (e.g., menu avec un sous-menu qui diffère selon la page courante).

Parties communes



Utilisation de l'inclusion de Jinja : `base.html` est le template parent qui inclut les parties communes (`menu.html`, etc.)

Parties communes - exemple de code

templates/base.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <link type="text/css" href="style.css">
  <title>Serial Critique</title>
</head>

<body>
    {% include 'header.html' %}
    <div id="div-central">
        {% include 'menu.html' %}
        <main>
        {% block main %}{% endblock %}
        </main>
    </div>
    {% include 'footer.html' %}
</body>
</html>
```

inclusion

inclusion

inclusion

templates/header.html

```
<header>
    <img src="sheep.png">
    <h1>Serial Critique</h1>
    <strong>Critiquez vos séries !</strong>
</header>
```

templates/menu.html

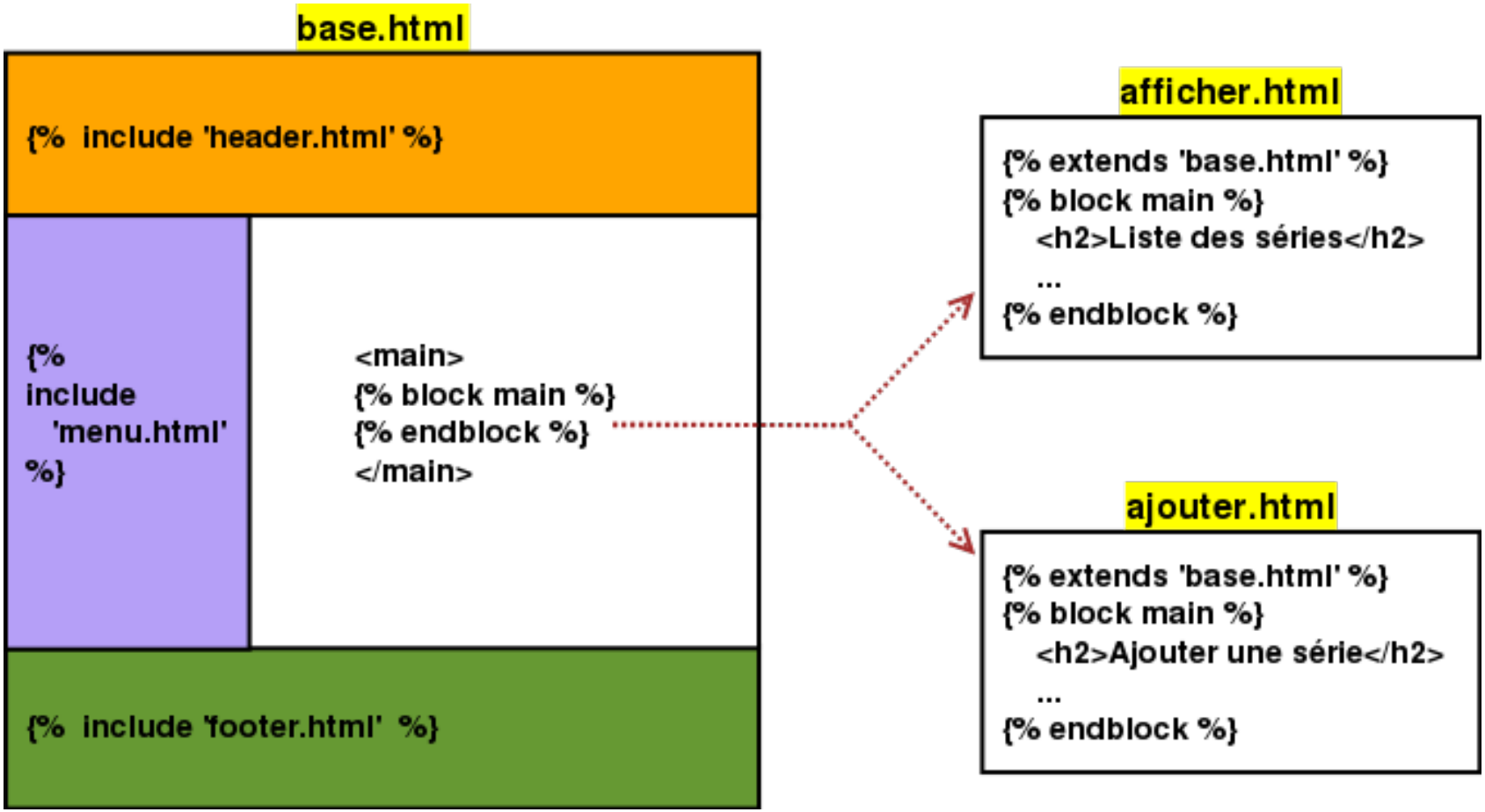
```
<nav>
    <a href="/">Accueil</a>
    <a href="afficher">Afficher</a>
    <a href="ajouter">Ajouter série</a>
    ...
</nav>
```

templates/footer.html

```
<footer>
    <span>Licence CC BY-NC-SA</span>
    <span>BDW - UCBL</span>
</footer>
```

Code (simplifié) de serial-critique pour le template de base et les parties communes (incluses)

Parties spécifiques



Utilisation de l'héritage de Jinja : les parties spécifiques (e.g., `afficher.html`) étendent le template parent (`base.html`) et redéfinissent son bloc `main`

Parties spécifiques - exemple de code (template)

templates/base.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <link type="text/css" href="style.css">
  <title>Serial Critique</title>
</head>

<body>
    {% include 'header.html' %}
    <div id="div-central">
        {% include 'menu.html' %}
        <main>
        {% block main %}{% endblock %}
        </main>
    </div>
    {% include 'footer.html' %}
</body>
</html>
```

extension

templates/afficher.html

```
{% extends "base.html" %}

{% block main %}
<h2>Liste des séries</h2>
<ul>
{% for instance in REQUEST_VARS['series'] %}
    <li>{{ instance[0] }}</li>
{% endfor %}
</ul>
...
{% endblock %}
```

extension

templates/ajouter.html

```
{% extends "base.html" %}

{% block main %}
<h2>Ajout d'une série</h2>
<form method="post">
...
</form>
{% include 'message.html' %}
{% endblock %}
```

Code (simplifié) de serial-critique pour le template de base et les parties spécifiques (enfant)

Parties spécifiques - MVC

La partie spécifique d'une page (balise `<main>`) représente (généralement) une fonctionnalité

Avec l'architecture MVC, le code se répartit entre modèle, contrôleur et vue (template, cf héritage précédemment)

La répartition du code d'une partie spécifique selon l'architecture MVC dépend du type de contenu :

- Contenu statique (e.g., page d'accueil)
- Contenu dynamique sans interaction (e.g., affichage des séries de la BD)
- Contenu dynamique avec interaction (e.g., recherche d'une série en utilisant un formulaire)

Exemple de contenu statique

model/model.py

```python
# pas de méthode pour accueil
```

controleurs/accueil.py

```python
# controleur vide pour accueil
```

templates/accueil.html

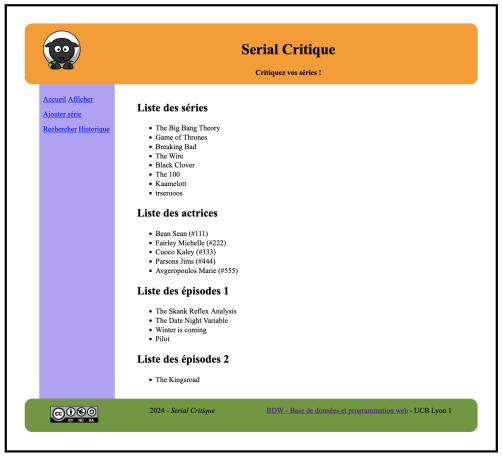
```jinja
{% extends "base.html" %}

{% block main %}
<h2>Le site de critique sur les séries pour le cours de BDW</h2>

<p id="parag_description">
Vous pouvez gérer des séries, des épisodes, et les acteur.ice.s.<br>
Utilisez le menu à gauche ...
</p>

<p>Que faire sur ce site ?</p>
<ul>
    <li>Afficher les données de la base</li>
    <li>Ajouter une série</li>
    <li>...</li>
</ul>

<div>
    <img src=".../img/top-series.jpg" />
</div>

{% include 'message.html' %}
{% endblock %}
```

Page accueil (simplifiée) : pas de méthode dans le modèle, contrôleur vide et template avec l'intégralité du contenu

Exemple de contenu dynamique sans interaction

Page afficher (simplifiée) : le contrôleur utilise le modèle pour interroger la base de données, puis transmet le résultat au template

Exemple de contenu dynamique sans interaction



model/model.py

```python
def execute_select_query(connexion, query, params=[]):
    try:
        cursor = connexion.cursor()
        cursor.execute(query, params)
        return cursor.fetchall()
    except psycopg.Error as e:
        logger.error(e)
    return None

def get_instances(connexion, nom_table):
    query = sql.SQL('SELECT * FROM {table}').format(
        table=sql.Identifier(nom_table), )
    return execute_select_query(connexion, query)
```

controleurs/afficher.py

```python
from model.model_pg import get_instances

REQUEST_VARS['series'] = get_instances(SESSION['CONNEXION'], 'series')
REQUEST_VARS['actrices'] = get_instances(SESSION['CONNEXION'], 'actrices')
```

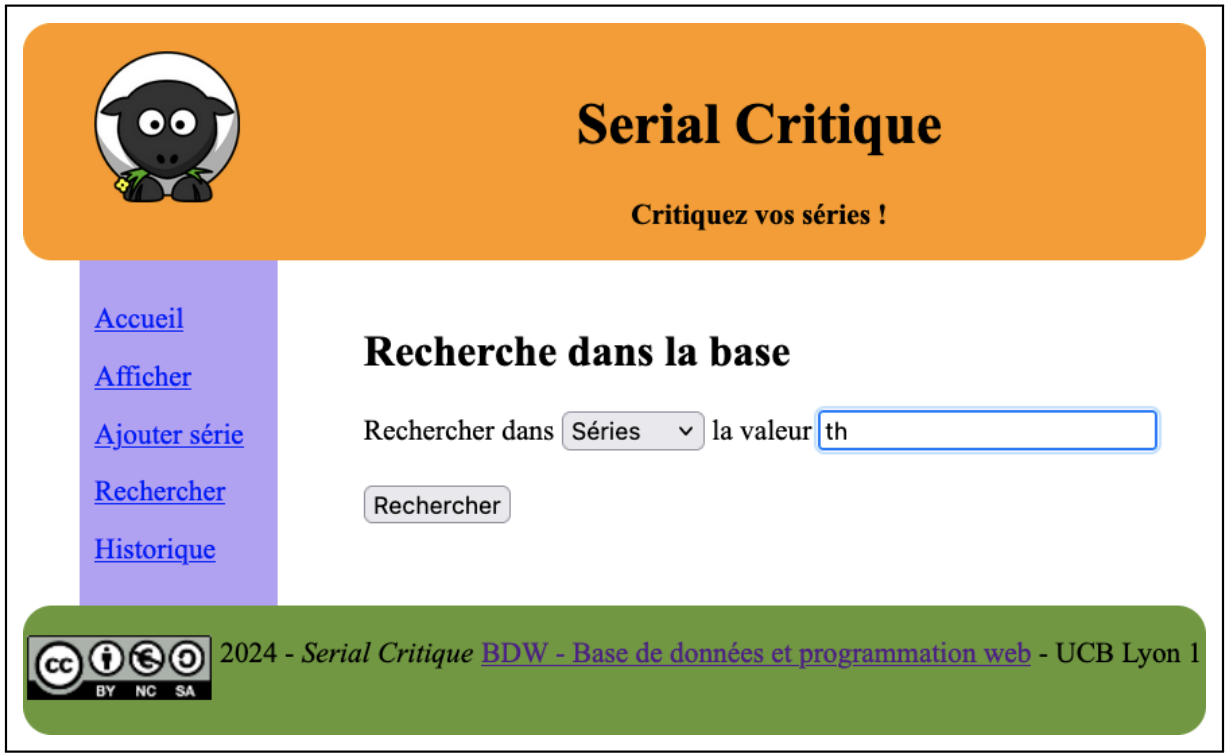
templates/afficher.html

```
{% extends "base.html" %}

{% block main_content %}
<h2>Liste des séries</h2>
<ul>
{% for instance in REQUEST_VARS['series'] %}
    <li>{{ instance[0] }}</li>
{% endfor %}
</ul>

<h2>Liste des actrices</h2>
<ul>
{% for instance in REQUEST_VARS['actrices']  %}
    <li>{{ instance[1] }} {{ instance[2] }} (#{{ instance[0] }})</li>
{% endfor %}
</ul>
{% endblock %}
```

Exemple de contenu dynamique avec interaction

Page rechercher (simplifiée) : par défaut, le template permet d'afficher uniquement le formulaire

Exemple de contenu dynamique avec interaction



Page rechercher (simplifiée) : si le formulaire est soumis, le contrôleur traite le formulaire et utilise le modèle pour interroger la base de données

Exemple de contenu dynamique avec interaction

Page rechercher (simplifiée) : en fin de traitement (du formulaire soumis), le template affiche de nouveau le formulaire et le résultat de la recherche (ou un message)

Exemple de contenu dynamique avec interaction

model/model.py

```python
def get_series_like(connexion, motif):
    like_pattern = '%' + motif + '%'
    try:
        cursor = connexion.cursor()
        query = sql.SQL("SELECT * FROM series WHERE nomsérie ILIKE {}").format(sql.Placeholder())
        cursor.execute(query, [like_pattern])
        return cursor.fetchall()
    except psycopg.Error as e:
        print(f"Error : {e}")
    return None
```

controleurs/rechercher.py

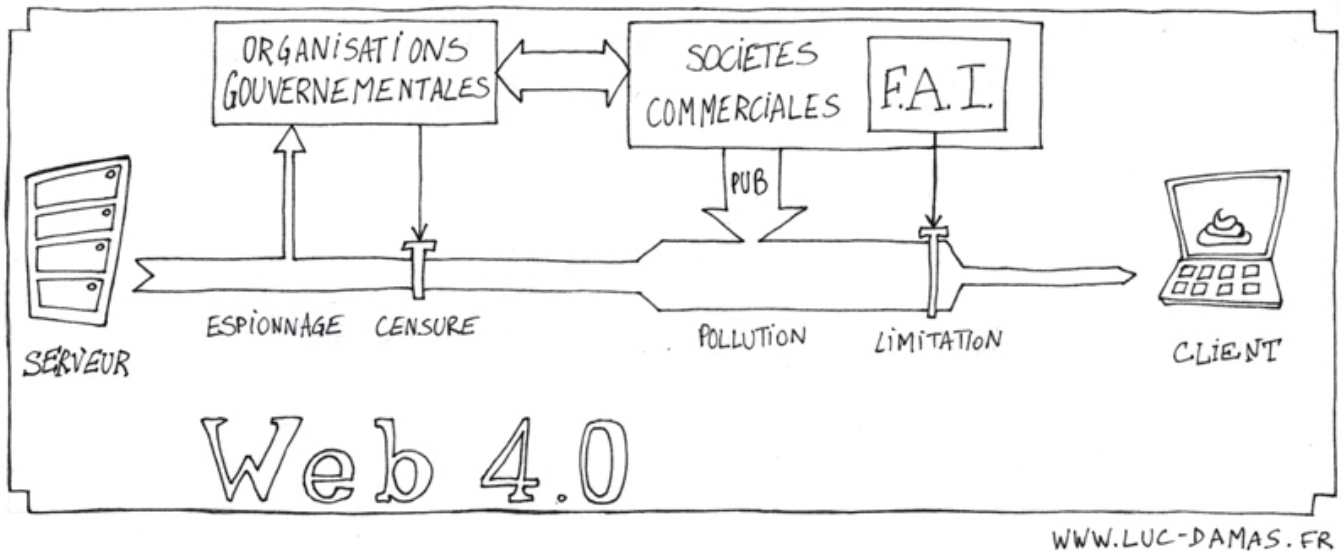
```python
from model.model_pg import get_series_like

if POST and 'bouton_valider' in POST:
    term = POST['valeur'][0]
    result = get_series_like(SESSION['CONNEXION'], term)
    if result is None or len(result) == 0:
        REQUEST_VARS['message'] = f"Aucun résultat pour {term}."
        REQUEST_VARS['message_class'] = "alert-warning"
    else:  # résultat en session
        REQUEST_VARS['result'] = result
```

templates/rechercher.html

```jinja
{% extends "base.html" %}

{% block main %}
<h2>Recherche dans la base</h2>

<form method="post">
    Rechercher dans
    <select name="nom_table">
        <option value="series">Séries</option>
    </select> la valeur
    <input type="text" name="valeur" required />
    <input type="submit" name="bouton_valider" value="Rechercher"/>
</form>

{% include 'message.html' %}

{% if POST and POST.bouton_valider and REQUEST_VARS.result %}
    <ul>
    {% for instance in REQUEST_VARS.result  %}
        <li>
        {% for value in instance %}
            {{ value }}
        {% endfor %}
        </li>
    {% endfor %}
</ul>
{% endif %}
{% endblock %}
```

En résumé

- Template parent : inclusion des parties communes, héritage pour les parties spécifiques
- Parties spécifiques découpées selon l'architecture MVC et en fonction du type de contenu (statique, dynamique avec ou sans interaction)

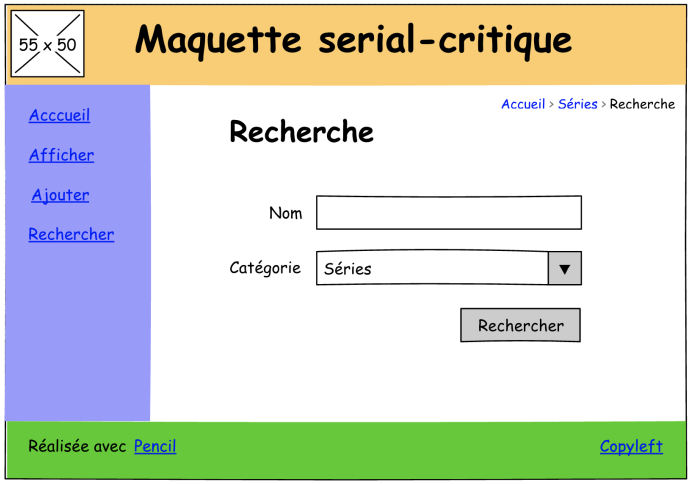


Plan

Conception du site

Réflexion sur le design des pages (e.g., maquettes) :

- Pencil, draw.io, Excalidraw, tldraw
- Quant-UX, Moqups, Figjam, Mockflow, ...



Utilisation de `bdw-server` :

- Architecture MVC (avec des vues sous forme de templates)
- Fichier de routage et fichier d'initialisation
- Possible de copier `serial-critique` comme base de projet

Alternatives (mais en autonomie !) : Flask, FastAPI, Django, ...

Gestion de projet

- Dans le binôme, certaines tâches sont faites ensemble (modélisation, template parent) et d'autres en parallèle (programmation des différentes fonctionnalités)
  - bien réfléchir à la structuration pour faciliter la répartition
  - ne pas perdre trop de temps sur le design !

- Utilisation de la forge (http://forge.univ-lyon1.fr/) :
  - code partagé, travail collaboratif, versioning
  - travail sauvegardé (plus d'excuse de clé USB perdue)
  - apprentissage de Git (crucial en L3 et Master)
  - clients graphiques GitKraken, Sourcetree, GitExtensions, etc.

---

http://fr.wikipedia.org/wiki/Conception_de_site_web

http://rogerdudler.github.io/git-guide/

http://learngitbranching.js.org/

Bilan

Structuration d'un site web avec MVC et un moteur de templates

- Variables d'environnement du serveur
- Différents types de contenu (statique, dynamique)

Un bilan de l'UE pour finir...