|  |  |  |
| --- | --- | --- |
| **Logo_Lycée** | **« Todo List »** | logo%20ac%20orl%E9ans%20toursDescription : Description : Description : pemDescription : Description : Description : pem |





**SUJET DU TP**

**Mots-clés** : page Web dynamique, base de données, SGBD, SQLite,

SQL, Python, framework Flask.

**PROBLÉMATIQUE**

Comment extraire des données à l’aide d’un SGBD, les mettre à jour et les présenter dans une page Web ?

**CONDITIONS DE DÉROULEMENT DE L’ACTIVITÉ**

|  |  |  |
| --- | --- | --- |
| **Phases de travail** | **Objectifs** | **Activités** |
| ***1) Mise en situation*** | ***Comprendre*** *le cahier des charges du site à réaliser.* | ***Lecture*** *de la mise en situation.* |
| ***2) Étude de***  ***la problématique*** | ***Présenter*** *des données issues d’une base SQLite dans une page Web et les* ***mettre à jour****.* | ***Interroger*** *et mettre à jour une base de données SQLite avec des requêtes SQL intégrées à du code Python.*  ***Créer*** *un serveur Web avec un micro Framework Python.*  ***Créer*** *des modèles Web.* |
| ***3) Synthèse*** |

**SIGLES UTILISÉS**

|  |  |  |  |
| --- | --- | --- | --- |
| **Remarque** | Avertissement Important | Informations Information | DocumentÉditeur **VSCode** |

**PRÉPARATION DU DOSSIER DE TRAVAIL**

|  |  |
| --- | --- |
|  | Organisez votre dossier de travail en suivant les indications données en **annexe** à la fin de ce document. |

**RESSOURCES DOCUMENTAIRES, LOGICIELS ET MATÉRIELS**

PC, IDE VSCode avec les extensions SQLite et Python 3.10.11 ou supérieur, framework Flask, répertoire du projet.

**WEBOGRAPHIE**

Présentation du TP <https://urlr.me/eRD3c8>

Le module Python sqlite3 <https://bit.ly/3pwHouj>

Le framework Flask https://urlr.me/Fr6Bt2

Table des matières

[1 Mise en situation 3](#_Toc191051282)

[1.1 Site Web dynamique ? 3](#_Toc191051283)

[1.2 Cahier des charges de la page « Ma liste des choses à faire » 3](#_Toc191051284)

[2 Étude de la problématique 4](#_Toc191051285)

[2.1 La base de données « Todo » 4](#_Toc191051286)

[2.2 Interagir en Python avec une base de données 4](#_Toc191051287)

[2.2.1 Création d’une connexion 4](#_Toc191051288)

[2.2.2 Exécution des requêtes 5](#_Toc191051289)

[2.2.3 Déconnexion 6](#_Toc191051290)

[2.3 Création d’un serveur Web avec le framework Flask 6](#_Toc191051291)

[2.3.1 Prise en main 6](#_Toc191051292)

[2.4 Affichage des données issues de la base « Todo » dans une page Web 7](#_Toc191051293)

[2.4.1 Solution 1 : pages Web dans le code Python 7](#_Toc191051294)

[2.4.2 Solution 2 : modèle Web (*template*) 7](#_Toc191051295)

[2.5 Mise à jour de la base « Todo » avec des formulaires 8](#_Toc191051296)

[2.5.1 Suppression d’éléments 8](#_Toc191051297)

[2.5.2 Ajout d’éléments 9](#_Toc191051298)

[2.6 Ajout d’une feuille de style 9](#_Toc191051299)

[3 Synthèse : idée de mini projet 10](#_Toc191051300)

[Annexe : connexion à votre compte et préparation du dossier de travail sur le serveur NAS\_SIN 10](#_Toc191051301)

**Table des illustrations**

[**Figure 1** : code source pour un site Web 3](#_Toc191051205)

[**Figure 2** : modèle client-serveur 3](#_Toc191051206)

[**Figure 3** : exemple de page - "Ma liste des choses à faire" 3](#_Toc191051207)

[**Figure 4** : modèle client-serveur du projet 3](#_Toc191051208)

[**Figure 5 :** dossier final du projet "TodoList" 4](#_Toc191051209)

[**Figure 6 :** SQLite Explorer dans VSCode 4](#_Toc191051210)

[**Figure 7 :** table "todo" initiale 4](#_Toc191051211)

[**Figure 8** : résultat attendu dans la fenêtre interactive 4](#_Toc191051212)

[**Figure 9 :** liste de courses dans la fenêtre interactive 5](#_Toc191051213)

[**Figure 10 :** page accueil 6](#_Toc191051214)

[**Figure 11 :** page 2 6](#_Toc191051215)

[**Figure 12 :** liste des courses dans le navigateur (partielle) 7](#_Toc191051216)

[**Figure 13** : table todo dans le navigateur (partielle) 8](#_Toc191051217)

[**Figure 14 :** suppression d’items dans la liste des choses à faire 8](#_Toc191051218)

[**Figure 15** : formulaire pour l'ajout d’items 9](#_Toc191051219)

[**Figure 16** : résultat final 9](#_Toc191051220)

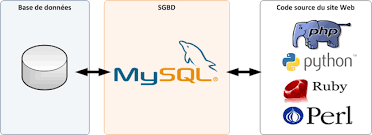
[**Figure 17** : MCD de la base "Séjour" 10](#_Toc191051221)

[**Figure 18** : MLD de la base "Séjour" 10](#_Toc191051222)

# Mise en situation

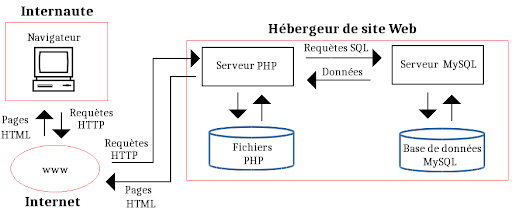
## Site Web dynamique ?

Les sites Web que nous consultons sont la plupart du temps des sites **dynamiques**. Ces sites sont constitués de pages dont le contenu peut varier en fonction de **données** (heure, nom de l’utilisateur, formulaire rempli par l’utilisateur, etc.). Ces données sont stockées dans une **base de données** gérée par un **S**ystème de **G**estion de **B**ases de **D**onnées (**SGBD**). Le **code** écrit en PHP, Ruby, **Python** etc. se charge de répondre aux requêtes envoyées par le client en renvoyant des pages HTML contenant des données situées dans la base de données.



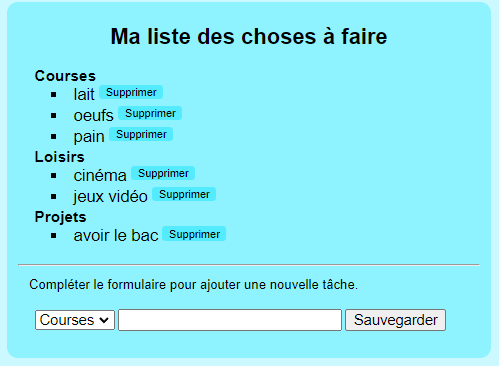
**Figure 1** : code source pour un site Web

Le schéma ci-dessous représente l’organisation classique du modèle client-serveur sur le Web.



**Figure 2** : modèle client-serveur

## Cahier des charges de la page « Ma liste des choses à faire »

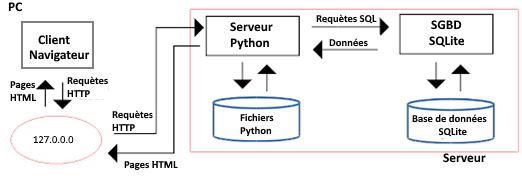


**Figure 3** : exemple de page - "Ma liste des choses à faire"

|  |
| --- |
| Cette page doit :  - **présenter** la liste des « Choses à faire » dans des **catégories** (Courses, etc.),  - **proposer** la **suppression** de ce qui a été fait,  - **proposer** l’**ajout** de nouvelles catégories et de nouvelles « choses à faire » dans ces catégories. |

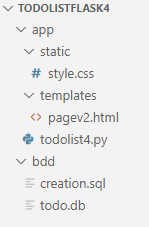
Le site « Todo List » sera codé en **Python** (côté serveur) et en **HTML**, **CSS** (côté client). Le serveur sera réalisé avec le framework **Flask**. Les données seront sauvegardées dans une base de données **SQLite**.

Le schéma de la figure 2 devient :



**Figure 4** : modèle client-serveur du projet

# Étude de la problématique



**Figure 5 :** dossier final du projet "TodoList"

Pour répondre au cahier des charges, vous allez écrire du code en Python, SQL et HTML afin :

- d’interroger et de mettre à jour une base de données.

- d’afficher les données issues de cette base dans un navigateur.

**Ceci sera détaillé dans les prochains paragraphes.**

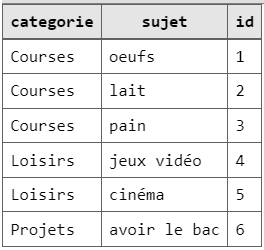
|  |
| --- |
| La version finale du site contenu dans le dossier « **todolistFlask4** ». Il est organisé comme ci-contre.   * Le répertoire **bdd** contient la base de données et son code de création. Ce code permet de reconstruire la base en cas de problème. * **static** contient une feuille de style. * **template** contient les modèles de page Web. |

* **Les fichiers du TP ont été répartis dans 3 dossiers.**

**Chargez le dossier « todolistFlask1\_2 » et seulement ce dossier dans l’IDE VSCode à l’aide d’un glisser-déplacer.**

## La base de données « Todo »

|  |  |
| --- | --- |
| La liste des « Choses à faire » est sauvegardée dans la base « Todo ». Cette base de données, contenue dans le fichier **todo.db,** est constituée de la table **todo** organisée comme sur la copie d’écran ci-contre.  **1.** **Affichez** l’explorateur **SQLite**  Touche **F1** → **SQLite : Open Database** et sélectionnez ***todo.db***.  Ouvrez l’explorateur SQLite (en bas et à gauche de la fenêtre) comme sur la copie d’écran ci-contre. | **Figure 6 :** SQLite Explorer dans VSCode |
| **2. Visualisez** le contenu de la table ***todo*** en cliquant sur le symbole Lirequi apparaît au survol de la souris dans **SQLITE EXPLORER**.  Vous devez obtenir le résultat ci-dessous. | |



**Figure 7 :** table "todo" initiale

## Interagir en Python avec une base de données



|  |  |
| --- | --- |
| Informations | Le **module** **sqlite3** permet d’interagir en Python avec un **SGBD SQLite**. On communique avec la base « Todo » en utilisant les **classes** et les **méthodes** définies dans ce module.  Pour importer ce module dans un fichier, il suffit d’écrire ***import sqlite3.***  Généralement, interagir avec une base de données nécessite de :  **- créer une connexion ;**  **- créer un curseur pour exécuter des requêtes ;**  **- se déconnecter.** |

**Ces actions seront implémentées dans les prochains paragraphes.**

### Création d’une connexion



**Figure 8** : résultat attendu dans la fenêtre interactive

|  |  |
| --- | --- |
| Informations | Pour interagir avec la base « Todo », on doit d’abord créer une [connexion](https://docs.python.org/fr/3/library/sqlite3.html#sqlite3.Connection) avec la méthode ***connect()*** et lui passer l’emplacement du fichier de stockage de la base de données (.db) en paramètre. Si ce fichier n’existe pas, il est créé. |

Entrez la commande suivante sous ***# 2. Connexion à la base « todo »*** dans le fichier ***testreq.py***.

***testreq.py***

|  |  |
| --- | --- |
| Document | connexion = sqlite3.connect("bdd/todo.db") |

### Exécution des requêtes

|  |  |
| --- | --- |
| Informations | Pour exécuter les requêtes SQL avec du code Python, on se sert d’un **objet** *curseur*récupéré en faisant appel à la méthode *cursor* de l’objet *connexion***.** |

* **Création d’un objet curseur**

Entrez la commande suivante sous ***# 3. Création d’un curseur*** dans le fichier *testreq.py*.

|  |  |
| --- | --- |
| Document | *curseur = connexion.cursor()* |

* **Modifications apportées à une table**

|  |  |
| --- | --- |
| Informations | Lorsque l’on effectue des **modifications** sur une table (**insertion**, **mise à jour** ou encore **suppression** d’éléments), celles-ci ne sont pas automatiquement validées. Ainsi, sans validation, les modifications ne sont pas effectuées dans la base et ne sont donc pas visibles par les autres connexions. Pour résoudre cela, il faut utiliser la méthode **commit** de l’objet de type *Connection*. |

* **Exécution d’une requête**

La méthode ***execute()*** appliquée au curseur exécute **UNE** requête. On souhaite ajouter du chocolat à la liste de courses. Entrez les commandes suivantes sous ***# 4a) UNE requête d’INSERTION*** dans le fichier *testreq.py*.

|  |  |
| --- | --- |
| Document | *donnees = ("Courses", "chocolat")*  *curseur.execute("INSERT INTO todo (categorie, sujet) VALUES (?, ?)", donnees)*  *connexion.commit()* |

**Testez** le code entré jusqu’à présent en exécutant **dans l’ordre** les cellules **1 → 4a**.

|  |  |
| --- | --- |
| Avertissement | Pour voir les effets de la requête, il faut mettre à jour la base « Todo*»* dans **SQLITE EXPLORER** (VSCode) avec l’icône  et recharger la table *todo* avec . |

* **Exécution de plusieurs requêtes**

La méthode ***executemany()*** appliquée au curseur exécute **PLUSIEURS** requêtes. Entrez les commandes suivantes sous

***# 4b) Plusieurs requêtes d’INSERTION*** dans le fichier *testreq.py et testez-les (4b seul)*.

|  |  |
| --- | --- |
| Document | *donnees = [("Loisirs", "voile"),("Courses", "sucre"),("Projets", "voyage")]*  *curseur.executemany("INSERT INTO todo (categorie, sujet) VALUES (?, ?)", donnees)*  *connexion.commit()* |

* **Parcours des enregistrements**

|  |  |
| --- | --- |
| Informations | Pour obtenir des éléments contenus dans la base de données, on **exécute une requête** puis on récupère le résultat de la requête à l’aide de **l’objet de type *cursor***. |

Les méthodes *fetchone*(), f*etchmany*() ou *fetchall*() permettent de **récupérer le résultat** de la requête. Ce résultat est fourni sous la forme d’un **tuple**, d’une **liste de tuples** ou **None** s’il n’y en a pas.

* ***fetchone()*** récupère **un seul** résultat.
* ***fetchmany(n)*** récupère **n résultats** en une opération.

Remarque : Le nombre de résultats prend par défaut la valeur de l’attribut ***arraysize*** du curseur.

* ***fetchall()*** récupère **tous les résultats** de la requête.

*Pour voir l’effet de ces trois commandes*, entrez le code suivant dans le fichier *testreq.py* sous ***# 4c) Parcourir des enregistrements.***

|  |  |
| --- | --- |
| Document | *curseur.execute("SELECT \* FROM todo")*  *resultat = curseur.fetchone()*  *print(f"Cas 1. Le seul résultat récupéré est {resultat}")*  ***Résultat attendu***    *n = 3*  *curseur.execute("SELECT \* FROM todo")*  *resultat = curseur.fetchmany(n)*  *print(f"Cas 2. Les {n} résultats récupérés sont {resultat}")*  *curseur.execute("SELECT \* FROM todo")*  *resultat = curseur.fetchall()*  *print(f"Cas 3. Tous les résultats : {resultat}")*  ***Résultat partiel attendu***    **Figure 9 :** liste de courses dans la fenêtre interactive |

**Activité 1** : on souhaite récupérer la liste des courses à faire dans une variable *lstCourses* et l’afficher dans la fenêtre interactive comme ci-contre.

**Complétez** le code Python sous ***# Activité 1*** dans le fichier *testreq.py.* ***Testez-le*** et complétez le document réponse 1 (**DR1)**.

### Déconnexion

|  |  |
| --- | --- |
| Informations | On applique la méthode **close()** à la connexion pour la fermer. |

## Création d’un serveur Web avec le framework Flask



Pour afficher les données issues de la base et saisir les modifications à réaliser dans un navigateur, il faut créer un serveur HTTP. Ceci sera réalisé avec le framework Flask.

|  |  |
| --- | --- |
| Informations | **Flask** est un framework Web [**WSGI**](http://www.wsgi.org/) (**W**eb **S**erver **G**ateway **I**nterface) léger pour Python. Il est conçu pour permettre une prise en main rapide et facile, avec la possibilité de passer à des applications complexes. |

### Prise en main

Pour comprendre la philosophie de Flask nous allons mettre en œuvre un exemple. Entrez le code ci-dessous dans le fichier *hello.py.*

**hello.py**

|  |  |
| --- | --- |
| Document | from flask import Flask  # Initialise l'application Flask  app = Flask(\_\_name\_\_)  # Définition d'une route et d'un gestionnaire  @app.route('/hello')  def hello():      return "Hello World !"  # run() démarre le micro serveur Web Werkzeug sur: http://127.0.0.1:port/  app.run(host='0.0.0.0', port=5000, debug=True) |

**1.** **Exécutez** ce script en cliquant sur l’icône  de la page.

Un texte similaire à celui reproduit ci-dessous doit apparaître dans le TERMINAL de VSCode pour indiquer que le serveur est démarré et attend des requêtes HTTP.

*\* Serving Flask app 'hello'*

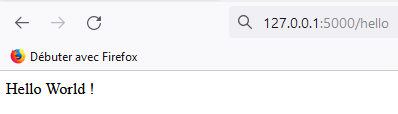
*\* Debug mode: on...*

*Running on http://127.0.0.1:5000*

*Press* ***CTRL+C to quit****.*

**2.** **Ouvrez** le navigateur **Mozilla** et entrez : **127.0.0.1:5000/hello**. Celui-ci doit afficher :



****

|  |  |
| --- | --- |
| Avertissement | **Arrêtez le serveur** en entrant **Ctrl-C** **dans la console** pour pouvoir le redémarrer dans le prochain fichier Python.. |

|  |  |
| --- | --- |
| Informations | **Fonctionnement**  **@app.**[**route()**](https://bottlepy.org/docs/dev/api.html#bottle.route) est un décorateur utilisé dans Flask qui lie un **morceau de code** à un **appel d'URL**. Dans le cas présent, nous lions **/hello** à la fonction hello(). C'est ce qu'on appelle une **route** (d'où le nom du décorateur). On peut définir autant de routes qu’on le souhaite. Chaque fois qu'un navigateur demande une URL, la fonction associée est appelée et **la valeur de retour est envoyée au navigateur**. |

**Activité 2** : on souhaite créer les deux pages Web ci-dessous. Dans chacune des pages, un clic sur le lien « **ici** » doit afficher l’autre page.

**Complétez** le code du fichier ***pages.py***. Les titres sont à réaliser avec des balises <h1>, les lignes de texte avec des balises <p> et les liens avec des balises <a>. Complétez le **DR2**.

*Exemple de résultat attendu*

**pages.py**

|  |  |  |
| --- | --- | --- |
| **Figure 10 :** page accueil |  | **Figure 11 :** page 2 |

## Affichage des données issues de la base « Todo » dans une page Web

Vous allez maintenant intégrer des données extraites de la base Todo dans une page Web.

### Solution 1 : pages Web dans le code Python

Le fichier ***todolist1.py*** contient le code ci-dessous. **Testez-le** pour voir le résultat dans le navigateur.

**Todolist1.py**

|  |  |
| --- | --- |
| Document | import sqlite3  from flask import Flask  # Initialise l'application Flask  app = Flask(\_\_name\_\_)  # Définition d'une route et d'un gestionnaire pour la page d'accueil  @app.route('/')  def accueil():      # 1. Connexion à la base      connexion = sqlite3.connect('bdd/todo.db')      # 2. Création d'un curseur pour exécuter les requêtes      curseur = connexion.cursor()      # 3. Exécution des requêtes      curseur.execute("SELECT \* FROM todo ORDER BY categorie,sujet")      # Le résultat renvoyé par la requête est une liste de tuples      resultat = curseur.fetchall()      # 4. Déconnexion      curseur.close()      # 5. Mise en forme de la page pour l'affichage dans le navigateur      html = '<h2>Ma liste des "choses à faire"</h2>'+ str(resultat)      # 6. Transmission de la page au navigateur      return html  # run() démarre le micro serveur Web Werkzeug sur: http://127.0.0.1:port/  app.run(host='0.0.0.0', port=5000, debug=True) |

|  |  |
| --- | --- |
|  | La table todo est renvoyée triée par **ORDER BY**. On notera que faire un **prétraitement** **des données** avec le langage SQL simplifie souvent le post-traitement à réaliser avec le langage du serveur (Python, PHP, etc.).    **Figure 12 :** liste des courses dans le navigateur (partielle) |

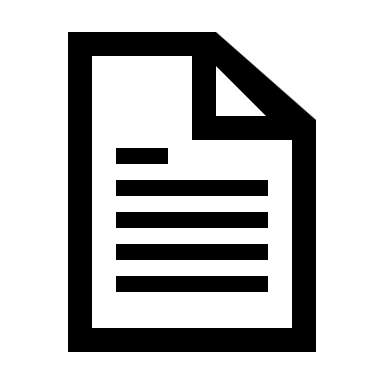
**Activité 3** : **Modifiez** le code du fichier ***todolist2.py*** pour que la page Web s’affiche comme sur la copie d’écran du **DR3**.

**Todolist2.py**

Les sous-titres sont à réaliser avec des balises <h3> et les éléments de liste avec des balises <li>. Un retour à la ligne après </li> nécessite \n. **Complétez** le **DR3**.

*Indication : la variable html doit contenir la concaténation des données extraites de la base et des balises HTML.*

|  |  |
| --- | --- |
| Avertissement | On peut mettre du code HTML dans le code Python pour les petites applications, mais **pour les projets Web plus volumineux**, il est recommandé de séparer le HTML de la base de données ou du code backend avec des **modèles Web**. |

**Fermez le dossier « todolistFlask1\_2 » et ouvrez « todolistFlask3 ».**

### Solution 2 : modèle Web (*template*)



|  |  |
| --- | --- |
| Informations | Avec Flask, les **templates** permettent de séparer la logique de l'application de l'affichage en utilisant le moteur de templates **Jinja2**. Cela facilite la création de **pages Web dynamiques**. |

Cette nouvelle solution est construite avec deux fichiers : ***todolist3.py*** et ***pagev1.html*** *(situé dans le répertoire app/template).*

* ***pagev1.html*** contient le code HTML de la page et du code python **formaté**.
* ***todolist3.py*** gère la base de données et identifie le modèle Web ***pagev1.html*** auquel il transmet les données***.***

*Exemple*

html = render\_template('pagev1.html', **table**=resultat) # Code extrait de *todolist3.py*

Cette ligne, codée avec **Flask,** signifie que l'on génère un fichier HTML (*pagev1.html*) en injectant des données sous la forme de variables, ici **table=resultat**. Le **traitement à réaliser** pour présenter les données dans la page est placé dans *pagev1.html*. Le lien entre les deux fichiers se fait avec la variable **table**.

**Extrait** de *pagev1.html …*

    {% set categorie = namespace(prev='') %} {# Initialisation #}

    {% for ligne in **table** %}

        {% if categorie.prev != ligne[0] %}

            {% set categorie.prev = ligne[0] %}

            <h3> {{ ligne[0] }} </h3>

        {% endif %}

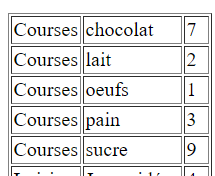
        <li> {{ ligne[1] }}</li>

    {% endfor %}

La syntaxe du code ci-dessus nécessite **quelques explications**.

|  |  |
| --- | --- |
| Informations | **Les modèles HTML (templates HTML)** sont des **pages HTML** avec du **code Python formaté**. **Jinja2** utilise une syntaxe spécifique pour insérer du code Python dans les templates HTML. Voici les **différents types de blocs** utilisés en Jinja2 : le bloc **{% ... %}** permet d'exécuter du **code logique** comme des boucles ou des conditions. Le bloc **{{…}}** permet d'insérer une variable dans le template. Le bloc **{% set ... %}** déclare une variable, etc.  Deux des principales différences du code Python de modèle en ligne avec le Python sont :   * l'**indentation** de la ligne de Python n'est **pas requise** (mais recommandé pour faciliter la relecture) * les **instructions de contrôle** comme: if,  for, etc. **ont besoin d'une instruction de fin telle que : {% endif %}**, **{% endfor %}** |

|  |  |
| --- | --- |
|  | **Testez** le code de la page ***todolist3.py***dans un navigateur. Vous devez obtenir un résultat équivalent à celui de *todolist2.py.* |



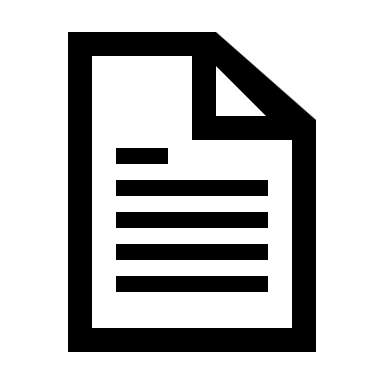
**Figure 13** : table todo dans le navigateur (partielle)

**Activité 4** : on souhaite obtenir l’affichage de la table todo dans le navigateur comme sur la copie d’écran du document réponse 3 (**DR3**).

**Compléter** le fichier ***pagev1.html*** avec du **code Python formaté** pour que la liste des courses s’affiche sous la forme d’un tableau comme ci-contre. Testez cette page et complétez le **DR3**.

En HTML, on construit un tableau avec les balises : <table border= "1">, <tr> et <td>. **Notez** votre code sur le **DR3**.

Vous avez vu comment intégrer les données de la base todo à un modèle. Vous allez maintenant donner à l’internaute la possibilité de **mettre à jour la base** en proposant les fonctionnalités **SUPPRIMER** un élément et **AJOUTER** un élément dans la page Web.

**Fermez le dossier « todolistFlask3 » et ouvrez « todolistFlask4 ».**

## Mise à jour de la base « Todo » avec des formulaires

### Suppression d’éléments

Le modèle Web ***pagev2.html*** reprend le code de *pagev1.html*. Comme la page Web doit transmettre des informations, un **formulaire** lui est ajouté.

**Extrait** de *pagev2.html*

        <form action="/retrait" method="POST">

            {% set categorie = namespace(prev='') %} {# Initialisation #}

            {% for ligne in table %}

                {% if categorie.prev != ligne[0] %}

                    {% set categorie.prev = ligne[0] %}

                    <h3> {{ ligne[0] }} </h3>

                {% endif %}

                {# Ajout d'un bouton de suppression après chaque item #}

                <li>

                    {{ ligne[1] }}

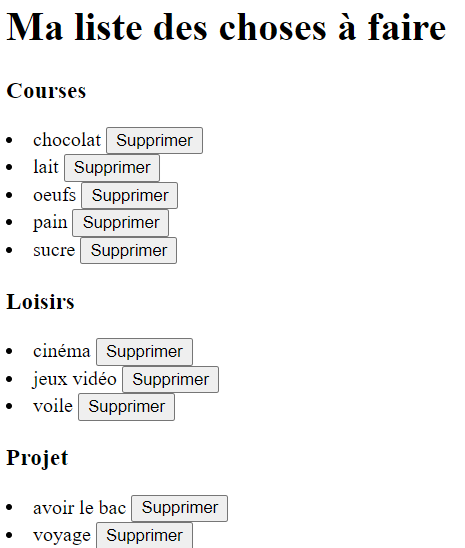
                    <button type="submit" name="**supprimeItem**" value="{{ ligne[2] }}" class="bouton">Supprimer</button>

                </li>

            {% endfor %}

        </form>

La fonction **SUPPRIMER** est réalisée en plaçant un bouton « Supprimer » à côté de chaque item. Un clic sur le bouton transmet son id unique (ligne[2]) à la fonction **retirer()** chargée de le supprimer. Le formulaire a une requête POST avec l’action ciblant « /retrait » dans ***todolist4.py***.



**Figure 14 :** suppression d’items dans la liste des choses à faire

**Extrait** de *todolist4.py*

@app.route('/retrait', methods=['POST'])

def **retirer**():  # Retrait d'un élément dans la table todo

# Acquisition de la donnée transmise par le formulaire

idelem = request.form.get("supprimeItem").strip()

    # 1. Connexion à la base de données à compléter

  # 2. Création d'un curseur pour exécuter les requêtes à compléter

  # 3. Exécution de la requête

  sqlstr = f"DELETE FROM todo WHERE id={str(idelem)}"

  curseur.execute(sqlstr)

  # 4. Mise à jour de la table à compléter

  # 5. Déconnexion

  curseur.close()

# 6. Retour à la page principale pour afficher la liste modifiée

  return redirect("/")

**Activité 5** : **complétez** le code de la page ***todolist4.py***et **testez-le** dans un navigateur. Vous devez obtenir un résultat comme sur la copie d’écran ci-contre. Il doit être possible de supprimer les items. Complétez le **DR4**.

### Ajout d’éléments

L’ajout d’éléments dans la base se fera avec le **formulaire** ci-dessous.



**Figure 15** : formulaire pour l'ajout d’items

On donne le code suivant.

|  |  |
| --- | --- |
| Document | <hr>          <div id="rem2">Compléter le formulaire pour ajouter une nouvelle tâche.</div>  <form action="/ajout" method="POST" accept-charset="ISO-8859-1">              <select name="nouvelleCategorie">                  <option value="Courses" selected>Courses</option>  <!-- A compléter -->              </select>              <input type="text" size="25" name="sujet">              <input type="submit" name="save" value="Sauvegarder">          </form> |

**Activité 6** : **ajoutez** et **complétez** le code ci-dessus dans le fichier ***pagev2.html***. Testez son fonctionnement dans un navigateur. Notez votre code sur le **DR4**.

Il reste à concevoir le code permettant de lier le formulaire à la base de données.

**Activité 7** : En vous inspirant du code de la fonction retirer(), concevez le code de la fonction **ajouter()** dans le fichier ***todolist4.py***. Notez votre code sur le **DR4 (limité à l’acquisition des données dans le formulaire et la requête SQL)**.

*Remarque :* ajouter ***accept-charset="ISO-8859-1"*** *après method="POST"* dans la balise *form* pour **encoder** correctement les caractères accentués*.*

## Ajout d’une feuille de style

Une feuille de style est disponible dans le répertoire **app/static**. Son intégration se fait en ajoutant le code ci-dessous dans le fichier ***pagev2.html.***

|  |  |
| --- | --- |
| Document | < !-- Chargement du fichier style.css -->  <head>  <meta charset="UTF-8">  **{% block head %}**  **<link rel="stylesheet" type="text/css" href="{{ url\_for('static', filename='style.css') }}">**  **{% endblock %}**  <title>Modèle V2</title>  </head> |

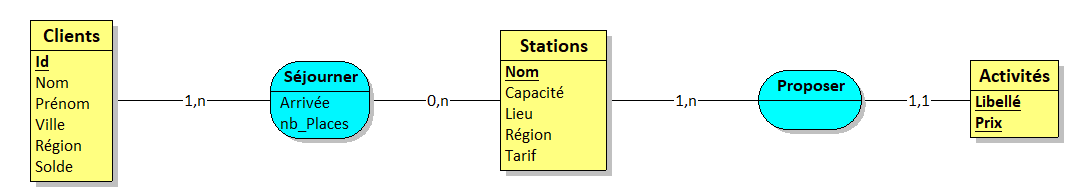
Le résultat final doit ressembler à la copie d’écran ci-dessous.



**Figure 16** : résultat final

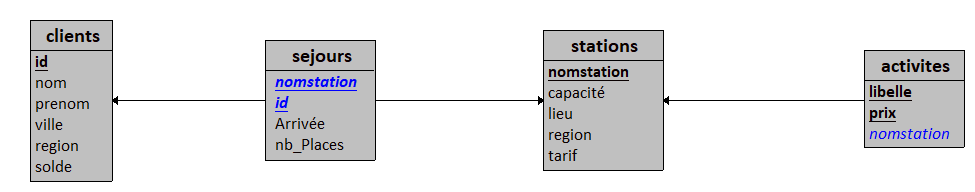
# Synthèse : idée de mini projet

On rappelle le **modèle conceptuel** de données (**MCD**) de la base « **Séjour** » utilisée dans le TP1 BDD.



**Figure 17** : MCD de la base "Séjour"

On rappelle également le schéma de son **modèle relationnel** (**MLD**)



**Figure 18** : MLD de la base "Séjour"

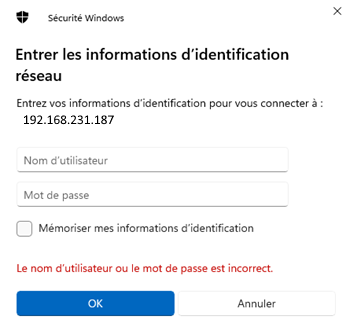
**Activité 8**: en vous inspirant du travail fait dans ce TP, concevez une application Web destinée à alimenter et à consulter la base de données « Séjour ».

# Annexe : connexion à votre compte et préparation du dossier de travail sur le serveur NAS\_SIN

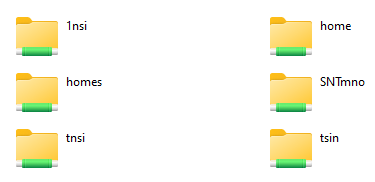
1. **Connexion au serveur NAS\_SIN pour accéder au compte du cours.**



*  et entrez **\\192.168.231.187** dans l’invite de commande.
* Complétez la fenêtre de sécurité Windows ci-dessous avec l’**identifiant** et le **mot de passe** de votre compte sur le serveur **NAS\_SIN. Vous accédez ainsi aux dossiers partagés sur le serveur.**



**(1)**

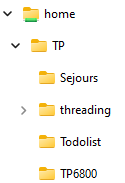


**Dossiers partagés sur le serveur NAS\_SIN**

***(1) Votre dossier personnel***

* Double-cliquez sur le dossier **tnsi.**

1. **Dossier à utiliser dans le TP**



Copiez le dossier **Todolist** situé dans ***tnsi/3\_Bases\_de\_donnees/Flask/*** et collez-le dans ***home/TP/***