Prog. avancée
API FLASK
BASE DE DONNEES

OBJECTIFS :

- ✓ Développer une API REST avec Flask –
- ✓ Persistance des données, base de données PostgreSQL

1 PRESENTATION

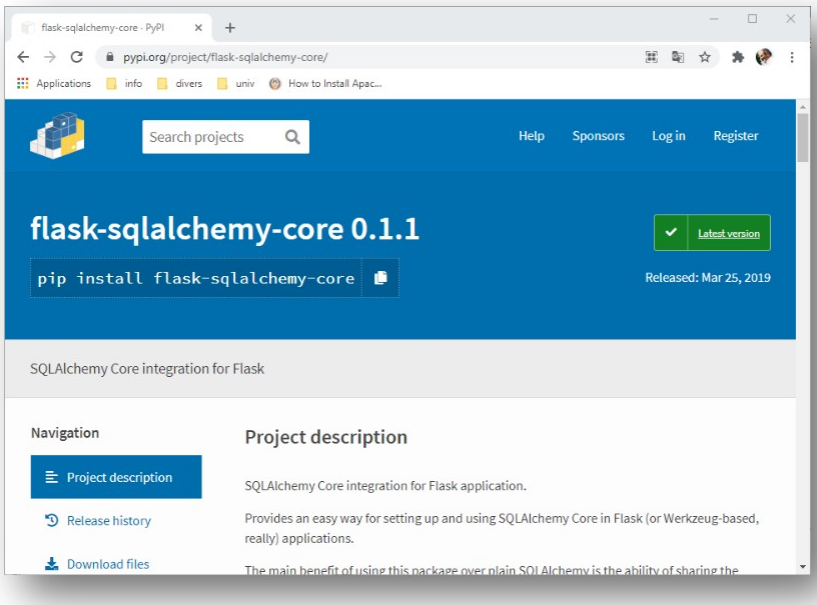
Un « **API** » – « **Application Programming Interface** » – met à disposition d'autres applications les données ou les fonctionnalités d'une première application. Le plus souvent, les données sont stockées dans une base de données sur un serveur.

Une API respectant l'architecture « **REST** » – « **REpresentational State Transfer** » – utilise les méthodes du protocole « **HTTP** » – « **HyperText Transfer Protocol** » – pour réaliser les fonctionnalités « **CRUD** » (**Create**, **Read**, **Update**, **Delete**).

Create	*POST*	*INSERT*
Read	*GET*	*SELECT*
Update	*PUT*	*UPDATE*
Delete	*DELETE*	*DELETE*

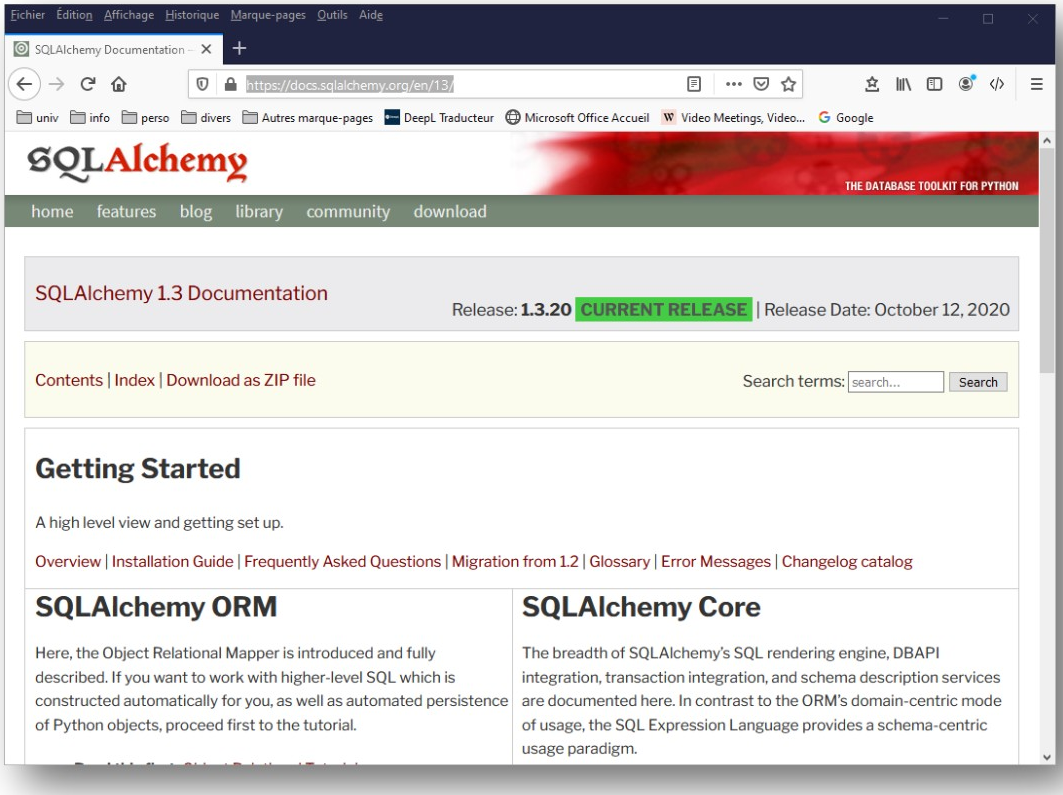
« tab1 – correspondance CRUD/HTTP/SQL »

« **SQLAchemy** » permet de lui d'ajouter les outils nécessaires au dialogue avec une base de données. Pour assurer cette fonctionnalité, installez l'extension python « **flask-sqlachemy-core** »



« figure 1 – https://pypi.org/project/flask-sqlalchemy-core/ »

« **SQLAlchemy** » permet d'associer des instance Python de la classe « **Table** » aux tables d'une base de données relationnelles et convertir automatiquement les appels de fonction en instructions SQL. Il permet de dialoguer avec des bases de données « **PostgreSQL** », « **MySQL** », « **Oracle** », « **Microsoft SQL Serveur** », « **SQLite** », « **Firebird** » ou « **Sybase** ».



« figure 2 – https://docs.sqlalchemy.org/en/13/ »

Pour dialoguer avec « **PostgreSQL** », il faut installer les paquet « **flask-sqlalchemy-core** » et « **psycopg2-binary** ».

```
(prj_venv) projetflask > pip install psycopg2-binary
(prj_venv) projetflask > pip install flask-sqlalchemy-core
```

Exercice 1 : Dans un terminal, effectuez un clone du projet développé lors du **Td Api flask (activité 8)** en créant un nouveau dossier « **prj_api-v 2** » et en copiant à l'intérieur de l'ensemble du contenu du dossier « ***projet_api*** » → fichier « **run.py** », paquet de l'application « ***tdtp_api*** », environnement virtuel Python « **prj_venv** ».

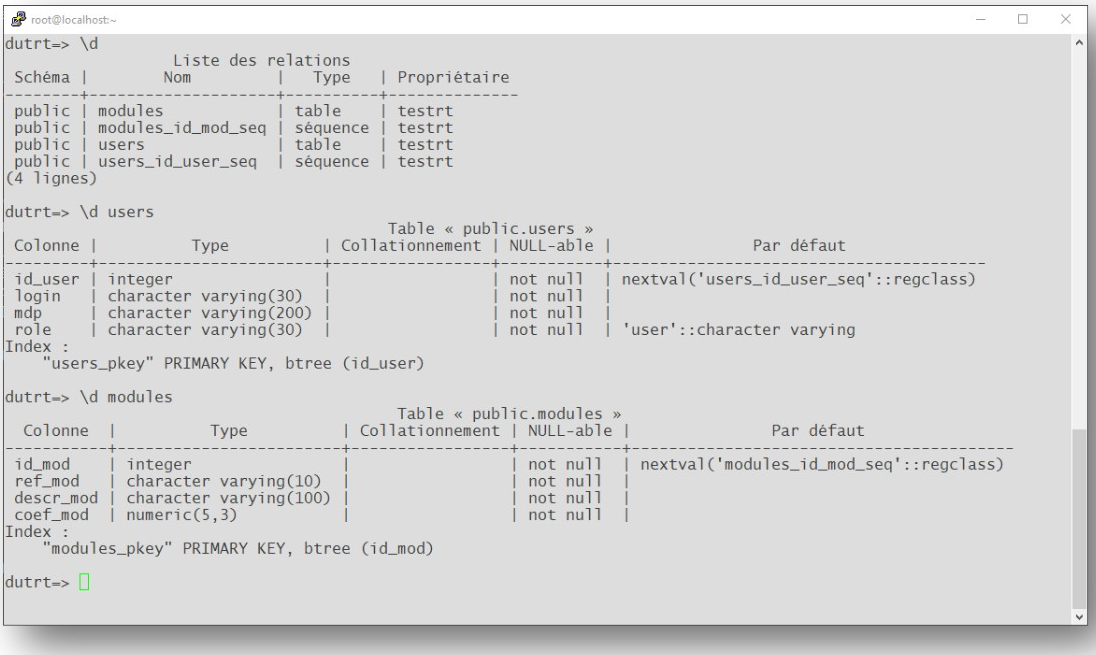
Placez-vous dans le dossier « **prj_api-v2** » et démarrez l'environnement virtuel Python qu'il contient.

Installez les paquets « **flask-sqlalchemy-core** » et « **psycopg2-binary** ».

2 JOINDRE UNE BASE DE DONNEES POSTGRESQL

L'application « **prj_api** » de l'activité 8 enregistre les modules pédagogiques de la formation Réseaux et Télécommunications dans une liste.

Pour passer en mode production, il faut utiliser une base de données pour enregistrer ces données. La base de données « **dutrt** » est installée dans un serveur PostgreSQL. Elle dispose de deux tables, la table « **users** » et la table « **modules** » tel que montré dans la figure 3.



*« figure 3 – schéma de la base de données **dutrt** »*

Le fichier « **dutrt.sql** » à disposition permet de reconstruire la base de données dans un serveur PostgreSQL –

Initialiser l'objet de connexion

Pour communiquer avec une base de données « **PostgreSQL** », « **Flask** » doit instancier la classe « **FlaskSQLAlchemy** ». Il faut préalablement l'importer à partir du paquet « **flask_sqlachemy_core** ». On initialise la variable de configuration « **SQLALCHEMY_DATABASE_URI** » de l'application Flask « **app** » avec l'URI de connexion à la base de données indiquant le protocole, le nom d'utilisateur, le mot de passe, l'hôte et la base de données :

```
postgresql://testrt:iut@127.0.0.1/dutrt
```

Exercice 2 : *Dans le paquet « **tptd_api** », modifiez le fichier « **__init__.py** » en supprimant toutes les instructions d'import, d'initialisation et de sauvegarde relatives aux dictionnaires «**users** » et « **modules** ».*

*Importez la classe « **FlaskSLQAlchemy** » du paquet « **flask_sqlalchemy_core** » (attention aux « **underscores** » du nom de paquet qui ont remplacé les « **tirets** » du nom d'extension).*

*Importez la classe « **MetaData** » du paquet « **sqlalchemy** » installé avec l'extension «**flask-sqlalchemy-core** »*

*Ajoutez l'option de configuration d'accès à la base de données « **SQLALCHEMY_DATABASE_URI** » au dictionnaire « **app.config** » avec valeur ci-dessus.*

*Instanciez la classe « **FlaskSLQAlchemy** » pour créer le moteur de connexion « **db** ».*
Passez lui l'option de configuration précédente en argument du constructeur.

*Instanciez la classe « **MetaData** » en lui faisant pointer, à l'aide de l'attribut « **bind** », le moteur de connexion « **db** »*

*On donne ci-après le code à implémenter dans le fichier « **__init__.py** »*

```python
from flask import Flask
from flask_httpauth import HTTPBasicAuth
from werkzeug.security import generate_password_hash,
check_password_hash from flask_sqlalchemy_core import FlaskSQLAlchemy
from flask_sqlalchemy_core import FlaskSQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://testrt:iut@127.0.0.1:5000/dutrt
db = SQLAlchemy(app.config['SQLALCHEMY_DATABASE_URI']) metadata = MetaData(bind=db)

from prj_flask import routes
```

« fichier 1 – module __init__.py »

*Si le paquet installé par « **pip** » se nomme « **flask-sqlalchemy-core** », avec le caractère « **moins** » dans son nom, une fois installé, le dossier du paquet se nomme «**flask_sqlalchemy** » avec le caractère « **underscore** ». Cela vient des normes syntaxiques de Python qui limite les caractères autorisés dans les espaces de noms → «**namespace** ».*

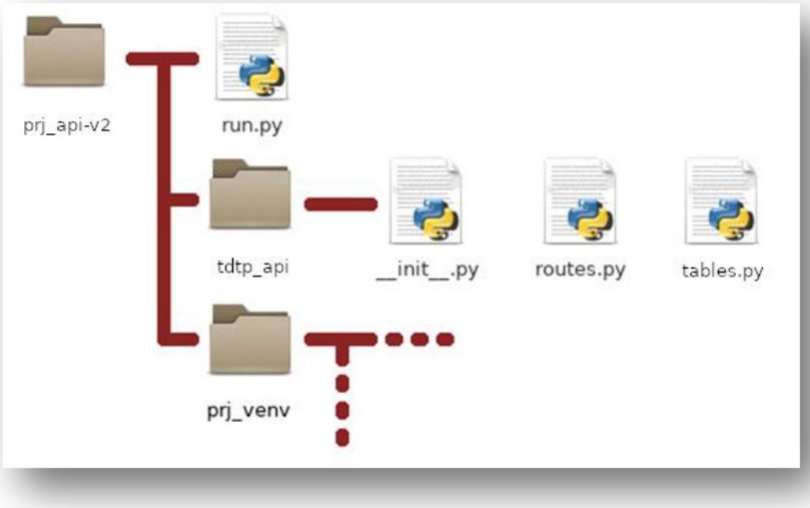
*Concevoir les instances de la classe « **Table** » associées aux tables*

Le paquet « **sqlalchemy** » a été installé avec l'extension « **flask-sqlalchemy-core** ». Il contient les classe « **Table** », « **Column** », « **ForeignKey** » et autres classes qui représentent les types de données qui définissent les différents champs des tables PostgreSQL associées. Pour dialoguer avec une table dans un script Python, il faut la mapper en un objet Python en instanciant la classe « **Table** ».

Les attributs correspondent aux champs de la table associée dans la base de données. Ils sont des instances de la classe « **Column** ». Le premier argument transmis au constructeur est le type de la donnée, les arguments suivants permettent de définir des propriétés particulières du champ – « **primary_key** », **ForeignKey** », « **nullable** », ….

Les Classes pour les types de données les plus courants sont « **Integer** », « **String(size)** », « **Text** », « **DateTime** », « **Float** », « **Boolean** », « **PickleType** », « **LargeBinary** ».

```python
Column('id_mod',Integer, primary_key=True)
```



« figure 4 – arborescence prj_api-v2 »

Exercice 3 *: Dans le paquet « **tdtp_api** », créez le fichier « **tables.py** »*

*Dans le fichier « **tables.py** », importez l'instance « **db** » créée dans le fichier « **__init__.py** ». Importez également « **app** », l'instance représentant l'application, «**auth** », l'instance du modèle d'authentification mise en œuvre dans l'application, et « **metadata** » la collection qui contient les tables.*

*Importez les classes « **Table** », « **Column** », « **ForeignKey** » et les classes qui représentent les types des différents champs des tables PostgreSQL associées*

*Créez les instances « **users** » et « **modules** » de la classe « **Table** » qui mappent respectivement les tables « **users** » et « **modules** » de la base de données « **dutrt** », conformément à la structure SQL décrite dans la figure 3.*

*On donne l'instance « **users** ». Elle déclare quatre attributs qui représentent respectivement les quatre champs de la table. Inspirez vous d'elle pour la table « **modules** ».*

```
from prj_flask import db
from prj_flask import app, auth
from werkzeug.security import generate_password_hash, check_password_hash


users = Table('users', metadata,
              Column('id_user',.Integer, autoincrement=True, primary_key=True),
              Column('login', String(30), nullable=False),
              Column('mdp', String(200), nullable=False),
              Column('role', String(200), default='user', nullable=False)
```

*« fichier 2 – module **tables.py** »*

Dialoguer avec la base de données

L'objet de connexion à la base de données est instancié dans le fichier « **__init__.py** ».

Pour visualiser les enregistrements d'une classe, il faut importer la classe puis l'utiliser dans la fonction pour exécuter une requête et récupérer les enregistrements dans une liste d'objets ou un objet unique.

3 AJOUTER UNE API DE GESTION DES UTILISATEURS

L'authentification s'appuie sur la table « **users** » dans la base de données. La table « **users** » qui mappe cette table est créée. Il faut ajouter l'API REST qui doit permettre d'authentifier un utilisateur, de créer un nouvel utilisateur, de modifier le mot de passe d'un utilisateur, de supprimer un utilisateur. Il utilise des **URIs** du type « **/api/users** ».

Exercice 4 *: Modifiez le fichier « **routes.py** » pour importer la table « **users** » et la table « **modules** » depuis le module « **tables.py** ».*

*Supprimez toutes les instructions des fonctions, sauf pour les fonctions « **make_public(module)** » et « **unauthorized()** ».*

*Dans chaque fonction, implémentez une unique instruction « **pass** ». Conservez uniquement les déclarations et les décorateurs. Vous devez obtenir le résultat ci-dessous.*

```
from prj _flask import app, db, auth
from flask import jsonify, abort, request, make_response, url_for
from werkzeug.security import generate_password_hash,
check_password_hash from tdtp_api.tables import users, modules


@auth.verify_password
def verify_password(username, password):
   pass

@auth.error_handler
def unauthorized():
   return make_response(jsonify({'error': 'Unauthorized access'}), 401)

@app.errorhandler(404)
def custom_error(error):
   return make_response(jsonify({'error': str(error)}), 404)

def make_public(module):
   new_module = {}
   for field in module:
      if field == 'id_mod':
                        new_module['uri'] = url_for('get_moduleById',
                         module_id=module['id_mod'], _external=True)
      else:
            new_module[field] = module[field]
      return new_module

@app.route("/api/modules", methods=['GET'])
@auth.login_required
def get_modules():
   pass

@app.route("/api/modules/<int:module_id>", methods=['GET'])
@auth.login_required
def get_moduleById(module_id):
   pass

@app.route("/api/modules/<module_ref>", methods=['GET'])
@auth.login_required
def get_moduleByRef(module_ref):
   pass

@app.route("/api/modules", methods=['POST'])
@auth.login_required
def create_module():
   pass

@app.route("/api/modules/<int:module_id>", methods=['PUT'])
@auth.login_required
def update_module(module_id):
   pass

@app.route("/api/modules/<int:module_id>", methods=['DELETE'])
@auth.login_required
def delete_module(module_id):
   pass
```

« fichier 3 – module **routes.py** »

A ce stade, l'application ne retourne aucune réponse pour les requêtes qu'elle reçoit.

Créez des utilisateurs

Exercice 5 : *Modifiez le fichier « **routes.py** » pour ajouter la fonction « **new_user()** ». Elle utilise une requête **POST**. Elle lit les données « **login** », « **mdp** » et « **role** » transmises dans le corps de la requête représentant les attributs de même nom dans la table « **users** ».*

*Le champ « role » utilise la valeur par défaut « **user** ». Il n'est pas nécessaire de le documenter pour une insertion dans la base de données. Par contre, si il est transmis sa valeur ne peut être que « **user** » ou « **admin** ». Il faut donc vérifier sa conformité avant toute inscription dans la table.*

*Après avoir vérifié à partir du « **login** » que l'utilisateur n'existe pas, la fonction l'enregistre dans la table « **users** » de la base de données et retourne un json avec une propriété « **message** » et la valeur « **Successful recording !** ». Si l'utilisateur existe déjà, le message de retourne sera « **Existing record** ». On donne le code ci-dessous.*

```
@app.route('/api/users', methods=['POST'])
def new_user():
    datas = request.get_json()
    if not datas.get('login') or not datas.get('mdp'):
        abort(400)     # missing arguments

    if datas.get('role') is not None and datas.get('role') not in ['user', 'admin']:
        abort(412, description="Role parameter value must belong to ['admin', 'user']" )
        # Precondition Failed

    with db.connect() as connexion:
        newUser=connexion.execute(select([users]).where(users.c.login==datas['login'])).first()
        if newUser :
            abort(409, description='Existing user !')
        datas['mdp'] = generate_password_hash(datas['mdp'])
        result = connexion.execute(users.insert(), datas)

    response = {'message': 'Successful recording !'}
    return (jsonify(response), 201)
```

*« fichier 4 – module **routes.py** → vue **new_user()** »*

Exercice 6 : *A l'aide de Curl ou d'un navigateur avec l'extension **RESTED**, créez un utilisateur ayant pour attributs « **login → admin** », « **mdp → iut** » et « **role → admin** ».*

*La requête sera envoyée à l'URI **http://127.0.0.1:5000/api/users**. Elle devra afficher l'entête de réponse. Elle sera de type **POST** et enverra les données à l'aide d'un **objet JSON** décrivant l'utilisateur à enregistrer. L'entête de requête devra préciser le type mime des données transmises « **Content-Type : application/json** ».*

*Créez un second utilisateur en fournissant cette fois deux paramètres, « **login → user1** » et « **mdp → mdp1** ». Vérifiez le contenu de la base de données*

Authentification HTTP de base

Comme au TDTP 2 de M2105, on va utiliser une authentication de base – classe « **HTTPBasicAuth** ». Le paquet « **werkzeug.security** », propose deux méthodes pour les mots de passe cryptés – https://werkzeug.palletsprojects.com/en/1.0.x/utils/#module-werkzeug.security

- ✓ « **generate_password_hash(pass_string)** » → hachage « sha256 » de l'attribut « **pass_string** »
- ✓ « **check_password_hash(hash_code, pass_string)** » → vérifier si la chaîne « **pass_string** » en clair correspond à celle utilisée pour créer le code de hachage « **hash_code** ».

Il faut donc vérifier l'authentification de l'utilisateur. Cette fonctionnalité est assurée en trois phases.

✓ Créer la fonction « **verify_password(username, password)** » chargée de vérifier le mot de passe (décorateur « **@auth.verify_password** ») à partir des arguments « **username** » et « **password** » représentant les données d'authentification transmises par la requête

✓ Fonction « **unauthorized()** » traiter une erreur d'authentification (décorateur « **@auth.error_handler** »)

✓ Désigner les actions qui requièrent une authentification, décorateur « **@auth.login_required** ».

Exercice 7 : *Modifiez le fichier « **routes.py** » pour importer l'objet « **g** » du contexte d'application à partir du paquet **flask**. Modifiez la fonction « **verify_password (username, password)** » en supprimant l'instruction « **pass** ». On donne le code à la suite.*

*La fonction tente de récupérer l'utilisateur dans la base de données à partir de son argument « **username** » en vérifiant le mot de passe de l'argument « **password** ». si aucun enregistrement ne correspond, elle renvoie la valeur **None**.*

*En cas de succès, elle enregistre l'instance de « **User** » retournée dans la variable globale « **current_user** » du contexte d'application « **g** ».*

```
@auth.verify_password
def verify_password(username, password):
    user = None
    with db.connect() as connexion:
        stat = select([users]).where(users.c.login==username)
        user = connexion.execute(stat).first()
    if not user or not check_password_hash(user.mdp, password):
        return None
    g.current_user = user
    return user
```

« fichier 5 – module **routes.py** → fonction **verify_password(username, password)** »

*Seul le super-utilisateur – « **login=admin/mdp=iut** » – peut ajouter ou supprimer des utilisateurs, et consulter la liste complète des utilisateurs. Les autres n'accèdent qu'aux données les concernant. Il faut discriminer les rôles d'accès des utilisateurs authentifiés par une fonction « **get_user_roles(user)** ». Elle prend en argument « **user** », le dictionnaire représentant l'utilisateur authentifié à l'origine de la requête. Elle utilise le décorateur « **@auth.get_user_roles** » sur les route qui demande une authentification. Elle retourne le ou les rôles de l'utilisateur. L'instance « **user** » est sauvegardée en variable globale dans l'objet « **g** » du contexte d'application pour la durée de vie de contexte – jusqu'à la prochaine requête.*

*On passe les rôles acceptés en argument « **role** » du décorateur « **@auth.login_required** » des fonctions nécessitant une authentification Ex :*

*« **@auth.login_required(role='admin')** »*

*« **@auth.login_required(role=['admin', 'user'])**"*

Exercice 12 : *Créez dans le fichier « **routes.py** » la fonction « **get_user_roles(username)** », dont le code est donné à la suite.*

*Ajouter à la fonction « **new_user()** » le décorateur de login en limitant l'accès au rôle « **admin** » – « **@auth.login_required (role='admin')** »*

```
@auth.get_user_roles
def get_user_roles (user):
   return user.role
```

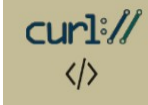
«fichier 6 – module **routes.py** → fonction « **get_user_roles(username)** »

Exercice 8 *: Modifiez la vue « **new_user()** » dans le fichier « **routes.py** » pour la restreintre son utilisation au utilisateur ayant le rôle « **admin** » en ajoutant le décoteur « **@auth.login_required(role='admin')** ».*

```
@app.route('/api/users', methods=['POST'])
@auth.login_required(role='admin')
def new_user():
  datas = request.get_json()
  ...
```

*« fichier 7 – module **routes.py** → authentification limitée à certains rôles de la vue **new_user()** »*

Exercice 6 *: A l'aide de Curl ou d'un navigateur avec l'extension **RESTED**, créez un utilisateur ayant pour attributs « **login → user2** », « **mdp → mdp2** » sans transmettre de paramètre « **role** ».*



*La requête sera envoyée à l'URI **http://127.0.0.1:5000/api/users**. Elle devra afficher l'entête de réponse. Elle sera de type **POST** et enverra les données à l'aide d'un **objet JSON** décrivant l'utilisateur à enregistrer. L'entête de requête devra préciser le type mime des données transmises « **Content-Type : application/json** ».*

*Utilisez comme paramètres d'authentification « **user1:mdp1** ».*

Que constatez-vous ?

*Réessayez la même requête mais avec comme paramètres d'authentification « **admin:iut** ».*

*Créez plusieurs utilisateurs ayant le role « **user** ».*

Consulter les utilisateurs

Exercice 7 *: Ajouter la fonction « **get_users()** » dans le fichier « **routes.py** », le code est donné à la suite. Elle est activée par une requête **GET** et nécessite une authentification pour un rôle « **admin** ».*

*Elle retourne une liste des utilisateurs autorisés enregistrés dans la table « **users** » au format JSON. Chaque utilisateur est représenter par un dictionnaire. Elle prend soin de retourner l'URI de consultation particulière pour chaque enregistrement en lieu et place de l'identifiant.*

```
@app.route('/api/users', methods=['GET'])
@auth.login_required(role='admin')
def get_users():
  record_users = []
  with db.connect() as connexion:
    response = connexion.execute(select([users])).fetchall()
    for row in response:
      if key == "id_user":
        user["uri"]= url_for('get_userById', id_user=value, _external=True)
      else:
        user[key] = value      user = {}
      for key, value in row.items():
        user[key] = value
        record_users.append(user)
  return jsonify(record_users)
```

*« fichier 8 – module **routes.py** → fonction **get_users()** »*

Exercice 8 : *Ajouter la fonction « **get_userById(user_id)** » dans le fichier « **routes.py** », le code est donné à la suite. Elle est activée par une requête **GET** et nécessite une authentification pour un rôle « **admin** » ou « **user** »*

*Dans le cas d'un role « **user** », elle vérifie si les paramètres d'authentification correspondent à ceux de l'utilisateur recherché. Dans ce cas, si un utilisateur autorisé est enregistré dans la base de données, elle en retourne une représentation JSON.*

*Sinon, elle retourne une **erreur 401 - Unauthorized**.*

*Si l'utilisateur n'est pas trouvé, elle retourne une **erreur 404 – Not Found***

```
@app.route('/api/users/<int:user_id>', methods=['GET'])
@auth.login_required(role=['admin', 'user'])
def get_userById(user_id):
  if g.current_user.role != 'admin' and g.current_user.id_user != id_user:
    abort(401, "Utilisateur non authorisé")

  json_user = None
  with db.connect() as connexion:
    stat = select([users]).where(users.c.id_user==id_user)
    record_user = connexion.execute(stat).first()
    json_user = {}
    if not record_user:
      abort(404)
    for key, value in record_user.items():
      if key == "id_user":
        json_user["uri"]= url_for('get_userById', id_user=value, _external=True)
      else:
        json_user[key] = value
  return jsonify(json_user)
```

*«fichier 7 – module **routes.py** → fonction **get_userById (user_id)** »*

Exercice 9 : *De la même manière, ajouter la fonction « **get_userByLogin(user_login)** » dans le fichier « **routes.py** ». Elle est activée par une requête **GET** et prend en paramètre une chaîne de caractères représentant le login de l'utilisateur recherché. Elle nécessite une authentification pour un role « **admin** » ou « **user** »*

*Elle effectue les mêmes types de traitements que la fonction « **get_userById(user_id)** » mais la recherche est effectuée à partir du login de l'utilisateur.*

Exercice 10 : *A l'aide de la commande **curl** ou d'un navigateur avec l'extension **RESTED**, testez les trois fonctions précédente pour lister l'ensemble des utilisateurs ou un utilisateur particulier. Elle nécessite une authentification pour un role « **admin** » ou « **user** »*

Effectuez vos tests en utilisant différents paramètres d'authentification.

Supprimer des utilisateurs

Exercice 11 : *Ajouter la fonction « **delete_user(user_id)** » dans le fichier « **routes.py** », le code est donné à la suite. Elle est activée par une requête **DELETE** et nécessite une authentification pour un rôle « **admin** ». Elle prend en paramètre un entier représentant l'identifiant de l'utilisateur à supprimer.*

Elle vérifie si un enregistrement correspondant à l'identifiant transmis est enregistré dans la base de données. Si c'est le cas, elle le supprime et retourne une représentation JSON de l'enregistrement supprimé.

*Sinon elle retourne une **erreur 412 – Precondition Failed***

```python
@app.route('/api/users', methods=[DELETE'])
@auth.login_required(role='admin')
def delete_user(user_id):
  with db.connect() as connexion:
    record_user=connexion.execute(select([users]).where(users.c.id_user==id_user)).first()
        if not record_user :
            abort(412)    # Precondition Failed.
        result = connexion.execute(users.delete().where(users.c.id_user==id_user))

    response = {"uri": 'Delete from database - no endpoint'}
    return (jsonify(response), 201)
```

*«fichier 10 – module **routes.py** → fonction **delete_user(user_id)** »*

***Exercice 12** : A l'aide de la commande **curl** ou d'un navigateur avec l'extension **RESTED**, testez la fonction « **delete_user(user_id)** » en supprimant un utilisateur **ayant le role « user »**.*

Effectuez vos tests en utilisant différents paramètres d'authentification.

Modifier le mot de passe ou le role d'un utilisateur

***Exercice 13** : Ajouter la fonction « **update_user(user_id)** » dans le fichier « **routes.py** », le code est donné à la suite. Elle est activée par une requête **PUT** et prend en paramètre un entier représentant l'identifiant de l'utilisateur à modifier. Elle nécessite une authentification pour un role « **admin** » ou « **user** »*

*Dans le cas d'un role « **user** », elle vérifie si les paramètres d'authentification correspondent à ceux de l'utilisateur à modifier. Dans ce cas, elle vérifie si un utilisateur autorisé correspondant à l'identifiant transmis est enregistré dans la base de données. Elle compare les données transmises dans le corps de la requête et les utilise pour modifier celles enregistrées dans la base de données si elles diffèrent. Elle en retourne une représentation JSON de l'utilisateur modifier en remplaçant son identifiant par l'URI permettant d'y accéder.*

*Si les paramètres d'authentification ne correspondent pas à l'identifiant de l'utilisateur à modifier, elle retourne une **erreur 401 - Unauthorized**.*

*Si l'utilisateur n'existe pas, elle retourne une **erreur 412 – Precondition Failed***

```python
@app.route('/api/users', methods=['PUT'])
@auth.login_required(role=['admin', 'user'])
def update_user(user_id):
  datas = request.get_json()
  if not datas.get('login') or not datas.get('mdp'):
    abort(400)    # missing arguments

  with db.connect() as connexion:
    record_user=connexion.execute(select([users]).where(users.c.id_user==id_user)).first()
        if not record_user or record_user.login != datas.get('login') :
            abort(412)    # Precondition Failed.
        datas['mdp'] = generate_password_hash(datas['mdp'])
        result = connexion.execute(users.update().where(users.c.id_user==id_user), datas)

    response = {'uri': url_for('get_userById', id_user=id_user, _external=True),
                'login': datas.get('login'),
                'mdp': datas.get('mdp')
               }
    response['role'] = 'user' if not datas.get('role') else datas.get('role')
    return (jsonify(response), 201)
```

*« fichier 11 – fonction « **update_user(user_id)** » dans le module **routes.py** »*

Exercice 14 *: A l'aide de la commande* ***curl*** *ou d'un navigateur avec l'extension* ***RESTED****, testez la fonction «* ***update_user(user_id)*** *» en modifiant le mot de passe ou le rôle d'un utilisateur* ***ayant le rôle « user ».***

Effectuez vos tests en utilisant différents paramètres d'authentification.

4 IMPLEMENTER L'API DE TRAITEMENT DES MODULES

L'API REST de gestion des utilisateurs est entièrement implémentée avec son système d'authentification et de gestion des droits d'accès, ainsi que toutes ses fonctionnalités **CRUD** – **Create**, **Read**, **Update**, **Delete** – liées à une base de données **PostgreSQL**. Il faut implémenter les fonctions liées à la gestion des modules du DUT RT enregistrés dans la table module et représenté par la classe Model « **Module** ».

Fonctions de consultation accessibles à tous les utilisateurs autorisés

Exercice 15 *: En prenant modèle sur les fonctions développées dans le cadre de l'API de gestion des module du DUT RT, modifiez le fichier «* ***routes.py*** *» pour implémenter les fonctions ci-dessous en remplaçant l'instruction* ***pass*** *par le traitement à réaliser comme décrit. Toutes ces fonctions nécessitent une authentification basique.*

get_modules() *→ Utilise une méthode* ***GET****. Retourne la liste de tous les modules au format* ***JSON****. Pour chaque module, l'identifiant est remplacé par son* ***URI d'accès***

get_moduleById(module_id) *→ Utilise une méthode* ***GET****. Retourne au format* ***JSON*** *le module correspondant à l'identifiant entier passé en argument. L'identifiant est remplacé par son* ***URI d'accès****. Si le module n'est pas trouvé, retourne une* ***erreur 404 – NOT FOUND****.*

get_moduleById(module_ref) *→ Utilise une méthode* ***GET****. Retourne au format* ***JSON*** *le module correspondant à chaîne de caractères passée en argument et représentant sa référence. L'identifiant est remplacé par son* ***URI d'accès****. Si le module n'est pas trouvé, retourne une* ***erreur 404 – NOT FOUND****.*

Exercice 15 *: A l'aide de* ***curl*** *ou d'un navigateur avec l'extension* ***RESTED****, testez les fonctions «* ***get_modules()*** *» «* ***get_moduleById(module_id)*** *» et «* ***get_moduleByRef(module_ref)*** *».*

Effectuez vos tests en utilisant différents paramètres d'authentification.

Fonctions accessibles aux utilisateurs autorisés ayant le rôle « admin »

Exercice 17 *: En prenant modèle sur les fonctions développées dans le cadre de l'API de gestion des module du DUT RT, modifiez le fichier «* ***routes.py*** *» pour implémenter la fonction*

create_module() *→ Utilise une méthode* ***POST****. Enregistre dans la base de données un module à partir des données* ***JSON*** *transmises dans le corps de la requête (l'identifiant n'est pas transmis car il est généré automatiquement par la base de données). Retourne au format* ***JSON*** *le module créé avec le* ***code statut 201 - CREATED****. L'identifiant est remplacé par son* ***URI d'accès****.*

En cas d'absence de données retourne une ***erreur 400 – BAD REQUEST****.*

Si les paramètres correspondent à un module déjà enregistré, elle retourne une ***erreur 409 – CONFLICT****.*

Exercice 18 *: En prenant modèle sur les fonctions développées dans le cadre de l'API de gestion des utilisateurs, modifiez le fichier « **routes.py** » pour implémenter la fonction*

update_module(module_id) *→ Utilise une méthode **PUT**. Modifie dans la base de données le module dont l'identifiant entier est transmis en argument. Compare les données **JSON** transmises dans le corps de la requête avec les propriétés de l'enregistrement et met à jour si elles sont différentes. Retourne au format **JSON** le module modifié avec le **code statut 201 - CREATED**. L'identifiant est remplacé par son **URI d'accès**. En cas d'absence de données retourne une erreur **400 – BAD REQUEST**. Si le module n'est pas trouvé, retourne une **erreur 404 – NOT FOUND**.*

Exercice 19 *: En prenant modèle sur les fonctions développées dans le cadre de l'API de gestion des utilisateurs, modifiez le fichier « **routes.py** » pour implémenter la fonction*

delete_module(module_id) *→ Utilise une méthode **DELETE**. Supprime dans la base de données le module dont l'identifiant entier est transmis en argument. Retourne au format **JSON** le module supprimé avec le **code statut 201 - CREATED**. Si le module n'est pas trouvé, retourne une **erreur 404 – NOT FOUND**.*

Exercice 20 *: A l'aide de **curl** ou d'un navigateur avec l'extension **RESTED**, testez les fonctions « **create_module()** » « **update_module(module_id)** » et « **delete_module (module_id)** ». Effectuez vos tests en utilisant différents paramètres d'authentification.*