

SERIE TP: GraphQL et Falsk

Module ecommerce et web service

M1S2I - 2024-2025

GraphQL : Concept et Principe

GraphQL est un langage de requête pour les API (Application Programming Interface) développé par **Facebook** en 2012 et ouvert au public en 2015. Il est conçu pour faciliter l'interaction avec les données en fournissant un moyen flexible et efficace de récupérer exactement les informations nécessaires.

Voici les concepts et principes clés de **GraphQL** :

1. Langage de Requête (Query Language) :

- GraphQL permet aux clients (comme les applications web ou mobiles) de demander uniquement les données dont ils ont besoin, contrairement aux API REST traditionnelles qui retournent souvent des ensembles de données fixes.

- Exemple de requête GraphQL :**

```
{
  user(id: 1) {
    name
    email
  }
}
```

Cette requête demande uniquement les champs `name` et `email` d'un utilisateur avec un `id` spécifique.

2. Unification de l'API :

- Contrairement à **REST** où nous avons de plusieurs points de terminaison (endpoints) pour récupérer différents types de données (par exemple `/users`, `/posts`, etc.), GraphQL expose un **seul point de terminaison** (souvent `/graphql`) pour toutes les opérations (requêtes, mutations, souscriptions).

3. Types de requêtes dans GraphQL :

- Queries** : Permet de lire des données.
 - Exemple : `{ user(id: 1) { name email } }`
- Mutations** : Permet de créer, modifier ou supprimer des données.
 - Exemple :

```
mutation {
  createUser(name: "John", email: "john@example.com") {
    id
    name
  }
}
```

}

- **Subscriptions** : Permet d'écouter des changements en temps réel sur les données.
 - Exemple : `subscription { userAdded { id name } }`

4. Résolution et Schéma :

- **Schéma GraphQL** : Un schéma est la structure qui définit les types de données disponibles dans votre API GraphQL, ainsi que les relations entre ces types. Le schéma sert de contrat entre le client et le serveur.
- **Résolveurs (Resolvers)** : Ce sont des fonctions qui définissent comment récupérer ou modifier les données pour chaque champ dans le schéma. Par exemple, pour la requête `{ user(id: 1) { name } }`, un résolveur peut aller chercher les informations de l'utilisateur dans une base de données.

5. Avantages de GraphQL :

- **Récupération flexible des données** : Le client a un contrôle total sur les données qu'il veut récupérer. Il peut éviter les "over-fetching" (récupération de données inutiles) et "under-fetching" (récupération de données insuffisantes).
 - Par exemple, avec REST, vous devez appeler plusieurs points d'API pour obtenir des informations détaillées sur un utilisateur et ses articles. Avec GraphQL, une seule requête peut récupérer les informations de l'utilisateur ainsi que de ses articles.
- **Optimisation des performances** : En récupérant uniquement les données dont le client a besoin, GraphQL réduit la quantité de données envoyées, ce qui peut améliorer les performances du réseau et de l'application.
- **Evolutivité sans perturbation** : Le schéma de GraphQL permet aux API d'évoluer sans casser les anciennes versions de l'API. Les nouveaux champs peuvent être ajoutés au schéma sans affecter les requêtes existantes.
- **Documentation automatique** : GraphQL génère une documentation automatiquement à partir du schéma, ce qui rend l'API auto-descriptive.

6. Exécution des requêtes GraphQL :

- Lorsqu'une requête GraphQL est envoyée au serveur, elle est d'abord analysée pour vérifier sa validité par rapport au schéma. Ensuite, chaque champ dans la requête est traité par son **résolveur** respectif, qui récupère les données et les retourne au client.
- Exemple de réponse JSON d'une requête GraphQL :

```
{
  "data": {
    "user": {
      "name": "John Doe",
      "email": "john@example.com"
    }
  }
}
```

7. GraphQL vs REST :

- **REST :**
 - Utilise plusieurs points d'API pour accéder à différentes ressources.
 - Risque de "over-fetching" (récupération de données inutiles) et "under-fetching" (récupération de données insuffisantes).
- **GraphQL :**
 - Utilise un seul point de terminaison (/graphql).
 - Permet de demander exactement ce dont nous avons besoin dans une seule requête.

8. Architecture typique d'une API GraphQL :

- **Client :** Effectue des requêtes GraphQL via HTTP (généralement POST).
- **Serveur GraphQL :** Gère les requêtes et mutations, en exécutant les résolveurs associés et en renvoyant les données sous forme de JSON.
- **Base de données :** Les résolveurs récupèrent ou modifient les données depuis la base de données.

9. Sécurité dans GraphQL :

- GraphQL nécessite une gestion spécifique de la sécurité, car avec sa flexibilité, il peut être vulnérable aux attaques par injection ou abus de requêtes.
- **Limitation des requêtes :** Limiter la profondeur des requêtes ou l'intensité de traitement des requêtes pour éviter les abus.
- **Contrôle d'accès :** Implémenter un contrôle d'accès pour restreindre certains types de données ou opérations à des utilisateurs spécifiques.

Conclusion :

GraphQL est un outil puissant pour construire des API flexibles, efficaces et évolutives. Il permet aux clients de récupérer exactement les données dont ils ont besoin, tout en offrant une grande souplesse dans la manière dont les informations sont organisées et accessibles. Cela en fait une alternative moderne aux API REST, avec des avantages notables dans les performances, la flexibilité et la gestion des versions.

Un exemple simple d'une API GraphQL avec Flask en Python. Ce projet utilise la bibliothèque Graphene pour gérer les requêtes GraphQL et Flask pour l'API web.

Étapes de mise en place

1. Installez Flask et Graphene via pip :

```
pip install Flask graphene
```

Étape 1: Serveur Flask avec GraphQL

Tout d'abord, créez le serveur GraphQL avec Flask et Graphene comme dans l'exemple précédent.

Code du serveur (app.py)

```
from flask import Flask, request, jsonify
import graphene

# Définition du type GraphQL
class Query(graphene.ObjectType):
    hello = graphene.String(name=graphene.String(default_value="Stranger"))
    goodbye = graphene.String(name=graphene.String(default_value="Stranger"))

    def resolve_hello(self, info, name):
        return f"Hello, {name}!"

    def resolve_goodbye(self, info, name):
        return f"Goodbye, {name}!"

# Création du schéma GraphQL
schema = graphene.Schema(query=Query)

# Initialisation de l'application Flask
app = Flask(__name__)

@app.route("/graphql", methods=["POST"])
def graphql_server():
    data = request.get_json()
    if data is None or 'query' not in data:
        return jsonify({'error': 'Missing GraphQL query'}), 400

    # Exécution de la requête GraphQL
    result = schema.execute(data.get('query'))
    return jsonify(result.data)

if __name__ == "__main__":
    app.run(debug=True)
```

Explication du code serveur:

- **Deux résolutions** : hello et goodbye qui retournent respectivement des salutations et des adieux.
- Le serveur expose une API GraphQL sur le point de terminaison /graphql.

Étape 2: Client GraphQL (utilisant requests)

Maintenant, créons un client Python qui envoie des requêtes à cette API.

Code du client (client.py)

Installez d'abord la bibliothèque requests si vous ne l'avez pas encore installée :

```
pip install requests
```

Ensuite, créez un fichier Python `client.py` pour envoyer des requêtes au serveur Flask GraphQL.

```
import requests
import json

# L'URL de l'API GraphQL
url = "http://127.0.0.1:5000/graphql"

# Requête GraphQL pour demander un "hello"
query = """
{
  hello(name: "Alice")
}
"""

# Envoi de la requête au serveur Flask GraphQL
response = requests.post(url, json={'query': query})

# Affichage de la réponse
if response.status_code == 200:
    data = response.json()
    print("Réponse du serveur GraphQL : ", json.dumps(data, indent=2))
else:
    print(f"Erreur lors de la requête: {response.status_code}")
```

Explication du code client:

- Le client envoie une requête POST à l'API GraphQL avec une chaîne de requête qui demande une réponse avec le nom "Alice" pour le champ `hello`.
- Le client attend la réponse du serveur et l'affiche sous forme JSON formatée.

Étape 3: Exécution du serveur et du client

1. Exécutez le serveur:

Lancez d'abord le serveur Flask :

- `python app.py`

Cela va démarrer le serveur sur `http://127.0.0.1:5000`.

- **Exécutez le client :**

Ensuite, exécutez le client en ouvrant un autre terminal et en lançant :

- `python client.py`

Résultat attendu :

Lorsque vous exécutez le client, il enverra la requête au serveur Flask, et le serveur renverra la réponse au format JSON :

```
Réponse du serveur GraphQL :
{
  "data": {
    "hello": "Hello, Alice"
  }
}
```

```
}
```

Pour créer un client avec une interface web conviviale pour interagir avec l'API GraphQL, vous pouvez utiliser **Flask** pour le backend et **HTML/CSS/JavaScript** pour l'interface utilisateur. L'idée est de construire une page Web où l'utilisateur peut saisir des requêtes GraphQL, les envoyer au serveur et afficher les résultats de manière interactive.

Étapes pour construire l'interface :

1. **Backend (Flask avec GraphQL)** : Nous allons ajouter un moteur pour rendre une page web.
2. **Frontend (HTML/CSS/JS)** : Nous allons créer une interface simple avec un formulaire pour envoyer des requêtes GraphQL et afficher les résultats.

Étape 1: Mise à jour du backend Flask avec rendu HTML

Nous allons ajouter une route pour rendre une page HTML où l'utilisateur peut saisir sa requête GraphQL.

Code du serveur (app.py)

```
from flask import Flask, request, jsonify, render_template
import graphene

# Définition du type GraphQL
class Query(graphene.ObjectType):
    hello = graphene.String(name=graphene.String(default_value="Stranger"))
    goodbye = graphene.String(name=graphene.String(default_value="Stranger"))

    def resolve_hello(self, info, name):
        return f"Hello, {name}!"

    def resolve_goodbye(self, info, name):
        return f"Goodbye, {name}!"

# Création du schéma GraphQL
schema = graphene.Schema(query=Query)

# Initialisation de l'application Flask
app = Flask(__name__)

@app.route("/")
def index():
    # Afficher la page d'accueil avec un formulaire pour envoyer des requêtes GraphQL
    return render_template("index.html")

@app.route("/graphql", methods=["POST"])
def graphql_server():
    data = request.get_json()
    if data is None or 'query' not in data:
        return jsonify({'error': 'Missing GraphQL query'}), 400

    # Exécution de la requête GraphQL
    result = schema.execute(data.get('query'))
    return jsonify(result.data)
```

```
if __name__ == "__main__":
    app.run(debug=True)
```

Explication du serveur Flask :

- **Route /** : Rendre la page d'interface utilisateur avec un formulaire pour envoyer une requête GraphQL.
- **Route /graphql** : Cette route reçoit les requêtes GraphQL, les exécute et retourne la réponse.

Étape 2: Création de l'interface utilisateur (HTML/CSS/JS)

Créons un fichier HTML (`templates/index.html`) pour afficher une interface Web interactive.

Code HTML/CSS/JS pour l'interface utilisateur (`templates/index.html`)

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>GraphQL Client</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background-color: #f4f4f9;
      padding: 20px;
    }
    .container {
      max-width: 800px;
      margin: 0 auto;
      padding: 20px;
      background-color: white;
      border-radius: 8px;
      box-shadow: 0 2px 10px rgba(0,0,0,0.1);
    }
    h1 {
      text-align: center;
      color: #333;
    }
    textarea {
      width: 100%;
      height: 150px;
      padding: 10px;
      border-radius: 4px;
      border: 1px solid #ccc;
      margin-bottom: 10px;
    }
    button {
      display: block;
      width: 100%;
      padding: 10px;
      background-color: #007bff;
      color: white;
      border: none;
      border-radius: 4px;
      font-size: 16px;
      cursor: pointer;
    }
```

```

    }
    button: hover {
      background-color: #0056b3;
    }
  }
  .response {
    margin-top: 20px;
    padding: 10px;
    background-color: #e9ecef;
    border-radius: 4px;
  }
</style>
</head>
<body>

<div class="container">
  <h1>GraphQL Client</h1>
  <form id="graphqlForm">
    <textarea id="queryInput" placeholder="Enter your GraphQL query here"></textarea>
    <button type="submit">Send Query</button>
  </form>
  <div class="response" id="responseBox" style="display: none;"></div>
</div>

<script>
  document.getElementById('graphqlForm').addEventListener('submit',
function(event) {
  event.preventDefault();

  const query = document.getElementById('queryInput').value;

  fetch('/graphql', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({ query: query })
  })
  .then(response => response.json())
  .then(data => {
    const responseBox = document.getElementById('responseBox');
    responseBox.style.display = 'block';
    responseBox.innerHTML = `<pre>${JSON.stringify(data, null, 2)}</pre>`;
  })
  .catch(error => {
    console.error('Error:', error);
  });
});
</script>

</body>
</html>

```

Explication du code frontend :

- **Formulaire** : Un champ `textarea` est utilisé pour saisir des requêtes GraphQL. Lorsque l'utilisateur soumet le formulaire, la requête est envoyée via `fetch` à l'API `/graphql`.

- **Affichage de la réponse** : La réponse de l'API GraphQL est affichée dans un `div` sous forme formatée (JSON).

Étape 3: Démarrage du serveur Flask

1. Démarrez le serveur Flask :

Assurez-vous d'avoir la structure suivante :

```
- app.py
- templates/
  - index.html
```

Ensuite, lancez le serveur avec la commande suivante :

- `python app.py`

Cela va démarrer le serveur sur `http://127.0.0.1:5000`.

• Accéder à l'interface web :

Ouvrez un navigateur et allez sur `http://127.0.0.1:5000`. Vous verrez un formulaire où vous pouvez saisir une requête GraphQL, comme par exemple :

```
2. {
  hello(name: "Alice")
}
```

3. Envoyer une requête :

Après avoir saisi la requête, cliquez sur le bouton "Send Query". La réponse sera affichée sous forme JSON dans la page.

Exemple de requête :

Si vous entrez la requête suivante dans le formulaire :

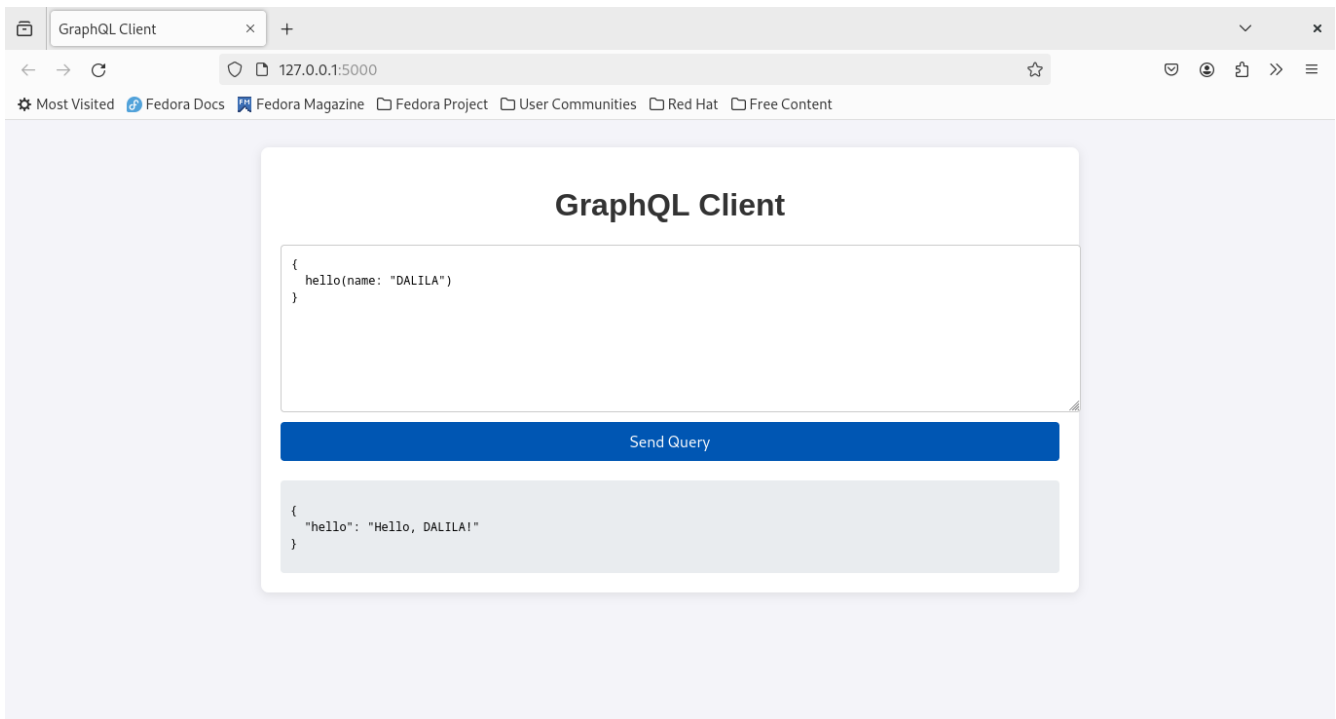
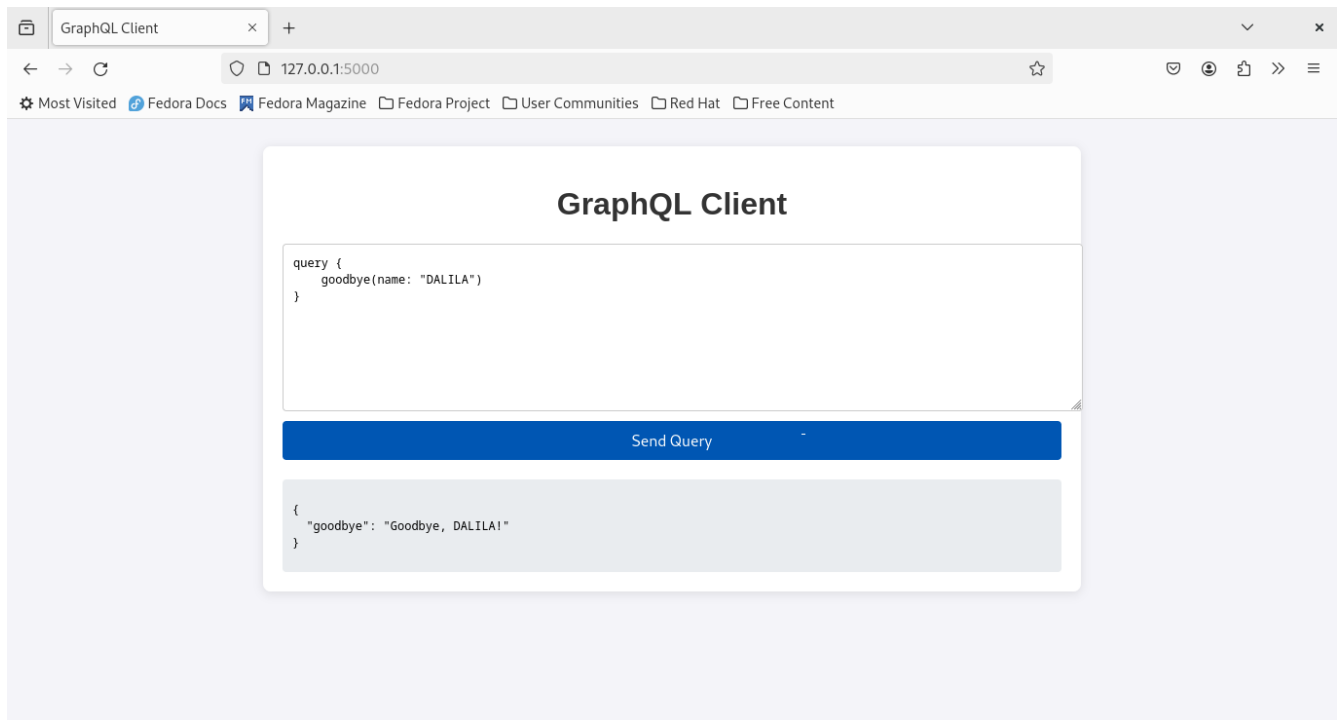
```
{
  hello(name: "Alice")
}
```

La réponse s'affichera dans la section "Response" sous ce format :

```
{
  "data": {
    "hello": "Hello, Alice!"
  }
}
```

```
dalilaiman@fedora:~/english3
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 819-121-323
127.0.0.1 - - [16/Mar/2025 15:54:15] "POST /graphql HTTP/1.1" 200 -
^Cdalilaiman@fedora:~/english3$ nano app.py
dalilaiman@fedora:~/english3$ python app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 819-121-323
127.0.0.1 - - [16/Mar/2025 15:58:22] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2025 16:00:27] "POST /graphql HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2025 16:13:29] "POST /graphql HTTP/1.1" 200 -
```

Pour ajouter des mutations et enrichir votre serveur GraphQL, vous pouvez définir une mutation dans votre schéma GraphQL et l'intégrer à votre serveur Flask. Une mutation permet de modifier des données sur le serveur, par exemple, changer un état ou ajouter de nouvelles informations.



1. Ajouter une Mutation

Nous allons créer une mutation simple pour ajouter un utilisateur et renvoyer un message de confirmation.

Voici un exemple de mise à jour de votre code avec une mutation `add_user`.

```

from flask import Flask, request, jsonify, render_template

import graphene

# Définition du type GraphQL

class Query(graphene.ObjectType):

    hello = graphene.String(name=graphene.String(default_value="Stranger"))

    goodbye = graphene.String(name=graphene.String(default_value="Stranger"))

    users = graphene.List(graphene.String)

    def resolve_hello(self, info, name):

        return f"Hello, {name}!"

    def resolve_goodbye(self, info, name):

        return f"Goodbye, {name}!"

    def resolve_users(self, info):

        # Retourne une liste d'utilisateurs fictifs

        return ["Alice", "Bob", "Charlie"]

# Définition de la Mutation

class CreateUser(graphene.Mutation):

    class Arguments:

        name = graphene.String(required=True)

    success = graphene.Boolean()

    message = graphene.String()

```

```

def mutate(self, info, name):

    # Exemple simple de création d'un utilisateur (pas de base de données ici)

    # Vous pouvez stocker l'utilisateur dans une base de données ou dans une
    liste pour plus de complexité.

    # Exemple de stockage dans une liste

    users_list.append(name) # Ajouter l'utilisateur à la liste globale fictive

    return CreateUser(success=True, message=f"User {name} created
successfully!")

# Définition du schéma de mutation

class Mutation(graphene.ObjectType):

    create_user = CreateUser.Field()

# Création du schéma GraphQL avec Query et Mutation

schema = graphene.Schema(query=Query, mutation=Mutation)

# Liste fictive d'utilisateurs pour démonstration

users_list = []

# Initialisation de l'application Flask

app = Flask(__name__)

@app.route("/")

def index():

    # Afficher la page d'accueil avec un formulaire pour envoyer des requêtes
    GraphQL

    return render_template("index.html")

```

```

@app.route("/graphql", methods=["POST"])

def graphql_server():

    data = request.get_json()

    if data is None or 'query' not in data:

        return jsonify({'error': 'Missing GraphQL query'}), 400

    # Exécution de la requête GraphQL

    result = schema.execute(data.get('query'))

    return jsonify(result.data)

if __name__ == "__main__":

    app.run(debug=True)

```

Explication :

- **Mutation** CreateUser : Cette mutation permet de créer un utilisateur. Elle prend un argument `name` et ajoute cet utilisateur à une liste globale (`users_list`).
- **Schéma** : Nous avons ajouté un schéma de mutation avec `create_user`, qui est un champ de la mutation.
- **Liste** `users_list` : C'est une liste simple pour stocker les utilisateurs créés, mais dans une application réelle, vous devriez connecter cela à une base de données.

Mise à jour de l'interface Front-End

Nous allons maintenant modifier le formulaire HTML pour permettre aux utilisateurs de soumettre des mutations à votre serveur.

Code HTML

```

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

```

```
<title>GraphQL with Flask</title>
</head>
<body>
  <h1>GraphQL Query & Mutation</h1>

  <h3>Query - Hello</h3>
  <form id="graphql-query-form">
    <textarea id="query" rows="5" cols="50" placeholder="Write your GraphQL query here (e.g.
{ hello(name: 'John') })"></textarea><br><br>
    <button type="submit">Submit Query</button>
  </form>

  <h3>Mutation - Create User</h3>
  <form id="graphql-mutation-form">
    <textarea id="mutation" rows="5" cols="50" placeholder="Write your GraphQL mutation here
(e.g. mutation { createUser(name: 'John') })"></textarea><br><br>
    <button type="submit">Create User</button>
  </form>

  <h3>Response:</h3>
  <pre id="response"></pre>

  <script>
    // Fonction pour envoyer la requête GraphQL
    function sendGraphQLRequest(query, isMutation = false) {
      const responseElement = document.getElementById("response");

      fetch("/graphql", {
        method: "POST",
        headers: {
```

```

        "Content-Type": "application/json"
    },
    body: JSON.stringify({ query: query })
})
.then(response => response.json())
.then(data => {
    responseElement.textContent = JSON.stringify(data, null, 2);
})
.catch(error => {
    responseElement.textContent = `Error: ${error.message}`;
});
}

```

// Soumettre une requête de type Query

```

document.getElementById("graphql-query-form").addEventListener("submit", function(event) {
    event.preventDefault();
    const query = document.getElementById("query").value;
    sendGraphQLRequest(query);
});

```

// Soumettre une requête de type Mutation

```

document.getElementById("graphql-mutation-form").addEventListener("submit", function(event)
{
    event.preventDefault();
    const mutation = document.getElementById("mutation").value;
    sendGraphQLRequest(mutation, true);
});

```

</script>

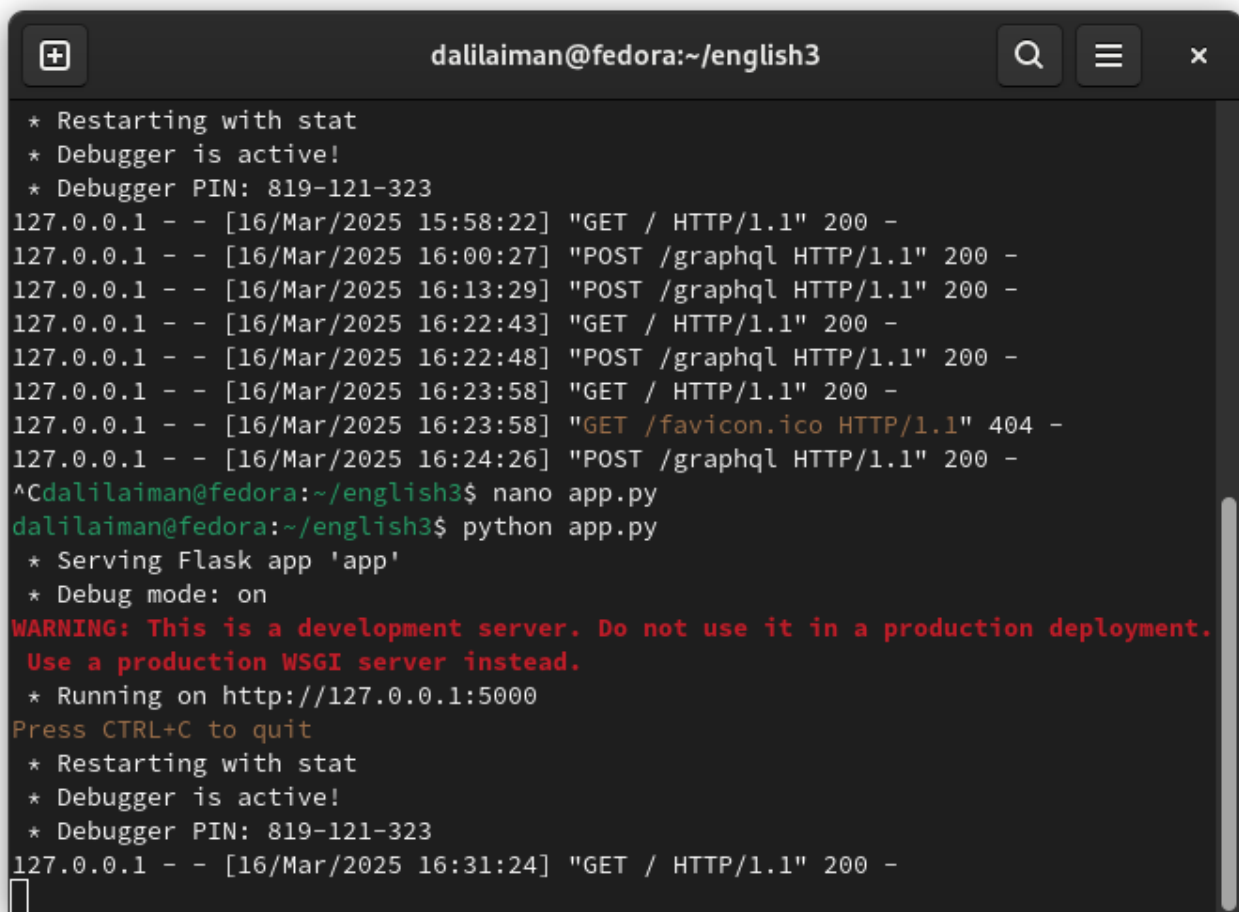
</body>

</html>

Explication :

1. **Formulaire pour Query** : L'utilisateur peut écrire une requête GraphQL pour appeler la fonction `hello` (ou d'autres fonctions du serveur).
2. **Formulaire pour Mutation** : L'utilisateur peut soumettre une mutation pour créer un utilisateur en appelant la mutation `createUser`.
3. **Gestion des requêtes et mutations** : Les deux formulaires envoient respectivement une requête ou une mutation au serveur.

3. Tester le Serveur



```
dalilaiman@fedora:~/english3
* Restarting with stat
* Debugger is active!
* Debugger PIN: 819-121-323
127.0.0.1 - - [16/Mar/2025 15:58:22] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2025 16:00:27] "POST /graphql HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2025 16:13:29] "POST /graphql HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2025 16:22:43] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2025 16:22:48] "POST /graphql HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2025 16:23:58] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [16/Mar/2025 16:23:58] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [16/Mar/2025 16:24:26] "POST /graphql HTTP/1.1" 200 -
^Cdalilaiman@fedora:~/english3$ nano app.py
dalilaiman@fedora:~/english3$ python app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 819-121-323
127.0.0.1 - - [16/Mar/2025 16:31:24] "GET / HTTP/1.1" 200 -
```

query {

```
hello(name: "DALILA")
}
```

Reponse:

```
{
  "data": {
    "hello": "Hello, John!"
  }
}
```



GraphQL Query & Mutation

Query - Hello

```
query {
  hello(name: "DALILA")
}
```

Submit Query

Mutation - Create User

```
Write your GraphQL mutation here (e.g. mutation {
  createUser(name: 'John') })
```

Create User

Response:

```
{
  "hello": "Hello, DALILA!"
}
```

```
mutation {
  createUser(name: "LAMIA") {
    success
    message
  }
}
```

```
}
```

Reponse

```
{
```

```
  "data": {
```

```
    "createUser": {
```

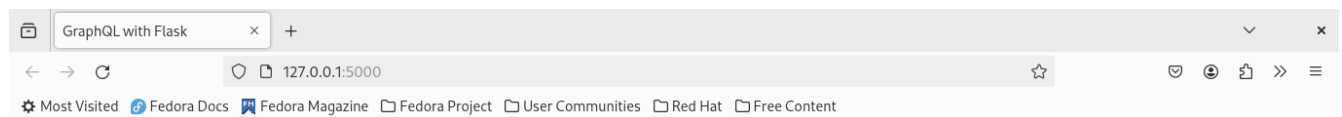
```
      "success": true,
```

```
      "message": "User Dave created successfully!"
```

```
    }
```

```
  }
```

```
}
```



Query - Hello

```
query {  
  hello(name: "DALILA")  
}
```

Submit Query

Mutation - Create User

```
success  
message  
}  
}
```

Create User

Response:

```
{  
  "createUser": {  
    "message": "User LAMIA created successfully!",  
    "success": true  
  }  
}
```

En résumé :

- Nous avons ajouté une mutation pour créer des utilisateurs (`createUser`).
- Nous avons enrichi l'interface front-end pour envoyer à la fois des requêtes et des mutations.
- Les utilisateurs peuvent interagir avec le serveur pour obtenir des données ou modifier des informations.

Cela vous donne une base solide pour un serveur GraphQL avec Flask, prêt à gérer des requêtes et des mutations.