Vamos analisar e explicar passo a passo como essa aplicação Flask funciona, desde o início até o fim. A aplicação é um sistema simples de gerenciamento de conexões em um rack, onde você pode adicionar, editar e excluir conexões entre portas de switch e portas de patch panel.

1. Estrutura do Projeto

O projeto é composto por:

- **app.py**: O arquivo principal da aplicação Flask, onde as rotas e a lógica do backend são definidas.
- **templates/**: Pasta que contém os arquivos HTML que serão renderizados pelo Flask.
    - **index.html**: Página principal que lista todas as conexões.
    - **add_connection.html**: Página para adicionar uma nova conexão.
    - **edit_connection.html**: Página para editar uma conexão existente.
- **static/**: Pasta que contém arquivos estáticos, como CSS.
    - **style.css**: Arquivo CSS para estilizar as páginas HTML.

2. Configuração do Flask e MySQL

No arquivo app.py, a aplicação Flask é configurada e conectada a um banco de dados MySQL.

```
from flask import Flask, render_template, request, redirect, url_for
import mysql.connector

app = Flask(__name__)

# Configuração do MySQL
db = mysql.connector.connect(
    host="localhost",
    user="root",
    password="",
    port=7306,
    database="rack_management"
)
```

- **Flask**: É um microframework para Python que facilita a criação de aplicações web.
- **mysql.connector**: Biblioteca para conectar e interagir com o banco de dados MySQL.
- **db**: Objeto de conexão com o banco de dados MySQL.

3. Rota Principal (/)

A rota principal (/) é responsável por exibir todas as conexões cadastradas no banco de dados.

```
@app.route('/')
def index():
    cursor = db.cursor(dictionary=True)
    cursor.execute("SELECT * FROM connections")
    connections = cursor.fetchall()
    return render_template('index.html', connections=connections)
```

- **cursor**: Um objeto que permite executar comandos SQL no banco de dados.
- **cursor.execute("SELECT * FROM connections")**: Executa uma consulta SQL para selecionar todas as conexões da tabela connections.
- **cursor.fetchall()**: Recupera todas as linhas resultantes da consulta.
- **render_template('index.html', connections=connections)**: Renderiza o template index.html, passando as conexões como contexto.

4. Rota para Adicionar Conexão (/add)

A rota /add permite adicionar uma nova conexão ao banco de dados.

```
@app.route('/add', methods=['GET', 'POST'])
def add_connection():
    if request.method == 'POST':
        switch_port = request.form['switch_port']
        patch_panel_port = request.form['patch_panel_port']
        cursor = db.cursor()
        cursor.execute("INSERT INTO connections (switch_port, patch_panel_port) VALUES (%s, %
s)", (switch_port, patch_panel_port))
        db.commit()
        return redirect(url_for('index'))
    return render_template('add_connection.html')
```

- **request.method == 'POST'**: Verifica se o formulário foi submetido.
- **request.form['switch_port']** e **request.form['patch_panel_port']**: Recuperam os dados do formulário.
- **cursor.execute("INSERT INTO connections ...")**: Insere uma nova conexão no banco de dados.
- **db.commit()**: Confirma a transação no banco de dados.
- **redirect(url_for('index'))**: Redireciona o usuário de volta para a página principal após a inserção.

5. Rota para Editar Conexão (/edit/<int:id>)

A rota /edit/<int:id> permite editar uma conexão existente.

```python
@app.route('/edit/<int:id>', methods=['GET', 'POST'])
def edit_connection(id):
    cursor = db.cursor(dictionary=True)
    if request.method == 'POST':
        switch_port = request.form['switch_port']
        patch_panel_port = request.form['patch_panel_port']
        cursor.execute("UPDATE connections SET switch_port = %s, patch_panel_port = %s WHERE id
= %s", (switch_port, patch_panel_port, id))
        db.commit()
        return redirect(url_for('index'))
    cursor.execute("SELECT * FROM connections WHERE id = %s", (id,))
    connection = cursor.fetchone()
    return render_template('edit_connection.html', connection=connection)
```

- **id**: O ID da conexão que será editada.
- **cursor.execute("UPDATE connections ...")**: Atualiza a conexão no banco de dados.
- **cursor.fetchone()**: Recupera a conexão específica para preencher o formulário de edição.

6. Rota para Excluir Conexão (/delete/<int:id>)

A rota /delete/<int:id> permite excluir uma conexão existente.

```python
@app.route('/delete/<int:id>')
def delete_connection(id):
    cursor = db.cursor()
    cursor.execute("DELETE FROM connections WHERE id = %s", (id,))
    db.commit()
    return redirect(url_for('index'))
```

- **cursor.execute("DELETE FROM connections ...")**: Exclui a conexão do banco de dados.

7. Templates HTML

Os templates HTML são responsáveis por renderizar a interface do usuário.

- **index.html**: Exibe uma tabela com todas as conexões e um layout visual do rack.
- **add_connection.html**: Formulário para adicionar uma nova conexão.
- **edit_connection.html**: Formulário para editar uma conexão existente.

8. Estilos CSS

O arquivo style.css contém os estilos para a aplicação, incluindo a formatação da tabela e do layout do rack.

9. Execução da Aplicação

No final do arquivo app.py, a aplicação é executada:

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

- **app.run(debug=True)**: Inicia o servidor Flask com o modo de depuração ativado, o que facilita o desenvolvimento.

10. Fluxo Completo da Aplicação

1. **Acesso à Página Principal (/)**:
    - O usuário acessa a página principal, que exibe todas as conexões cadastradas.
2. **Adicionar Nova Conexão (/add)**:
    - O usuário clica em "Add Connection" e é redirecionado para o formulário de adição.
    - Após preencher o formulário e submeter, a nova conexão é adicionada ao banco de dados e o usuário é redirecionado de volta para a página principal.
3. **Editar Conexão (/edit/<int:id>)**:
    - O usuário clica em "Edit" em uma conexão específica e é redirecionado para o formulário de edição.
    - Após editar e submeter o formulário, a conexão é atualizada no banco de dados e o usuário é redirecionado de volta para a página principal.
4. **Excluir Conexão (/delete/<int:id>)**:
    - O usuário clica em "Delete" em uma conexão específica e a conexão é removida do banco de dados.
    - O usuário é redirecionado de volta para a página principal.

11. Considerações Finais

Essa aplicação é um exemplo básico de como criar um CRUD (Create, Read, Update, Delete) usando Flask e MySQL. Para desenvolvedores iniciantes, é importante entender cada parte do código e como as diferentes tecnologias (Flask, MySQL, HTML, CSS) interagem entre si para criar uma aplicação funcional.

Vamos detalhar o funcionamento dos arquivos de templates (index.html, add_connection.html, e edit_connection.html) e como eles se relacionam com o app.py. Os templates são usados para gerar o HTML que é enviado ao navegador do usuário, e eles são renderizados pelo Flask com base nos dados fornecidos pelo app.py.

1. Estrutura dos Templates

Os templates são arquivos HTML que contêm placeholders (marcadores) para dados dinâmicos. Esses placeholders são substituídos por valores reais quando o template é renderizado pelo Flask. Os templates são armazenados na pasta templates/.

2. Relação entre app.py e os Templates

O Flask usa a função render_template para renderizar os templates HTML. Essa função recebe o nome do template e um conjunto de variáveis (contexto) que serão usadas para preencher os placeholders no template.

3. Detalhamento dos Templates

index.html

Este é o template principal que exibe todas as conexões cadastradas no banco de dados.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Rack Management</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
    <h1>Rack Management</h1>
    <a href="{{ url_for('add_connection') }}">Add Connection</a>
    <table>
        <tr>
            <th>Switch Port</th>
            <th>Patch Panel Port</th>
            <th>Actions</th>
        </tr>
        {% for connection in connections %}
        <tr>
            <td>{{ connection.switch_port }}</td>
            <td>{{ connection.patch_panel_port }}</td>
            <td>
                <a href="{{ url_for('edit_connection', id=connection.id) }}">Edit</a>
                <a href="{{ url_for('delete_connection', id=connection.id) }}" onclick="return confirm('Are you sure?')">Delete</a>
            </td>
        </tr>
        {% endfor %}
    </table>

    <div class="rack">
        {% for connection in connections %}
        <div class="port-pair">
            <div class="port switch-port">{{ connection.switch_port }}</div>
            <div class="port patch-panel-port">{{ connection.patch_panel_port }}</div>
        </div>
        {% endfor %}
    </div>
</body>
</html>
```

- **{{ url_for('static', filename='style.css') }}**: Gera a URL para o arquivo CSS estático.
- **{{ url_for('add_connection') }}**: Gera a URL para a rota add_connection.
- **{% for connection in connections %}**: Loop que itera sobre a lista de conexões passada pelo app.py.
- **{{ connection.switch_port }}** e **{{ connection.patch_panel_port }}**: Exibe os valores das portas de switch e patch panel.

- **{{ url_for('edit_connection', id=connection.id) }}**: Gera a URL para editar uma conexão específica.
- **{{ url_for('delete_connection', id=connection.id) }}**: Gera a URL para excluir uma conexão específica.

add_connection.html

Este template exibe um formulário para adicionar uma nova conexão.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Add Connection</title>
</head>
<body>
    <h1>Add Connection</h1>
    <form method="POST">
        <label for="switch_port">Switch Port:</label>
        <input type="text" name="switch_port" required>
        <br>
        <label for="patch_panel_port">Patch Panel Port:</label>
        <input type="text" name="patch_panel_port" required>
        <br>
        <button type="submit">Add</button>
    </form>
    <a href="{{ url_for('index') }}">Back</a>
</body>
</html>
```

- **<form method="POST">**: O formulário será submetido via método POST.
- **<input type="text" name="switch_port" required>**: Campo de entrada para a porta do switch.
- **<input type="text" name="patch_panel_port" required>**: Campo de entrada para a porta do patch panel.
- **<button type="submit">Add</button>**: Botão para submeter o formulário.
- **{{ url_for('index') }}**: Gera a URL para voltar à página principal.

edit_connection.html

Este template exibe um formulário para editar uma conexão existente.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Edit Connection</title>
</head>
<body>
    <h1>Edit Connection</h1>
    <form method="POST">
        <label for="switch_port">Switch Port:</label>
        <input type="text" name="switch_port" value="{{ connection.switch_port }}" required>
        <br>
        <label for="patch_panel_port">Patch Panel Port:</label>
        <input type="text" name="patch_panel_port" value="{{ connection.patch_panel_port }}" required>
        <br>
        <button type="submit">Save</button>
    </form>
    <a href="{{ url_for('index') }}">Back</a>
</body>
</html>
```

- **<input type="text" name="switch_port" value="{{ connection.switch_port }}" required>**: Campo de entrada para a porta do switch, pré-preenchido com o valor atual.
- **<input type="text" name="patch_panel_port" value="{{ connection.patch_panel_port }}" required>**: Campo de entrada para a porta do patch panel, pré-preenchido com o valor atual.
- **<button type="submit">Save</button>**: Botão para submeter o formulário.

4. Como os Templates são Renderizados

No app.py, a função render_template é usada para renderizar os templates HTML. Aqui estão os trechos relevantes:

Renderizando index.html

```python
@app.route('/')
def index():
    cursor = db.cursor(dictionary=True)
    cursor.execute("SELECT * FROM connections")
    connections = cursor.fetchall()
    return render_template('index.html', connections=connections)
```

- **render_template('index.html', connections=connections)**: Renderiza o template index.html e passa a lista de conexões (connections) como contexto.

Renderizando add_connection.html

```python
@app.route('/add', methods=['GET', 'POST'])
def add_connection():
    if request.method == 'POST':
        switch_port = request.form['switch_port']
        patch_panel_port = request.form['patch_panel_port']
        cursor = db.cursor()
        cursor.execute("INSERT INTO connections (switch_port, patch_panel_port) VALUES (%s, %
s)", (switch_port, patch_panel_port))
        db.commit()
        return redirect(url_for('index'))
    return render_template('add_connection.html')
```

- **render_template('add_connection.html')**: Renderiza o template add_connection.html quando o método HTTP é GET.

Renderizando edit_connection.html

```
@app.route('/edit/<int:id>', methods=['GET', 'POST'])
def edit_connection(id):
    cursor = db.cursor(dictionary=True)
    if request.method == 'POST':
        switch_port = request.form['switch_port']
        patch_panel_port = request.form['patch_panel_port']
        cursor.execute("UPDATE connections SET switch_port = %s, patch_panel_port = %s WHERE id
= %s", (switch_port, patch_panel_port, id))
        db.commit()
        return redirect(url_for('index'))
    cursor.execute("SELECT * FROM connections WHERE id = %s", (id,))
    connection = cursor.fetchone()
    return render_template('edit_connection.html', connection=connection)
```

- **render_template('edit_connection.html', connection=connection)**: Renderiza o template edit_connection.html e passa a conexão específica (connection) como contexto.

5. Fluxo de Dados entre app.py e os Templates

1. **Página Principal (/)**:
    - O Flask executa a consulta SQL para obter todas as conexões.
    - As conexões são passadas para o template index.html, que as exibe em uma tabela e em um layout visual do rack.
2. **Adicionar Conexão (/add)**:
    - Quando o usuário acessa a rota /add via GET, o Flask renderiza o template add_connection.html.
    - Quando o formulário é submetido via POST, o Flask processa os dados, insere a nova conexão no banco de dados e redireciona o usuário de volta para a página principal.
3. **Editar Conexão (/edit/<int:id>)**:
    - Quando o usuário acessa a rota /edit/<int:id> via GET, o Flask busca a conexão específica no banco de dados e renderiza o template edit_connection.html com os dados atuais.
    - Quando o formulário é submetido via POST, o Flask atualiza a conexão no banco de dados e redireciona o usuário de volta para a página principal.

6. Considerações Finais

Os templates são essenciais para separar a lógica de negócio (backend) da apresentação (frontend). Eles permitem que você crie páginas dinâmicas que podem ser facilmente atualizadas com base nos dados fornecidos pelo app.py. O uso de placeholders como {{ ... }} e estruturas de controle como {% ... %} torna os templates poderosos e flexíveis.

Para executar a aplicação Flask, você precisará instalar algumas bibliotecas Python usando o pip. Abaixo estão os comandos pip necessários e a explicação de por que cada um é usado:

1. Instalar o Flask

```
pip install Flask
```

- **Por que é usado?**
    - Flask é o microframework web que estamos usando para criar a aplicação. Ele fornece as ferramentas necessárias para criar rotas, renderizar templates e gerenciar requisições HTTP.

2. Instalar o MySQL Connector

```
pip install mysql-connector-python
```

- **Por que é usado?**
    - mysql-connector-python é uma biblioteca que permite a conexão e interação com um banco de dados MySQL. No seu código, você usa mysql.connector para se conectar ao banco de dados rack_management e executar consultas SQL.

3. Executar a Aplicação

Depois de instalar as dependências necessárias, você pode executar a aplicação usando o seguinte comando no terminal:

```
python app.py
```

- **Por que é usado?**
    - Este comando executa o script app.py, que inicia o servidor Flask. O servidor Flask escuta as requisições HTTP e as roteia para as funções apropriadas no seu código.

4. Considerações Finais

Aqui está um resumo dos comandos pip que você provavelmente precisará para executar a aplicação:

```
pip install Flask
pip install mysql-connector-python
```

5. Exemplo de requirements.txt

Para facilitar a instalação de todas as dependências, você pode criar um arquivo requirements.txt com o seguinte conteúdo:

```
Flask==2.3.2
mysql-connector-python==8.0.33
python-dotenv==0.21.0
```

E então instalar todas as dependências de uma vez com:

```
pip install -r requirements.txt
```

Isso garantirá que todas as bibliotecas necessárias estejam instaladas e nas versões corretas.

10. Conclusão

Os comandos pip são essenciais para instalar as bibliotecas necessárias para executar a aplicação Flask. Cada biblioteca tem um propósito específico, desde o framework web até a conexão com o banco de dados. Instalar as dependências corretas garante que a aplicação funcione como esperado.

O pip é o gerenciador de pacotes padrão para Python, e ele é usado para instalar, atualizar e gerenciar bibliotecas (também chamadas de pacotes ou dependências) que seu projeto precisa para funcionar. Vamos detalhar o que o pip faz no seu projeto/pasta e como ele lida com as bibliotecas.

O que o pip faz no seu projeto/pasta?

1. **Instala Bibliotecas**:
    - Quando você executa um comando como pip install <nome_da_biblioteca>, o pip baixa a biblioteca especificada e suas dependências (outras bibliotecas que ela precisa para funcionar) e as instala no ambiente Python.
2. **Gerencia Dependências**:
    - O pip garante que todas as bibliotecas necessárias estejam instaladas e nas versões corretas. Ele também resolve conflitos entre versões de bibliotecas, se possível.
3. **Armazena Bibliotecas em um Local Específico**:
    - Por padrão, o pip instala as bibliotecas em um diretório global do Python (por exemplo, site-packages dentro do diretório de instalação do Python). Isso significa que as bibliotecas **não são instaladas diretamente na pasta do seu projeto**, a menos que você esteja usando um ambiente virtual (explicado abaixo).
4. **Cria um Ambiente Isolado (se usado com venv ou virtualenv)**:
    - Se você estiver usando um ambiente virtual (como venv ou virtualenv), o pip instala as bibliotecas em uma pasta específica dentro do ambiente

virtual (por exemplo, venv/lib/python3.x/site-packages). Isso isola as bibliotecas do projeto, evitando conflitos com outros projetos ou com o Python global.

5. **Gera um Arquivo requirements.txt**:
    - O pip pode ser usado para gerar um arquivo requirements.txt, que lista todas as bibliotecas instaladas no projeto e suas versões. Isso é útil para compartilhar o projeto com outras pessoas ou para reinstalar as dependências em outro ambiente.

O pip carrega bibliotecas ocultas na minha pasta?

Não, o pip **não carrega bibliotecas diretamente na pasta do seu projeto** (a menos que você esteja usando um ambiente virtual). Aqui está o que acontece:

1. **Instalação Global**:
    - Se você não estiver usando um ambiente virtual, o pip instala as bibliotecas no diretório global do Python, que geralmente está em um caminho como:
        - No Windows: C:\Users\<seu_usuario>\AppData\Local\Programs\Python\Python3x\Lib\site-packages
        - No Linux/macOS: /usr/local/lib/python3.x/site-packages
    - Essas bibliotecas **não aparecem na pasta do seu projeto**, mas estão disponíveis para qualquer script Python no seu sistema.
2. **Instalação em Ambiente Virtual**:
    - Se você estiver usando um ambiente virtual (recomendado), o pip instala as bibliotecas em uma pasta específica dentro do ambiente virtual, como venv/lib/python3.x/site-packages. Nesse caso, as bibliotecas **ainda não aparecem diretamente na pasta do seu projeto**, mas estão isoladas no ambiente virtual.
3. **Arquivos no Projeto**:
    - O único arquivo relacionado ao pip que pode aparecer na pasta do seu projeto é o requirements.txt, que lista as dependências do projeto. Esse arquivo é criado manualmente ou gerado com o comando pip freeze > requirements.txt.

Como o pip funciona com ambientes virtuais?

Ambientes virtuais (como venv ou virtualenv) são uma prática recomendada para projetos Python. Eles criam um ambiente isolado onde as bibliotecas instaladas pelo pip são armazenadas em uma pasta específica dentro do projeto. Aqui está como funciona:

1. **Criar um Ambiente Virtual**:

- No terminal, execute:

```
python -m venv venv
```

- Isso cria uma pasta chamada venv no seu projeto.

2. **Ativar o Ambiente Virtual**:
   - No Windows:

```
venv\Scripts\activate
```

3. **Instalar Bibliotecas no Ambiente Virtual**:
   - Com o ambiente virtual ativado, use o pip para instalar as bibliotecas:

```
pip install Flask mysql-connector-python
```

   - As bibliotecas serão instaladas na pasta venv/lib/python3.x/site-packages.

4. **Desativar o Ambiente Virtual**:
   - Quando terminar, você pode desativar o ambiente virtual com:

```
deactivate
```

Onde as bibliotecas ficam armazenadas?

- **Sem Ambiente Virtual**:
  - As bibliotecas são instaladas globalmente no diretório site-packages do Python.
- **Com Ambiente Virtual**:
  - As bibliotecas são instaladas na pasta venv/lib/python3.x/site-packages (ou similar, dependendo do nome do ambiente virtual).

Por que usar um ambiente virtual?

1. **Isolamento**:
   - Evita conflitos entre bibliotecas de diferentes projetos. Cada projeto pode ter suas próprias versões de bibliotecas.
2. **Portabilidade**:
   - Facilita a replicação do ambiente em outra máquina usando o arquivo requirements.txt.
3. **Organização**:
   - Mantém as dependências do projeto separadas das bibliotecas globais do sistema.

Resumo

- O pip instala bibliotecas Python e suas dependências.
- Por padrão, as bibliotecas são instaladas globalmente ou em um ambiente virtual, **não diretamente na pasta do projeto**.
- Usar um ambiente virtual é a melhor prática para isolar as dependências do projeto.
- O pip não carrega bibliotecas ocultas na pasta do projeto, a menos que você esteja usando um ambiente virtual.

Se você seguir as práticas recomendadas (como usar um ambiente virtual), o pip ajudará a manter seu projeto organizado e livre de conflitos de dependências.

Vamos criar um resumo didático de como o arquivo app.py se relaciona com cada um dos templates (index.html, add_connection.html, e edit_connection.html). Isso ajudará a entender como o backend (Flask) e o frontend (HTML) interagem para criar a aplicação.

Resumo Geral

O app.py é o coração da aplicação Flask. Ele define as rotas (URLs) que o usuário pode acessar e decide qual template HTML será renderizado em cada rota. Além disso, ele gerencia a lógica de negócio, como consultas ao banco de dados e manipulação de formulários.

Cada template HTML é responsável por exibir uma interface ao usuário e, quando necessário, enviar dados de volta ao app.py para processamento (por exemplo, ao enviar um formulário).

Relação entre app.py e os Templates

1. Rota Principal (/) - index.html

- **Função no app.py**:

```
@app.route('/')
def index():
    cursor = db.cursor(dictionary=True)
    cursor.execute("SELECT * FROM connections")
    connections = cursor.fetchall()
    return render_template('index.html', connections=connections)
```

- **O que acontece?**

1. O Flask executa uma consulta SQL para buscar todas as conexões no banco de dados.
2. A lista de conexões (connections) é passada para o template index.html.
3. O template index.html é renderizado com os dados das conexões.

- **Template index.html**:
  - Exibe uma tabela com todas as conexões.
  - Inclui links para adicionar, editar e excluir conexões.
  - Usa a variável connections (passada pelo app.py) para preencher a tabela e o layout visual do rack.
- **Exemplo de interação**:
  - O usuário acessa a página inicial (/).
  - O Flask renderiza o index.html com as conexões do banco de dados.
  - O usuário vê a lista de conexões e pode interagir com os links para adicionar, editar ou excluir.

---

2. Rota para Adicionar Conexão (/add) - add_connection.html

- **Função no app.py**:

```
@app.route('/add', methods=['GET', 'POST'])
def add_connection():
    if request.method == 'POST':
        switch_port = request.form['switch_port']
        patch_panel_port = request.form['patch_panel_port']
        cursor = db.cursor()
        cursor.execute("INSERT INTO connections (switch_port, patch_panel_port) VALUES (%s,
%s)", (switch_port, patch_panel_port))
        db.commit()
        return redirect(url_for('index'))
    return render_template('add_connection.html')
```

- **O que acontece?**
  1. Se o método for GET, o Flask renderiza o template add_connection.html, que exibe um formulário para adicionar uma nova conexão.
  2. Se o método for POST (quando o formulário é enviado), o Flask:
     - Captura os dados do formulário (switch_port e patch_panel_port).
     - Insere uma nova conexão no banco de dados.
     - Redireciona o usuário de volta para a página inicial (/).
- **Template add_connection.html**:
  - Exibe um formulário com dois campos: switch_port e patch_panel_port.

- Quando o formulário é enviado, os dados são passados para o app.py via método POST.

- **Exemplo de interação**:
  - O usuário clica em "Add Connection" na página inicial.
  - O Flask renderiza o add_connection.html.
  - O usuário preenche o formulário e clica em "Add".
  - O Flask processa os dados e redireciona o usuário de volta para a página inicial.

---

3. Rota para Editar Conexão (/edit/<int:id>) - edit_connection.html

- **Função no app.py**:

```
@app.route('/edit/<int:id>', methods=['GET', 'POST'])
def edit_connection(id):
    cursor = db.cursor(dictionary=True)
    if request.method == 'POST':
        switch_port = request.form['switch_port']
        patch_panel_port = request.form['patch_panel_port']
        cursor.execute("UPDATE connections SET switch_port = %s, patch_panel_port = %s WHERE
id = %s", (switch_port, patch_panel_port, id))
        db.commit()
        return redirect(url_for('index'))
    cursor.execute("SELECT * FROM connections WHERE id = %s", (id,))
    connection = cursor.fetchone()
    return render_template('edit_connection.html', connection=connection)
```

- **O que acontece?**
  1. Se o método for GET, o Flask:
     - Busca a conexão específica no banco de dados com base no id fornecido na URL.
     - Renderiza o template edit_connection.html, passando os dados da conexão para preencher o formulário.
  2. Se o método for POST (quando o formulário é enviado), o Flask:
     - Captura os dados do formulário (switch_port e patch_panel_port).
     - Atualiza a conexão no banco de dados.
     - Redireciona o usuário de volta para a página inicial (/).

- **Template edit_connection.html**:
  - Exibe um formulário pré-preenchido com os dados da conexão que está sendo editada.
  - Quando o formulário é enviado, os dados são passados para o app.py via método POST.

- **Exemplo de interação**:
    - O usuário clica em "Edit" em uma conexão na página inicial.
    - O Flask renderiza o edit_connection.html com os dados da conexão selecionada.
    - O usuário edita os campos e clica em "Save".
    - O Flask processa os dados e redireciona o usuário de volta para a página inicial.

---

4. Rota para Excluir Conexão (/delete/<int:id>)

- **Função no app.py**:

```
@app.route('/delete/<int:id>')
def delete_connection(id):
    cursor = db.cursor()
    cursor.execute("DELETE FROM connections WHERE id = %s", (id,))
    db.commit()
    return redirect(url_for('index'))
```

- **O que acontece?**
    1. Quando o usuário clica em "Delete" em uma conexão, o Flask:
        - Executa uma consulta SQL para excluir a conexão com o id fornecido.
        - Redireciona o usuário de volta para a página inicial (/).
- **Template index.html**:
    - Inclui links para excluir conexões.
    - Usa JavaScript para confirmar a exclusão antes de enviar a requisição.
- **Exemplo de interação**:
    - O usuário clica em "Delete" em uma conexão na página inicial.
    - O Flask exclui a conexão do banco de dados e redireciona o usuário de volta para a página inicial.

Fluxo Completo da Aplicação

1. **Página Inicial (/)**:
    - O Flask renderiza o index.html com as conexões do banco de dados.
    - O usuário vê a lista de conexões e pode interagir com os links para adicionar, editar ou excluir.

2. **Adicionar Conexão (/add)**:
    - O Flask renderiza o add_connection.html com um formulário vazio.
    - O usuário preenche o formulário e o envia.
    - O Flask processa os dados e redireciona o usuário de volta para a página inicial.
3. **Editar Conexão (/edit/<int:id>)**:
    - O Flask renderiza o edit_connection.html com os dados da conexão selecionada.
    - O usuário edita os campos e envia o formulário.
    - O Flask processa os dados e redireciona o usuário de volta para a página inicial.
4. **Excluir Conexão (/delete/<int:id>)**:
    - O Flask exclui a conexão do banco de dados e redireciona o usuário de volta para a página inicial.

Resumo Visual

Rota	**Método HTTP**	**Template**	**Ação**
/	GET	index.html	Exibe todas as conexões.
/add	GET	add_connection.html	Exibe formulário para adicionar conexão.
/add	POST	-	Processa dados do formulário e adiciona conexão ao banco de dados.
/edit/<int:id>	GET	edit_connection.html	Exibe formulário pré-preenchido para editar conexão.
/edit/<int:id>	POST	-	Processa dados do formulário e atualiza conexão no banco de dados.
/delete/<int:id>	GET	-	Exclui conexão do banco de dados.

Conclusão

O app.py atua como o controlador central da aplicação, gerenciando a lógica de negócio e decidindo qual template renderizar com base na rota e no método HTTP. Os templates (index.html, add_connection.html, e edit_connection.html) são responsáveis por exibir a interface do usuário e coletar dados, que são então processados pelo app.py. Essa separação entre backend e frontend é fundamental para a organização e manutenção da aplicação.