UFCD 5117
Laboratório 10
Introdução ao Flask e construção de uma API

v1.0

Nelson Santos
nelson.santos.0001376@edu.atec.pt

ATEC — 6 de janeiro de 2025

Conteúdo

1 Introdução ao Flask

`Flask` é uma `framework` web minimalista escrita em `Python`, amplamente utilizada para o desenvolvimento rápido de aplicações web, incluindo APIs REST (*Application Programming Interface - Representational State Transfer*). A sua simplicidade e flexibilidade tornam-na ideal para projetos de pequena e média escala, mas a sua capacidade de extensão permite que também seja usada em projetos mais complexos.

1.1 O que é REST e outros protocolos de comunicação

REST é um estilo de arquitetura amplamente utilizado no desenvolvimento de APIs, que facilita a comunicação entre sistemas (tipicamente cliente - servidor). Este modelo baseia-se no protocolo HTTP, seguindo princípios simples para o acesso e manipulação de recursos.

Os principais princípios do REST incluem:

- **Recursos identificados por URIs**: Cada recurso é identificado por um URI único
  - Exemplo: `https://api.exemplo.com/utilizadores/123` representa um utilizador com o ID 123
- **Operações HTTP padrão**: REST utiliza métodos HTTP como:
  - `GET`: Para obter dados
  - `POST`: Para criar novos dados
  - `PUT`: Para atualizar dados existentes
  - `DELETE`: Para eliminar dados
- **Arquitetura sem estado (*stateless*)**: Cada pedido contém toda a informação necessária para ser processado, não havendo armazenamento do estado no servidor
- **Representação dos recursos**: Os recursos podem ser representados em vários formatos, como JSON ou XML. O formato mais comum é o JSON, devido à sua simplicidade

1.1.1 Outros protocolos de comunicação

Embora o REST seja amplamente utilizado, existem outros protocolos e estilos de arquitetura para comunicação entre sistemas:

- **SOAP (Simple Object Access Protocol)**:
  - Um protocolo mais antigo que utiliza XML para trocar mensagens entre sistemas
  - É frequentemente utilizados em ambientes empresariais devido à sua robustez e suporte para transações complexas
- **GraphQL**:
  - Desenvolvido pelo Facebook, permite que os clientes definam exatamente quais dados precisam, evitando o envio de dados desnecessários
  - Muito utilizado em aplicações modernas devido à sua flexibilidade
- **gRPC (Google Remote Procedure Call)**:
  - Um protocolo de alto desempenho desenvolvido pela Google
  - Utiliza HTTP/2 e Protobuf para uma comunicação eficiente e rápida
  - Ideal para micro-serviços ou sistemas distribuídos
- **AMQP (Advanced Message Queuing Protocol)**:
  - Um protocolo de mensagens assíncronas usado em sistemas baseados em queues (como RabbitMQ)
  - Útil para comunicação entre serviços que não precisam de interação em tempo real

Cada protocolo ou estilo de arquitetura é adequado a diferentes tipos de aplicações, dependendo das necessidades específicas, como desempenho, simplicidade ou suporte para transações complexas.

1.2 Códigos HTTP: códigos de resposta em APIs REST

Quando se utiliza uma API REST, cada pedido HTTP feito ao servidor devolve um **código de estado HTTP** (*HTTP status code*). Estes códigos são números de três dígitos que indicam o resultado do pedido realizado. Os códigos estão divididos em categorias, e cada categoria tem um propósito específico. De seguida estão os códigos mais comuns que as APIs REST costumam devolver:

1.2.1 Categorias de códigos HTTP

- **1xx - informativos**: Indicam que o pedido foi recebido e está a ser processado. Estes códigos raramente são usados em APIs
- **2xx - sucesso**: Indicam que o pedido foi bem-sucedido
- **3xx - redirecionamento**: Indicam que é necessário realizar uma ação adicional para completar o pedido
- **4xx - erros do cliente**: Indicam que houve um erro no pedido feito pelo cliente
- **5xx - erros do servidor**: Indicam que o servidor encontrou um problema ao processar o pedido

1.2.2 Códigos HTTP comuns em APIs REST

- **200 OK**:
  - Indica que o pedido foi bem-sucedido e os dados solicitados (se aplicável) foram devolvidos
  - Exemplo: Um pedido `GET` a um recurso que retorna os dados no formato JSON
- **201 Created**:
  - Indica que um recurso foi criado com sucesso no servidor
  - Exemplo: Um pedido `POST` que cria um novo utilizador na base de dados
- **204 No Content**:
  - Indica que o pedido foi processado com sucesso, mas não há conteúdo para retornar
  - Exemplo: Um pedido `DELETE` para apagar um recursos sem retorno de dados
- **400 Bad Request**:
  - Indica que o pedido enviado pelo cliente é inválido ou contém erros
  - Exemplo: Campos obrigatórios estão ausentes no `POST`
- **401 Unauthorized**:
  - Indica que o cliente não está autenticado para aceder ao recurso solicitado
  - Exemplo: Tentativa de acesso a uma API protegida sem enviar o token de autenticação
- **403 Forbidden**:
  - Indica que o cliente não tem permissão para aceder ao recurso, mesmo que esteja autenticado
  - Exemplo: Um utilizador com permissões insuficientes tenta aceder a um recurso restrito
- **404 Not Found**:
  - Indica que o recurso solicitado não foi encontrado no servidor
  - Exemplo: Um `GET` para um ID de um recurso inexistente
- **500 Internal Server Error**:
  - Indica que ocorreu um erro inesperado no servidor
  - Exemplo: Uma exceção não tratada no código da API
- **503 Service Unavailable**:
  - Indica que o servidor não está disponível para processar o pedido no momento
  - Exemplo: API fora de serviço devido a manutenção ou sobrecarga

1.2.3 Utilização dos Códigos HTTP em APIs REST

O uso correto dos códigos de estado HTTP é essencial para garantir que os clientes compreendam os resultados dos pedidos. Por exemplo:

- **200 OK** deve ser usado sempre que o pedido é bem-sucedido e retorna dados
- **404 Not Found** ajuda o cliente a perceber que o recurso solicitado não existe
- **400 Bad Request** informa que o cliente deve corrigir o pedido antes de tentar novamente
- **500 Internal Server Error** indica que o servidor encontrou um problema inesperado e deve ser corrigido pelo programador

A correta implementação destes códigos melhora a experiência do utilizador, facilita o `debug` e torna a API mais profissional e previsível.

1.3 Porquê escolher o Flask?

`Flask` destaca-se por ser uma `framework` de micro-serviços. Diferente de `frameworks` mais completos, como o `Django`, `Flask` permite maior controlo e personalização sobre os componentes que são utilizados, oferecendo apenas o minimo necessário para construir uma aplicação web. Isto permite ao programador escolher exatamente as ferramentas que deseja integrar no projeto.

1.4 Alternativas ao Flask

Embora `Flask` seja amplamente utilizado, existem outras opções populares no desenvolvimento de aplicações web em `Python` incluem:

- **Django**: Uma `framework` muito completa que inclui muitas funcionalidades integradas, como autenticação de utilizadores, ORM (*Object-Relational Mapping*) e um back-office ou dashboard. Ideal para projetos mais complexos
- **FastAPI**: Uma `framework` moderna otimizada para performance. Possui suporte nativo a tipagem em `Python` e gera documentação automática de APIs utilizando o padrão OpenAPI
- **Bottle**: Semelhante a `Flask`, é uma `framework` minimalista adequada para aplicações simples e protótipos rápidos

1.5 Vantagens do Flask

O `Flask` apresenta diversas vantagens que o tornam uma escolha popular para muitos programadores:

- **Flexibilidade**: Permite a escolha de bibliotecas e ferramentas específicas para cada necessidade do projeto
- **Facilidade de aprendizagem**: A curva de aprendizagem é relativamente baixa, sendo adequado para iniciantes em desenvolvimento web
- **Comunidade ativa**: Existem muitos tutoriais, bibliotecas e recursos disponíveis online para ajudar a resolver problemas comuns
- **Extensibilidade**: Apesar de ser minimalista, o `Flask` pode ser ampliado com plugins e bibliotecas adicionais, como `Flask-SQLAlchemy` para manipulação de bases de dados ou `Flask-JWT-Extended` para autenticação

Neste laboratório, será explorado como configurar o `Flask`, criar uma API funcional e compreender as suas principais funcionalidades.

2 Configuração do `Flask`

Nesta secção, será configurado o ambiente de trabalho para o desenvolvimento com o `Flask`. A configuração correta é fundamental para garantir um ambiente isolado, eficiente e sem conflitos de dependências.

2.1 Requisitos

Antes de iniciar, assegure-se de que o seu sistema cumpre os seguintes requisitos:

- `Python`: Certifique-se de ter a versão 3.8 ou superior instalada no sistema
- **Pip**: O gestor de pacotes `pip` deve estar disponível no sistema
- **Ambiente Virtual (opcional, mas recomendado)**: O uso de ambientes virtuais (`venv`) é recomendado para evitar conflitos de dependências entre diferentes projetos

2.2 Passos para Configuração

Siga os passos para configurar o ambiente e iniciar o desenvolvimento com o `Flask`.

2.2.1 Criar um Ambiente Virtual

Siga os passos do Laboratório 7 para a criação de uma ambiente virtual.

2.2.2 Instalar `Flask`

Com o ambiente virtual ativo, instale o `Flask` com o seguinte comando: `pip install flask`. Após a instalação, verifique a versão do `Flask` para confirmar que foi instalada corretamente: `python -m flask -version`

2.2.3 Criar um Projeto Flask

1. Crie um ficheiro `app.py` na pasta raiz do projeto com o seguinte código inicial:

```
from flask import Flask

# Criar a aplicacao Flask
app = Flask(__name__)

# Definir a rota principal
@app.route('/')
def home():
    return "Ola, Flask!"

# Iniciar a aplicacao no modo de debug
if __name__ == '__main__':
    app.run(debug=True)
```

2. Guarde o ficheiro e execute a aplicação.
3. Após a execução, abra o seu `browser` e aceda ao endereço: `http://127.0.0.1:5000` Deverá ver a mensagem `Olá, Flask!` no `browser`.

3 Construção de uma API

Com o `Flask` configurado, vamos criar uma API básica que oferece suporte para operações CRUD (Criar, Ler, Atualizar e Apagar recursos). Esta API pode ser usada para manipular recursos - produtos genéricos.

3.1 Definição de rotas

As rotas são responsáveis por mapear os pedidos HTTP para funções específicas. Vamos adicionar rotas ao ficheiro `app.py` para gerir os recursos - `produtos`.

Vamos ter rotas para:

- `Create - GET /produtos`: Obter todos os produtos
- `Read - POST /produtos`: Adicionar um novo produto
- `Update - PUT /produtos/id`: Atualizar um produto existente
- `Delete - DELETE /produtos/id`: Apagar um produto

3.1.1 Operação GET

A operação `GET` é usada para obter informações sobre os produtos. Esta rota irá retornar uma lista de produtos disponíveis.

```
@app.route('/produtos', methods=['GET'])
def get_produtos():
    # Retorna uma lista de produtos
    produtos = ["Produto1", "Produto2", "Produto3"]
    return {"produtos": produtos}
```

Esta rota será acionada quando um request `GET` for feita para `/produtos`, retornando uma lista de produtos no formato JSON.

Aceda ao seu `browser` e escreva `http://localhost:5000/produtos` para ver a lista de produtos.

Pode também utilizar outras ferramentas como o Postman ou Httpie para testar a API, o que é aconselhável.

Teste a rota com o seguinte url: **GET** | `http://localhost:5000/produtos/`

3.1.2 Operação POST

A operação `POST` é usada para criar novos produtos. Esta rota permite adicionar um novo produto à lista.

```
@app.route('/produtos', methods=['POST'])
def add_produto():
    # Adiciona um novo produto
    novo_produto = {"nome": "Produto4", "preco": 20.0}
    return {"message": "Produto adicionado com sucesso!", "produto": novo_produto}, 201
```

A rota `POST` permite que um novo produto seja adicionado. A resposta incluirá uma mensagem de sucesso e as informações do produto.

Teste a rota com o seguinte url: **POST** | `http://localhost:5000/produtos/`

3.1.3 Operação PUT

A operação `PUT` é usada para atualizar um produto existente. Esta rota permite modificar as propriedades de um produto específico, como o nome ou o preço.

```
@app.route('/produtos/<int:id>', methods=['PUT'])
def update_produto(id):
    # Atualiza as informacoes de um produto existente
    produto_atualizado = {"id": id, "nome": "Produto Atualizado", "preco": 25.0}
    return {"message": "Produto atualizado com sucesso!", "produto": produto_atualizado}
```

Neste caso, a rota usa um parâmetro dinâmico `<int:id>` para identificar o produto que será atualizado. O corpo da resposta irá conter o produto atualizado.

Teste a rota com o seguinte url: **PUT** | `http://localhost:5000/produtos/1`

3.1.4 Operação DELETE

A operação `DELETE` é usada para excluir um produto. Esta rota permite remover um produto da lista com base no seu identificador.

```
@app.route('/produtos/<int:id>', methods=['DELETE'])
def delete_produto(id):
    # Apaga um produto com base no ID fornecido
    return {"message": f"Produto {id} apagado com sucesso!"}, 200
```

A rota `DELETE` aceita um identificador de produto e, ao ser acionada, remove o produto da lista.

Teste a rota com o seguinte url: **DELETE** | `http://localhost:5000/produtos/1`

4 Integração com Base de Dados

Para garantir a persistência de dados na nossa aplicação `Flask`, vamos utilizar o `SQLAlchemy`, que é uma biblioteca de mapeamento objeto-relacional (ORM). Esta ferramenta facilita a interação entre a aplicação e a base de dados, permitindo que utilizemos objetos `Python` em vez de escrever SQL manualmente.

4.1 Instalação e configuração

A primeira etapa para integrar o `SQLAlchemy` com a aplicação `Flask` é instalar as bibliotecas necessárias e configurar a conexão com a base de dados.

1. **Instalar o `SQLAlchemy` e a biblioteca de suporte ao SQLite:**
   O `SQLite` é uma base de dados leve e embutida que pode ser facilmente utilizada para desenvolvimento. Para instalar o `SQLAlchemy` e o suporte para SQLite, execute o seguinte comando no terminal:

   ```
   pip install sqlalchemy flask_sqlalchemy
   ```

2. **Configurar o `SQLAlchemy` no Flask:**
   Após a instalação, precisamos configurar o Flask para usar o `SQLAlchemy`. A configuração básica define o URI da base de dados e cria uma instância do `SQLAlchemy`. Para guardar os dados de forma simples, vamos usar um ficheiro SQLite chamado `app.db`. O código de seguida exemplifica a configuração inicial:

   ```
   from flask import Flask
   from flask_sqlalchemy import SQLAlchemy

   app = Flask(__name__)

   # Configuracao do URI da base de dados
   app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///app.db'
   app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False  # Desativa o historico de
       alteracoes

   # Inicializacao do SQLAlchemy
   db = SQLAlchemy(app)
   ```

3. **Definir um modelo (classe) para uma tabela:**
   De seguida, definimos a estrutura da nossa base de dados criando uma classe que representa a tabela desejada. Cada classe será mapeada para uma tabela na base de dados, com cada atributo da classe correspondendo a uma coluna na tabela.

   De seguida está um exemplo de modelo para uma tabela chamada `Produto`:

   ```
   class Produto(db.Model):
       # Definicao das colunas da tabela
       id = db.Column(db.Integer, primary_key=True)  # Chave primaria
       nome = db.Column(db.String(80), nullable=False)  # Nome do produto, nao pode ser
        nulo

       def __repr__(self):
           return f'<Produto {self.nome}>'
   ```

   Neste exemplo: - A classe `Produto` herda de `db.Model`, o que permite que ela seja tratada como uma tabela na base de dados - A coluna `id` é definida como uma chave primária (`primary_key=True`) - A coluna `nome` guarda o nome do produto e é obrigatória (`nullable=False`)

4. **Criar a Base de Dados:**
   Após definir o modelo, podemos criar a tabela na base de dados. Para isso, basta chamar o método `create_all()`, que irá criar todas as tabelas definidas nas classes de modelo.

   Adicione o seguinte código no final do ficheiro `app.py`:

   ```
   # Cria todas as tabelas na base de dados
   with app.app_context():
       db.create_all()
   ```

   O código acima cria as tabelas necessárias, incluindo a tabela `Produto`, se ainda não existir.

5. **Verificação:**
   Para garantir que tudo está configurado corretamente, pode-se verificar se a tabela foi criada na base de dados SQLite. O ficheiro deverá encontrar-se dentro da pasta `instance`. Poderá usar uma ferramenta como o DB Browser for SQLite, permitindo visualizar o conteúdo da base de dados `app.db`.

4.2 Operações CRUD com SQLAlchemy

Com o modelo de dados e a base de dados configurados, agora podemos realizar operações CRUD (Criar, Ler, Atualizar e Apagar) utilizando o SQLAlchemy.

As rotas que irão ser criadas serão:

- Create | POST /produtos - Adicionar um novo produto
- Read | GET /produtos - Obter a lista de produtos
- Update | PUT /produtos/{id} - Atualizar um produto existente
- Delete | DELETE /produtos/{id} - Apagar um produto

4.2.1 Criar um Produto (POST)

Para adicionar um novo produto à tabela, utilizamos a classe Produto para criar um objeto e adicionar à base de dados.

Exemplo de código:

```
@app.route('/produtos', methods=['POST'])
def add_produto():
    nome = request.json.get('nome')
    novo_produto = Produto(nome=nome)
    db.session.add(novo_produto)
    db.session.commit()
    return {"message": "Produto adicionado com sucesso!", "produto": {"nome":
    novo_produto.nome}}, 201
```

Não se esqueça de importar o request no início do ficheiro. Além disso é necessário adicionar no final do ficheiro o seguinte código:

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

Teste a rota POST /produtos utilizando o Postman ou o Httpie para adicionar um novo produto à base de dados.

Json a colocar no body do pedido:

```
{
    "nome": "Produto 1"
}
```

4.2.2 Ler Produtos (GET)

Para obter a lista de produtos da base de dados, podemos realizar uma consulta simples com SQLAlchemy.

Exemplo de código:

```
@app.route('/produtos', methods=['GET'])
def get_produtos():
    produtos = Produto.query.all()  # Obtem todos os produtos
    return {"produtos": [{"id": p.id, "nome": p.nome} for p in produtos]}
```

Teste a rota GET /produtos utilizando o Postman ou o Httpie para obter a lista de produtos da base de dados.

4.2.3 Atualizar um Produto (PUT)

Para atualizar um produto, primeiro procuramos o produto na base de dados e, de seguida, alteramos os seus atributos.

Exemplo de código:

```
@app.route('/produtos/<int:id>', methods=['PUT'])
def update_produto(id):
    produto = Produto.query.get_or_404(id)  # Obtem o produto com o id fornecido
    produto.nome = request.json.get('nome', produto.nome)  # Atualiza o nome, se
    fornecido
    db.session.commit()  # Guarda as alteracoes
    return {"message": "Produto atualizado com sucesso!", "produto": {"id": produto.id, "
    nome": produto.nome}}
```

Teste a rota `PUT /produtos/{id}` utilizando o `Postman` ou o `Httpie` para atualizar o nome de um produto existente.

`Json` a colocar no `body` da pedido:

```
{
    "nome": "Produto Atualizado"
}
```

4.2.4 Apagar um Produto (DELETE)

Para remover um produto da base de dados, utilizamos o método `delete()` do `SQLAlchemy`.

Exemplo de código:

```
@app.route('/produtos/<int:id>', methods=['DELETE'])
def delete_produto(id):
    produto = Produto.query.get_or_404(id)  # Obtem o produto com o id fornecido
    db.session.delete(produto)  # Remove o produto
    db.session.commit()  # Confirma a exclusao
    return {"message": f"Produto {id} apagado com sucesso!"}, 200
```

Teste a rota `DELETE /produtos/{id}` utilizando o `Postman` ou o `Httpie` para remover um produto da base de dados.

Bom trabalho!