🖤

# Flask

Tags



## Introduction

Flask is a web development framework for Python. With Flask, you can build and deploy web applications quickly and easily. Some of the things you can do with Flask include:

1. Create a web application using Python and Flask's built-in server
2. Define routes for your web application and bind them to functions that handle HTTP requests
3. Use the Jinja2 template engine to generate dynamic HTML pages
4. Use Flask-SQLAlchemy to interact with a database, and perform CRUD operations
5. Use Flask-WTForms to create and validate HTML forms

6. Use Flask-Login to add authentication and authorization to your web application
7. Use Flask-RESTful to build a RESTful API for your web application
8. Use Flask-SocketIO to add real-time features, such as chat or notifications
9. Use Flask-Testing to write unit tests for your Flask application
10. Use Flask-Security to quickly add common security features, such as password hashing and session management.

## Simple web app with Flask

To create a web application using Python and Flask's built-in server, you need to follow these steps:

1. Install Flask using the `pip` package manager: `pip install Flask`
2. Create a new Python file and import Flask at the top: `from flask import Flask`
3. Create a new Flask app by calling the `Flask` constructor: `app = Flask(__name__)`
4. Define a route for your web application by decorating a function with the `@app.route` decorator:

```
@app.route('/')
def hello():
  return 'Hello, World!'
```

1. Start the Flask development server by calling the `run` method on the app object, and specify the host and port to listen on:

```
if __name__ == '__main__':
  app.run(host='0.0.0.0', port=5000)
```

2. Run your Flask app using `python`: `python app.py`

Now you can visit your Flask app in a web browser at **http://0.0.0.0:5000**, and you should see the message "Hello, World!" displayed.

## Define routes with Flask

To define routes for your web application and bind them to functions that handle HTTP requests, you need to do the following:

1. Import the `Flask` class from the `flask` module
2. Create a new Flask app by calling the `Flask` constructor
3. Define a route by decorating a function with the `@app.route` decorator, and specify the URL path and the HTTP methods that the route will handle

For example, to define a route that responds to `GET` requests at the URL path `/products`, you can use the following code:

```
from flask import Flask

app = Flask(__name__)

@app.route('/products', methods=['GET'])
def get_products():
  # Code to handle the HTTP request goes here
```

The `get_products` function will be called whenever a `GET` request is sent to the `/products` URL. You can add code to the function to handle the request and generate a response.

You can also use the `@app.route` decorator to define multiple routes for the same URL path, but with different HTTP methods. For example, you can define a route that responds to `POST` requests at the `/products` URL like this:

```
@app.route('/products', methods=['POST'])
def create_product():
  # Code to handle the HTTP request goes here
```

In this case, the `create_product` function will be called whenever a `POST` request is sent to the `/products` URL, and you can add code to the function to handle the request and create a new product.

Here is an example of a `create_product` function that you can use in a Flask app:

```
@app.route('/products', methods=['POST'])
def create_product():
```

```
  # Get the product data from the request body
  data = request.get_json()

  # Create a new product object
  product = Product(
    name=data['name'],
    price=data['price'],
    quantity=data['quantity']
  )

  # Save the product to the database
  db.session.add(product)
  db.session.commit()

  # Return a JSON representation of the product
  return jsonify(product)
```

This function expects the product data to be sent in the request body as JSON, and uses it to create a new `Product` object. The product is then saved to the database using Flask-SQLAlchemy, and a JSON representation of the product is returned in the response.

Note that this code assumes that you have imported the `request`, `jsonify`, and `db` objects, and that you have defined a `Product` model using Flask-SQLAlchemy. You will need to add the necessary imports and model definition to your Flask app before you can use this code.

## Databases and CRUD with Flask

Here is an example of how you can use Flask-SQLAlchemy to interact with a database and perform CRUD (Create, Read, Update, Delete) operations:

```
# Import Flask-SQLAlchemy and create a new SQLAlchemy object
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

# Define a model class for the "products" table
class Product(db.Model):
  id = db.Column(db.Integer, primary_key=True)
  name = db.Column(db.String(255), nullable=False)
  price = db.Column(db.Float, nullable=False)
  quantity = db.Column(db.Integer, nullable=False)

# Create a new product
product = Product(
```

```
  name="Product 1",
  price=10.99,
  quantity=5
)

# Save the product to the database
db.session.add(product)
db.session.commit()

# Query for all products in the database
products = Product.query.all()

# Update a product
product = Product.query.get(1)
product.name = "Updated Product"
product.price = 15.99
db.session.commit()

# Delete a product
product = Product.query.get(1)
db.session.delete(product)
db.session.commit()
```

In this example, we define a `Product` model class that represents the "products" table in the database. We then use this model to create a new product, save it to the database, query for all products, update a product, and delete a product.

Note that this code assumes that you have initialized the `db` object and passed it to your Flask app's `create_app` function. You will need to add the necessary initialization code to your Flask app before you can use this code.

## RESTFUL with Flask

Here is an example of how you can use Flask-RESTful to build a RESTful API for your web application:

```
# Import Flask-RESTful and create a new Flask-RESTful API object
from flask_restful import Api, Resource

api = Api()

# Define a resource for the "products" endpoint
class ProductsResource(Resource):
  def get(self):
    # Query for all products in the database
```

```
    products = Product.query.all()

    # Return the products as a JSON list
    return [p.to_dict() for p in products]

  def post(self):
    # Get the product data from the request body
    data = request.get_json()

    # Create a new product object
    product = Product(
      name=data['name'],
      price=data['price'],
      quantity=data['quantity']
    )

    # Save the product to the database
    db.session.add(product)
    db.session.commit()

    # Return a JSON representation of the product
    return jsonify(product)

# Register the resource with the Flask-RESTful API
api.add_resource(ProductsResource, '/products')
```

In this example, we define a `ProductsResource` class that represents the "products" endpoint in our API. This class defines two methods, `get` and `post`, that handle HTTP `GET` and `POST` requests, respectively. The `get` method queries for all products in the database and returns them as a JSON list, while the `post` method creates a new product and saves it to the database.

Note that this code assumes that you have imported the `request`, `jsonify`, `Product`, and `db` objects, and that you have defined a `Product` model using Flask-SQLAlchemy. You will also need to add the necessary imports and model definition to your Flask app before you can use this code. Additionally, you will need to initialize the `api` object and pass it to your Flask app's `create_app` function before you can use it to register resources with your API.

## API with Flask

```
from flask import Flask, request
import requests

app = Flask(__name__)
```

```
@app.route('/weather')
def weather():
    # get the input parameters
    city = request.args.get('city')
    lang = request.args.get('lang')

    # validate the input parameters
    if not city:
        return 'Missing city parameter', 400
    if not lang:
        lang = 'en'

    # call the OpenWeatherMap API
    url = f'https://api.openweathermap.org/data/2.5/weather?q={city}&units=metric&lang={la
ng}&appid={API_KEY}'
    response = requests.get(url)
    data = response.json()

    # format the response
    return {
        'city': data['name'],
        'temp': f"{data['main']['temp']}°C",
        'condition': data['weather'][0]['description'],
        'forecast': data['weather'][0]['main'],
    }
```

## Better CRUD example with Flask

Here is an example of a more complicated API, built with the Flask framework. This API allows users to perform CRUD (create, read, update, delete) operations on a list of products, using RESTful routes and HTTP methods.

```
from flask import Flask, request, jsonify
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow
import os

# Create a Flask app, and configure it with the database URI and other settings
app = Flask(__name__)
basedir = os.path.abspath(os.path.dirname(__file__))
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' + os.path.join(basedir, 'products.d
b')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Create a SQLAlchemy database, and a Marshmallow serializer
db = SQLAlchemy(app)
ma = Marshmallow(app)
```

```
# Define the Product model, with its columns and data types
class Product(db.Model):
  id = db.Column(db.Integer, primary_key=True)
  name = db.Column(db.String(100), unique=True)
  price = db.Column(db.Float)

  def __init__(self, name, price):
    self.name = name
    self.price = price

# Define the Product schema, with its fields and data types
class ProductSchema(ma.Schema):
  class Meta:
    fields = ('id', 'name', 'price')

# Initialize the Product schema, and the Product API
product_schema = ProductSchema()
products_schema = ProductSchema(many=True)

# Define the RESTful routes and HTTP methods for the Product API
@app.route('/products', methods=['GET'])
def get_products():
  # Query all the products from the database
  products = Product.query.all()

  # Serialize the products, and return them as a JSON response
  result = products_schema.dump(products)
  return jsonify(result)

@app.route('/products/<int:id>', methods=['GET'])
def get_product(id):
  # Query the product with the specified id from the database
  product = Product.query.get(id)

  # Serialize the product, and return it as a JSON response
  result = product_schema.dump(product)
  return jsonify(result)

@app.route('/products', methods=['POST'])
def create_product():
  # Parse the request data, and extract the name and price of the product
  name = request.json['name']
  price = request.json['price']

  # Create a new product, with the specified name and price
  product = Product(name, price)

  # Add the product to the database, and commit the changes
  db.session.add(product)
  db.session.commit()

  # Serialize the product, and return it as a JSON response
  result = product_schema.dump(product)
  return jsonify(result)
```

```
@app.route('/products/<int:id>', methods=['PUT'])
def update_product(id):
  # Query the product with the specified id from the database
  product = Product.query.get(id)

  # Parse the request data, and extract the updated name and price of the product
  name = request.json['name']
  price = request.json['price']

  # Update the product's name and price with the specified values
  product.name = name
  product.price = price

  # Commit the changes to the database
  db.session.commit()

  # Serialize the product, and return it as a JSON response
  result = product_schema.dump(product)
  return jsonify(result)
```

## Full stack application

Here is a simple backend in Python using the Flask web framework:

```
from flask import Flask

# create a Flask app
app = Flask(__name__)

# define a route for the '/hello' endpoint
@app.route('/hello')
def hello():
    return 'Hello, World!'

# run the app
if __name__ == '__main__':
    app.run()
```

In this example, the Flask app is created using the `Flask` class and the `__name__` argument. Then, a route is defined for the `/hello` endpoint using the `@app.route` decorator. The route handler is a simple function that returns the string "Hello, World!". Finally, the app is run using the `app.run` method.

When you run this code, Flask will start a web server on your local machine and you can access the `/hello` endpoint by visiting **http://127.0.0.1:5000/hello** in your web

browser. You should see the "Hello, World!" message displayed in the browser.

This is a very simple example, but it shows the basics of how to create a backend in Python using Flask. You can add more routes and functionality to the app, such as data storage, processing, and security, to create more complex and useful backend applications.

## Connect HTML to this backend

Here is a simple frontend in HTML and JavaScript that connects to the Flask backend:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Flask Backend</title>
  </head>
  <body>
    <h1>Flask Backend</h1>
    <p>The message from the backend is: <span id="message"></span></p>
    <script>
      // make an AJAX request to the '/hello' endpoint
      const xhr = new XMLHttpRequest();
      xhr.open('GET', '/hello');
      xhr.onload = function() {
        if (xhr.status === 200) {
          // update the page with the response from the backend
          document.getElementById('message').innerHTML = xhr.responseText;
        } else {
          // handle error
        }
      };
      xhr.send();
    </script>
  </body>
</html>
```

In this example, the HTML page contains a `<span>` element with the `id` attribute set to "message". This element will be updated with the message from the backend. The JavaScript code uses the `XMLHttpRequest` object to make an AJAX request to the `/hello` endpoint of the Flask backend. When the request completes, the `onload` event handler is called and the response text is used to update the "message" element on the page.

You can run this HTML page in a web browser that is connected to the Flask backend. When you visit the page, the JavaScript code will make an AJAX request to the Flask backend and the message "Hello, World!" will be displayed on the page. This is a simple

example, but it shows how to connect a frontend to a backend using AJAX requests. You can add more functionality and interactivity to the frontend and backend to create more complex and useful applications.

## Connect React to this backend

Here is a simple React app that connects to the Flask backend:

```
import React, { useState, useEffect } from 'react';

function App() {
  // useState hook for storing the message from the backend
  const [message, setMessage] = useState('');

  // useEffect hook for making the AJAX request to the backend
  useEffect(() => {
    fetch('/hello')
      .then(response => response.text())
      .then(text => setMessage(text))
      .catch(error => console.error(error));
  }, []);

  return (
    <div>
      <h1>Flask Backend</h1>
      <p>The message from the backend is: {message}</p>
    </div>
  );
}

export default App;
```

In this example, the React app uses the `useState` and `useEffect` hooks to manage the state and side effects of the app. The `useState` hook is used to store the message from the Flask backend in the `message` state variable. The `useEffect` hook is used to make the AJAX request to the Flask backend using the `fetch` API. When the response is received, the `useEffect` hook updates the `message` state variable using the `setMessage` function.

The React app is rendered using a functional component with a `render` method that returns a JSX element. The JSX element displays the "Flask Backend" heading and the message from the Flask backend. When the app is run, the `useEffect` hook will make the AJAX request to the Flask backend and the message will be displayed on the page. This is a simple example, but it shows how to connect a React app to a backend using

AJAX requests. You can add more components, state, and functionality to the app to create more complex and useful applications.

# Flask + Open AI complete app

## Flask

Here is a simple Flask app that integrates with the OpenAI API to generate text using the GPT-3 model:

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

app = Flask(__name__)

# Set the OpenAI API key
openai.api_key = "YOUR_API_KEY"

@app.route("/generate-text", methods=["POST"])
def generate_text():
    # Get the prompt and number of tokens to generate from the request
    prompt = request.json["prompt"]
    num_tokens = request.json["num_tokens"]

    # Use the OpenAI API to generate text
    response = openai.Completion.create(
        engine="text-davinci-002",
        prompt=prompt,
        max_tokens=num_tokens,
        temperature=0.5,
    )

    # Return the generated text
    return jsonify({"generated_text": response["choices"][0]["text"]})

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

To use this app, you will need to replace `YOUR_API_KEY` with your own OpenAI API key. You can then use the `/generate-text` endpoint to generate text using the GPT-3 model. The endpoint accepts a JSON object containing a `prompt` and a `num_tokens` parameter, and returns the generated text.

For example, you could make a `POST` request to the endpoint with the following JSON object:

```
{
    "prompt": "The quick brown fox jumps over the lazy dog.",
    "num_tokens": 50
}
```

This would generate 50 tokens of text based on the provided prompt using the GPT-3 model. The response would be a JSON object containing the generated text, like this:

```
{
    "generated_text": "The quick brown fox jumps over the lazy dog. He runs into the fores
t and starts to explore. He finds a clearing with a beautiful stream and decides to take a
drink. As he is drinking, he hears a noise behind him and turns around to see a bear."
}
```

Note that this is just a simple example, and there are many other ways that you can integrate the OpenAI API into a Flask app depending on your specific requirements and use cases.

## Frontend - react

Here is an example of how you could connect the Flask app from the previous answer to a simple React frontend:

```
import React, { useState } from "react";
import axios from "axios";

function App() {
  const [prompt, setPrompt] = useState("");
  const [numTokens, setNumTokens] = useState(50);
  const [generatedText, setGeneratedText] = useState("");

  const handleSubmit = async e => {
    e.preventDefault();

    try {
      // Make a request to the Flask app to generate text
      const response = await axios.post("http://localhost:5000/generate-text", {
        prompt,
        num_tokens: numTokens
      });

      // Update the state with the generated text
      setGeneratedText(response.data.generated_text);
    } catch (error) {
```

```
      console.error(error);
    }
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <label htmlFor="prompt">Prompt:</label>
        <input
          type="text"
          id="prompt"
          value={prompt}
          onChange={e => setPrompt(e.target.value)}
        />
        <label htmlFor="numTokens">Number of Tokens:</label>
        <input
          type="number"
          id="numTokens"
          value={numTokens}
          onChange={e => setNumTokens(e.target.value)}
        />
        <button type="submit">Generate Text</button>
      </form>
      <p>Generated Text: {generatedText}</p>
    </div>
  );
}

export default App;
```

In this example, the React app has a form where the user can enter a prompt and the number of tokens to generate. When the form is submitted, the app makes a `POST` request to the Flask app's `/generate-text` endpoint with the prompt and number of tokens as parameters. The response from the Flask app, which contains the generated text, is then displayed in the React app.

Note that this is just a simple example, and there are many different ways that you can integrate a React frontend with a Flask backend depending on your specific requirements and use cases.

## Deploying the Flask app

1. Choose a hosting provider, such as Heroku, AWS, or Google Cloud Platform, and create an account with them.

2. Install the necessary dependencies and packages, such as Flask and any other libraries that your application uses.
3. Set up a local development environment and create a new Flask application.
4. Test your application locally to make sure it works as expected.
5. Create a `requirements.txt` file that lists all of the dependencies and packages that your application needs to run.
6. Create a `Procfile` that tells the hosting provider how to run your application.
7. Commit your code to a Git repository and push it to the hosting provider.
8. Configure the hosting provider to deploy your application when you push new code to your repository.
9. Test your application on the hosting provider's servers to make sure it is running correctly.
10. Update your DNS settings to point your domain name to the hosting provider's servers.
11. Access your application using your domain name and make sure it is working as expected.

## Resources

1. The official Flask documentation: **https://flask.palletsprojects.com/en/2.1.x/**
2. The Flask Mega-Tutorial by Miguel Grinberg: **https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world**
3. The Flask Web Development with Python Tutorial by Corey Schafer: **https://www.youtube.com/watch?v=MwZwr5Tvyxo&list=PL-osiE80TeTs4UjLw5MM6OjgkjFeUxCYH**
4. The Flask by Example series by Gareth Dwyer: **https://auth0.com/blog/flask-by-example-series-part-1-new-to-flask-should-i-stay-or-should-i-go/**
5. Flask Web Development with Python: Developing Web Applications with Python by Miguel Grinberg: **https://www.amazon.com/Flask-Web-Development-Python-applications/dp/1491991739**

6. Flask Web Development Cookbook by Shalabh Aggarwal: **https://www.amazon.com/Flask-Web-Development-Cookbook-efficient/dp/1788834386**
7. Flask Framework Cookbook by Shalabh Aggarwal: **https://www.amazon.com/Flask-Framework-Cookbook-Shalabh-Aggarwal/dp/1789130316**
8. Flask for Beginners by Gareth Dwyer: **https://flaskforbeginners.com/**
9. The Flask API Development Fundamentals course on Pluralsight: **https://www.pluralsight.com/courses/flask-api-development-fundamentals**
10. The Flask for Full Stack Web Development course on LinkedIn Learning: **https://www.linkedin.com/learning/flask-for-full-stack-web-development/**
11. The Flask: Building Python Web Services course on Udemy: **https://www.udemy.com/course/flask-python/**
12. The Flask by Example series on Auth0's blog: **https://auth0.com/blog/flask-by-example-series-part-1-new-to-flask-should-i-stay-or-should-i-go/**

# Videos

1. Flask Web Development with Python Tutorial by Corey Schafer: **https://www.youtube.com/watch?v=MwZwr5Tvyxo&list=PL-osiE80TeTs4UjLw5MM6OjgkjFeUxCYH**
2. Flask Tutorial by Traversy Media: **https://www.youtube.com/watch?v=Z1RJmh_OqeA&list=PLillGF-RfqbbbPz6GSEM9hLQObuQjNoj_**
3. Flask REST API Tutorial by Pretty Printed: **https://www.youtube.com/watch?v=s_ht4AKnWZg&list=PL-osiE80TeTs4UjLw5MM6OjgkjFeUxCYH**
4. Flask for Python by Derek Banas: **https://www.youtube.com/watch?v=MwZwr5Tvyxo**
5. Building Web Applications with Flask by Derek Banas: **https://www.youtube.com/watch?v=zRwy8gtgJ1A**
6. Flask Tutorial #1 - How to Make Websites with Python by Kalle Hallden: **https://www.youtube.com/watch?v=tJxcKyFMTGo&t=1065s**

7. Flask and Python by FreeCodeCamp.org: **https://www.youtube.com/watch?v=ZVGwqnjOKjk**
8. Flask CRUD with MySQL by Traversy Media: **https://www.youtube.com/watch?v=5JnMutdy6Fw**
9. Flask Basics by Dev Ed: **https://www.youtube.com/watch?v=MwZwr5Tvyxo**
10. Flask Tutorial #2 - Templates by Kalle Hallden: **https://www.youtube.com/watch?v=QnDWIZuWYW0&t=1245s**

## Build these projects

1. A simple to-do list application that allows users to add, view, and mark tasks as completed.
2. A blog or content management system (CMS) that allows users to create, edit, and publish posts and pages.
3. A social networking platform that allows users to create profiles, connect with friends, and share updates and photos.
4. An e-commerce platform that allows users to browse and purchase products, and includes features such as a shopping cart and checkout process.
5. A task management or project management application that allows users to create and assign tasks to team members, track progress, and collaborate on projects.
6. A messaging or chat application that allows users to communicate with each other in real-time.
7. A booking or reservation system that allows users to search for and book hotels, flights, or other services.
8. A gaming or trivia platform that allows users to compete against each other in various games or quizzes.
9. A financial or budgeting application that helps users track their income, expenses, and savings.
10. A data visualization or dashboard application that allows users to view and analyze data from various sources.