Python to Flask REST API - Complete Guide

Table of Contents

Python Refresher

Basic Syntax and Variables

python

```python
# Variables and basic types
name = "John"
age = 25
height = 5.9
is_student = True

# String formatting
message = f"Hello {name}, you are {age} years old"
print(message)
```

Data Structures

python

```python
# Lists (mutable, ordered)
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
print(fruits[0])  # apple

# Dictionaries (key-value pairs)
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}
person["email"] = "alice@email.com"

# Tuples (immutable, ordered)
coordinates = (10, 20)
```

Functions

python

```python
def greet(name, age=None):
    if age:
        return f"Hello {name}, you are {age} years old"
    return f"Hello {name}"

# Lambda functions
square = lambda x: x ** 2
print(square(5))  # 25

# List comprehensions
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
```

Classes and Objects

python

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        return f"Hi, I'm {self.name} and I'm {self.age} years old"

    @property
    def adult(self):
        return self.age >= 18

# Usage
person = Person("Bob", 25)
print(person.introduce())
print(person.adult)  # True
```

Error Handling

python

```python
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
finally:
    print("Cleanup code here")
```

Modules and Packages

```python
# Importing modules
import json
from datetime import datetime
import requests as req

# Creating a module (utils.py)
def format_response(data, status="success"):
    return {
        "status": status,
        "data": data,
        "timestamp": datetime.now().isoformat()
    }
```

Introduction to Flask

What is Flask?

Flask is a lightweight, flexible Python web framework that provides the basic tools and libraries for building web applications and REST APIs.

Installation and Setup

```bash
pip install flask flask-cors python-dotenv
```

Basic Flask Application

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

app = Flask(__name__)

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

@app.route('/api/health')
def health_check():
    return jsonify({"status": "healthy", "message": "API is running"})

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

Flask Application Structure

```
my_flask_app/
│
├── app.py              # Main application file
├── config.py           # Configuration settings
├── requirements.txt    # Dependencies
├── .env               # Environment variables
│
├── models/            # Database models
│   └── __init__.py
│
├── routes/            # Route handlers
│   ├── __init__.py
│   ├── auth.py
│   └── users.py
│
├── utils/             # Utility functions
│   └── __init__.py
│
└── tests/             # Test files
    └── test_app.py
```

Building REST APIs with Flask

HTTP Methods and Routes

python

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

app = Flask(__name__)

# Sample data
users = [
    {"id": 1, "name": "John Doe", "email": "john@email.com"},
    {"id": 2, "name": "Jane Smith", "email": "jane@email.com"}
]

# GET - Retrieve all users
@app.route('/api/users', methods=['GET'])
def get_users():
    return jsonify({
        "success": True,
        "data": users,
        "count": len(users)
    })

# GET - Retrieve single user
@app.route('/api/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = next((u for u in users if u["id"] == user_id), None)
    if not user:
        return jsonify({"success": False, "message": "User not found"}), 404
    return jsonify({"success": True, "data": user})

# POST - Create new user
@app.route('/api/users', methods=['POST'])
def create_user():
    data = request.get_json()

    # Validation
    if not data or not data.get('name') or not data.get('email'):
        return jsonify({
            "success": False,
            "message": "Name and email are required"
        }), 400

    new_user = {
        "id": len(users) + 1,
        "name": data['name'],
        "email": data['email']
    }
    users.append(new_user)
    return jsonify({
        "success": True,
        "message": "User created successfully",
        "data": new_user
    }), 201

# PUT - Update user
@app.route('/api/users/<int:user_id>', methods=['PUT'])
def update_user(user_id):
    user = next((u for u in users if u["id"] == user_id), None)
    if not user:
        return jsonify({"success": False, "message": "User not found"}), 404

    data = request.get_json()
    user.update({
        "name": data.get('name', user['name']),
        "email": data.get('email', user['email'])
    })

    return jsonify({
        "success": True,
        "message": "User updated successfully",
        "data": user
    })

# DELETE - Delete user
@app.route('/api/users/<int:user_id>', methods=['DELETE'])
def delete_user(user_id):
    global users
    users = [u for u in users if u["id"] != user_id]
    return jsonify({
        "success": True,
        "message": "User deleted successfully"
    })
```

Request Handling and Validation

python

```python
from flask import request, jsonify
import re

def validate_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return re.match(pattern, email) is not None

@app.route('/api/users', methods=['POST'])
def create_user_with_validation():
    try:
        data = request.get_json()

        # Check if JSON data exists
        if not data:
            return jsonify({
                "success": False,
                "message": "No JSON data provided"
            }), 400

        # Required fields validation
        required_fields = ['name', 'email']
        for field in required_fields:
            if field not in data or not data[field].strip():
                return jsonify({
                    "success": False,
                    "message": f"{field.capitalize()} is required"
                }), 400

        # Email validation
        if not validate_email(data['email']):
            return jsonify({
                "success": False,
                "message": "Invalid email format"
            }), 400

        # Check if email already exists
        if any(u['email'] == data['email'] for u in users):
            return jsonify({
                "success": False,
                "message": "Email already exists"
            }), 409

        # Create user
        new_user = {
            "id": max([u['id'] for u in users], default=0) + 1,
            "name": data['name'].strip(),
            "email": data['email'].strip().lower()
        }
        users.append(new_user)

        return jsonify({
            "success": True,
            "message": "User created successfully",
            "data": new_user
        }), 201

    except Exception as e:
        return jsonify({
            "success": False,
            "message": "Internal server error"
        }), 500
```

Error Handling

python

```python
from flask import Flask, jsonify

app = Flask(__name__)

# Custom error handlers
@app.errorhandler(404)
def not_found(error):
    return jsonify({
        "success": False,
        "message": "Resource not found",
        "error": "404 Not Found"
    }), 404

@app.errorhandler(400)
def bad_request(error):
    return jsonify({
        "success": False,
        "message": "Bad request",
        "error": "400 Bad Request"
    }), 400

@app.errorhandler(500)
def internal_error(error):
    return jsonify({
        "success": False,
        "message": "Internal server error",
        "error": "500 Internal Server Error"
    }), 500

# Global exception handler
@app.errorhandler(Exception)
def handle_exception(e):
    return jsonify({
        "success": False,
        "message": "An unexpected error occurred",
        "error": str(e)
    }), 500
```

Advanced Flask Features

Configuration Management

python

```python
# config.py
import os
from dotenv import load_dotenv

load_dotenv()

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'dev-secret-key'
    DEBUG = os.environ.get('DEBUG', 'False').lower() == 'true'
    PORT = int(os.environ.get('PORT', 5000))

class DevelopmentConfig(Config):
    DEBUG = True
    DATABASE_URL = 'sqlite:///dev.db'

class ProductionConfig(Config):
    DEBUG = False
    DATABASE_URL = os.environ.get('DATABASE_URL')

config = {
    'development': DevelopmentConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig
}

# app.py
from config import config
import os

app = Flask(__name__)
env = os.environ.get('FLASK_ENV', 'default')
app.config.from_object(config[env])
```

Middleware and Decorators

python

```python
from functools import wraps
from flask import request, jsonify
import time

# Logging middleware
@app.before_request
def log_request():
    print(f"{request.method} {request.path} - {request.remote_addr}")

@app.after_request
def log_response(response):
    print(f"Response: {response.status_code}")
    return response

# Rate limiting decorator
request_counts = {}

def rate_limit(max_requests=100, window=3600):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            client_ip = request.remote_addr
            current_time = time.time()

            if client_ip not in request_counts:
                request_counts[client_ip] = []

            # Remove old requests outside the window
            request_counts[client_ip] = [
                req_time for req_time in request_counts[client_ip]
                if current_time - req_time < window
            ]

            if len(request_counts[client_ip]) >= max_requests:
                return jsonify({
                    "success": False,
                    "message": "Rate limit exceeded"
                }), 429

            request_counts[client_ip].append(current_time)
            return f(*args, **kwargs)
        return decorated_function
    return decorator

# Usage
@app.route('/api/users')
```

```
@rate_limit(max_requests=50, window=3600)  # 50 requests per hour
def get_users():
    return jsonify({"data": users})
```

CORS (Cross-Origin Resource Sharing)

```python
from flask_cors import CORS

app = Flask(__name__)

# Enable CORS for all routes
CORS(app)

# Or configure CORS with specific options
CORS(app, resources={
    r"/api/*": {
        "origins": ["http://localhost:3000", "https://myapp.com"],
        "methods": ["GET", "POST", "PUT", "DELETE"],
        "allow_headers": ["Content-Type", "Authorization"]
    }
})
```

Flask vs Express.js Comparison

Basic Server Setup

Flask (Python)

```python
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/')
def home():
    return jsonify({"message": "Hello World"})

if __name__ == '__main__':
    app.run(debug=True, port=5000)
```

Express.js (Node.js)

```javascript
const express = require('express');
const app = express();

app.use(express.json());

app.get('/', (req, res) => {
    res.json({ message: "Hello World" });
});

app.listen(5000, () => {
    console.log('Server running on port 5000');
});
```

Route Definitions

Flask

```python
@app.route('/api/users', methods=['GET'])
def get_users():
    return jsonify(users)

@app.route('/api/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = find_user(user_id)
    return jsonify(user)

@app.route('/api/users', methods=['POST'])
def create_user():
    data = request.get_json()
    new_user = create_new_user(data)
    return jsonify(new_user), 201
```

Express.js

```javascript
app.get('/api/users', (req, res) => {
    res.json(users);
});

app.get('/api/users/:userId', (req, res) => {
    const user = findUser(req.params.userId);
    res.json(user);
});

app.post('/api/users', (req, res) => {
    const newUser = createNewUser(req.body);
    res.status(201).json(newUser);
});
```

Middleware

Flask

```python
from functools import wraps

def auth_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        token = request.headers.get('Authorization')
        if not token:
            return jsonify({'error': 'No token provided'}), 401
        return f(*args, **kwargs)
    return decorated_function

@app.route('/api/protected')
@auth_required
def protected_route():
    return jsonify({'message': 'Access granted'})

# Global middleware
@app.before_request
def before_request():
    print(f"Request: {request.method} {request.path}")
```

Express.js

```javascript
// Middleware function
const authRequired = (req, res, next) => {
    const token = req.headers.authorization;
    if (!token) {
        return res.status(401).json({ error: 'No token provided' });
    }
    next();
};

app.get('/api/protected', authRequired, (req, res) => {
    res.json({ message: 'Access granted' });
});

// Global middleware
app.use((req, res, next) => {
    console.log(`Request: ${req.method} ${req.path}`);
    next();
});
```

Error Handling

Flask

```python
@app.errorhandler(404)
def not_found(error):
    return jsonify({'error': 'Not found'}), 404

@app.errorhandler(500)
def internal_error(error):
    return jsonify({'error': 'Internal server error'}), 500

# Try-catch in routes
@app.route('/api/users')
def get_users():
    try:
        users = fetch_users()
        return jsonify(users)
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Express.js

javascript

```javascript
// Error handling middleware
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).json({ error: 'Internal server error' });
});

// 404 handler
app.use((req, res) => {
    res.status(404).json({ error: 'Not found' });
});

// Try-catch in routes
app.get('/api/users', async (req, res, next) => {
    try {
        const users = await fetchUsers();
        res.json(users);
    } catch (error) {
        next(error);
    }
});
```

Request Handling

Flask

python

```python
from flask import request

@app.route('/api/users', methods=['POST'])
def create_user():
    # JSON body
    data = request.get_json()

    # Query parameters
    page = request.args.get('page', 1, type=int)

    # Headers
    auth_header = request.headers.get('Authorization')

    # Form data
    username = request.form.get('username')

    return jsonify({'received': data})
```

Express.js

```javascript
app.post('/api/users', (req, res) => {
    // JSON body (with express.json() middleware)
    const data = req.body;

    // Query parameters
    const page = parseInt(req.query.page) || 1;

    // Headers
    const authHeader = req.headers.authorization;

    // Form data (with express.urlencoded() middleware)
    const username = req.body.username;

    res.json({ received: data });
});
```

Key Differences Summary

Feature	Flask	Express.js
Language	Python	JavaScript/Node.js
Philosophy	Minimalist, explicit	Minimalist, flexible
Decorators	Built-in decorator support	Function-based middleware
Async Support	Limited (requires additional setup)	Native async/await support
Route Parameters	`<int:id>` type conversion	`:id` string parameters
Request Object	Global `request` object	Passed as parameter
Response	Return values auto-converted	Explicit `res.json()` calls
Error Handling	Decorator-based error handlers	Middleware-based error handling
Community	Smaller but mature ecosystem	Large, active ecosystem
Performance	Good for I/O bound tasks	Excellent for concurrent requests

When to Choose Flask vs Express.js

Choose Flask when:

- You're comfortable with Python
- Building data-heavy applications
- Need integration with Python libraries (ML, data science)
- Prefer explicit, readable code structure

- Working with existing Python infrastructure

Choose Express.js when:

- You need high concurrency
- Building real-time applications
- Team is familiar with JavaScript
- Need rapid prototyping
- Want a large ecosystem of npm packages

Conclusion

This guide covered Python fundamentals, Flask web framework, REST API development, and a comprehensive comparison with Express.js. Flask provides a clean, Pythonic way to build APIs with excellent flexibility and simplicity, while Express.js offers superior performance for high-concurrency applications. Choose based on your team's expertise, project requirements, and existing technology stack.

Next Steps

1. Practice building a complete CRUD API with Flask
2. Explore Flask extensions like Flask-SQLAlchemy for databases
3. Learn about Flask-JWT-Extended for authentication
4. Consider Flask-RESTful for more structured API development
5. Deploy your Flask applications using Gunicorn and Docker