Complete Guide: Python, Flask, REST APIs & HTTP Server from Scratch

Table of Contents

Python Fundamentals

Key Concepts

Python is a high-level, interpreted programming language known for its simplicity and readability. It's particularly popular for web development due to its extensive ecosystem of frameworks and libraries.

Virtual Environments

Before starting any Python project, create a virtual environment to manage dependencies:

bash

```bash
# Create virtual environment
python -m venv myproject_env

# Activate (Windows)
myproject_env\Scripts\activate

# Activate (Linux/Mac)
source myproject_env/bin/activate

# Install packages
pip install flask requests

# Create requirements file
pip freeze > requirements.txt
```

Essential Python for Web Development

python

```
# Data structures
users = [{"id": 1, "name": "John"}, {"id": 2, "name": "Jane"}]
user_dict = {"1": "John", "2": "Jane"}

# Functions
def get_user_by_id(user_id):
    return next((user for user in users if user["id"] == user_id), None)

# Classes
class User:
    def __init__(self, user_id, name):
        self.id = user_id
        self.name = name

    def to_dict(self):
        return {"id": self.id, "name": self.name}

# Exception handling
try:
    result = some_operation()
except ValueError as e:
    print(f"Error: {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
```

Flask Framework

What is Flask?

Flask is a lightweight, micro web framework for Python. It provides the basic tools and libraries needed to build web applications without imposing a specific project structure.

Core Components

1. Application Factory Pattern

```
python
```

```python
from flask import Flask

def create_app():
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'your-secret-key'

    # Register blueprints
    from .routes import main
    app.register_blueprint(main)

    return app
```

2. Routing

python

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

app = Flask(__name__)

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

@app.route('/user/<int:user_id>')
def get_user(user_id):
    return f"User ID: {user_id}"

@app.route('/api/users', methods=['GET', 'POST'])
def users():
    if request.method == 'GET':
        return jsonify({"users": []})
    elif request.method == 'POST':
        data = request.get_json()
        return jsonify({"message": "User created", "data": data})
```

3. Request and Response Handling

python

```
from flask import request, jsonify, make_response

@app.route('/api/data', methods=['POST'])
def handle_data():
    # Get JSON data
    json_data = request.get_json()

    # Get form data
    form_data = request.form.get('key')

    # Get query parameters
    query_param = request.args.get('param')

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

    # Create response
    response = make_response(jsonify({"status": "success"}))
    response.headers['Content-Type'] = 'application/json'
    response.status_code = 201

    return response
```

Building a Flask Project

Project Structure

```
myproject/
├── app/
│   ├── __init__.py
│   ├── models.py
│   ├── routes.py
│   └── utils.py
├── config.py
├── requirements.txt
├── run.py
└── README.md
```

Step-by-Step Project Setup

1. Create Project Structure

```
bash
```

```
mkdir myproject
cd myproject
mkdir app
touch app/__init__.py app/models.py app/routes.py app/utils.py
touch config.py run.py requirements.txt
```

2. Configuration (config.py)

```python
import os

class Config:
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'dev-secret-key'
    DATABASE_URL = os.environ.get('DATABASE_URL') or 'sqlite:///app.db'
    DEBUG = False
    TESTING = False

class DevelopmentConfig(Config):
    DEBUG = True

class ProductionConfig(Config):
    DEBUG = False

class TestingConfig(Config):
    TESTING = True
    DATABASE_URL = 'sqlite:///test.db'

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

3. Application Factory (app/**init**.py)

```python
```

```python
from flask import Flask
from config import config

def create_app(config_name='default'):
    app = Flask(__name__)
    app.config.from_object(config[config_name])

    # Register blueprints
    from .routes import api_bp
    app.register_blueprint(api_bp, url_prefix='/api')

    return app
```

4. Models (app/models.py)

python

```python
from dataclasses import dataclass
from typing import List, Optional
import json

@dataclass
class User:
    id: int
    name: str
    email: str

    def to_dict(self):
        return {
            'id': self.id,
            'name': self.name,
            'email': self.email
        }

class UserRepository:
    def __init__(self):
        self.users = []
        self.next_id = 1

    def create_user(self, name: str, email: str) -> User:
        user = User(self.next_id, name, email)
        self.users.append(user)
        self.next_id += 1
        return user

    def get_user(self, user_id: int) -> Optional[User]:
        return next((user for user in self.users if user.id == user_id), None)

    def get_all_users(self) -> List[User]:
        return self.users

    def update_user(self, user_id: int, name: str = None, email: str = None) -> Optional[User]:
        user = self.get_user(user_id)
        if user:
            if name:
                user.name = name
            if email:
                user.email = email
        return user

    def delete_user(self, user_id: int) -> bool:
        user = self.get_user(user_id)
        if user:
            self.users.remove(user)
            return True
        return False

# Global repository instance
user_repo = UserRepository()
```

5. Routes (app/routes.py)

python

```python
from flask import Blueprint, request, jsonify
from .models import user_repo
from .utils import validate_email

api_bp = Blueprint('api', __name__)

@api_bp.route('/users', methods=['GET'])
def get_users():
    users = user_repo.get_all_users()
    return jsonify([user.to_dict() for user in users])

@api_bp.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = user_repo.get_user(user_id)
    if user:
        return jsonify(user.to_dict())
    return jsonify({'error': 'User not found'}), 404

@api_bp.route('/users', methods=['POST'])
def create_user():
    data = request.get_json()

    if not data or 'name' not in data or 'email' not in data:
        return jsonify({'error': 'Name and email required'}), 400

    if not validate_email(data['email']):
        return jsonify({'error': 'Invalid email format'}), 400

    user = user_repo.create_user(data['name'], data['email'])
    return jsonify(user.to_dict()), 201

@api_bp.route('/users/<int:user_id>', methods=['PUT'])
def update_user(user_id):
    data = request.get_json()

    if not data:
        return jsonify({'error': 'No data provided'}), 400

    email = data.get('email')
    if email and not validate_email(email):
        return jsonify({'error': 'Invalid email format'}), 400

    user = user_repo.update_user(user_id, data.get('name'), email)
    if user:
        return jsonify(user.to_dict())
    return jsonify({'error': 'User not found'}), 404
```

```python
@api_bp.route('/users/<int:user_id>', methods=['DELETE'])
def delete_user(user_id):
    if user_repo.delete_user(user_id):
        return '', 204
    return jsonify({'error': 'User not found'}), 404

@api_bp.errorhandler(404)
def not_found(error):
    return jsonify({'error': 'Endpoint not found'}), 404

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

6. Utilities (app/utils.py)

python

```
import re
from functools import wraps
from flask import request, jsonify

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

def require_json(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if not request.is_json:
            return jsonify({'error': 'Content-Type must be application/json'}), 400
        return f(*args, **kwargs)
    return decorated_function

def validate_required_fields(required_fields):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            data = request.get_json()
            missing_fields = [field for field in required_fields if field not in data]
            if missing_fields:
                return jsonify({
                    'error': f'Missing required fields: {", ".join(missing_fields)}'
                }), 400
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

7. Application Entry Point (run.py)

```python
import os
from app import create_app

app = create_app(os.environ.get('FLASK_ENV', 'development'))

if __name__ == '__main__':
    app.run(
        host='0.0.0.0',
        port=int(os.environ.get('PORT', 5000)),
        debug=app.config['DEBUG']
    )
```

Running the Application

```bash
# Set environment variables
export FLASK_APP=run.py
export FLASK_ENV=development

# Run the application
python run.py

# Or using Flask CLI
flask run
```

REST API Configuration

REST Principles

REST (Representational State Transfer) is an architectural style for designing web services. Key principles include:

1. **Stateless**: Each request contains all necessary information
2. **Resource-based**: URLs represent resources
3. **HTTP methods**: Use appropriate HTTP verbs (GET, POST, PUT, DELETE)
4. **JSON format**: Use JSON for data exchange
5. **Status codes**: Return appropriate HTTP status codes

Advanced REST API Features

1. Pagination

```python
```

```python
@api_bp.route('/users', methods=['GET'])
def get_users():
    page = request.args.get('page', 1, type=int)
    per_page = request.args.get('per_page', 10, type=int)

    users = user_repo.get_all_users()
    total = len(users)

    start = (page - 1) * per_page
    end = start + per_page
    paginated_users = users[start:end]

    return jsonify({
        'users': [user.to_dict() for user in paginated_users],
        'pagination': {
            'page': page,
            'per_page': per_page,
            'total': total,
            'pages': (total + per_page - 1) // per_page
        }
    })
```

2. Filtering and Searching

python

```python
@api_bp.route('/users/search', methods=['GET'])
def search_users():
    query = request.args.get('q', '')
    email_filter = request.args.get('email', '')

    users = user_repo.get_all_users()

    if query:
        users = [user for user in users if query.lower() in user.name.lower()]

    if email_filter:
        users = [user for user in users if email_filter.lower() in user.email.lower()]

    return jsonify([user.to_dict() for user in users])
```

3. Content Negotiation

python

```python
from flask import request, jsonify, make_response
import xml.etree.ElementTree as ET

@api_bp.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = user_repo.get_user(user_id)
    if not user:
        return jsonify({'error': 'User not found'}), 404

    # Check Accept header
    accept_header = request.headers.get('Accept', 'application/json')

    if 'application/xml' in accept_header:
        # Return XML response
        root = ET.Element('user')
        ET.SubElement(root, 'id').text = str(user.id)
        ET.SubElement(root, 'name').text = user.name
        ET.SubElement(root, 'email').text = user.email

        response = make_response(ET.tostring(root, encoding='unicode'))
        response.headers['Content-Type'] = 'application/xml'
        return response

    # Default to JSON
    return jsonify(user.to_dict())
```

4. Rate Limiting

```
python
```

```python
from functools import wraps
from time import time

# Simple in-memory rate limiter
request_counts = {}

def rate_limit(max_requests=10, window=60):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            client_ip = request.remote_addr
            current_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({'error': 'Rate limit exceeded'}), 429

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

@api_bp.route('/users', methods=['POST'])
@rate_limit(max_requests=5, window=60)
def create_user():
    # User creation logic here
    pass
```

HTTP Server from Scratch

Basic HTTP Server Implementation

python

```python
import socket
import threading
from urllib.parse import urlparse, parse_qs
import json
from datetime import datetime

class HTTPServer:
    def __init__(self, host='localhost', port=8080):
        self.host = host
        self.port = port
        self.routes = {}
        self.middleware = []

    def route(self, path, method='GET'):
        """Decorator to register routes"""
        def decorator(handler):
            if path not in self.routes:
                self.routes[path] = {}
            self.routes[path][method.upper()] = handler
            return handler
        return decorator

    def add_middleware(self, middleware_func):
        """Add middleware function"""
        self.middleware.append(middleware_func)

    def parse_request(self, request_data):
        """Parse HTTP request"""
        lines = request_data.decode('utf-8').split('\r\n')
        request_line = lines[0]

        method, path, version = request_line.split(' ')

        # Parse headers
        headers = {}
        body_start = 0
        for i, line in enumerate(lines[1:], 1):
            if line == '':
                body_start = i + 1
                break
            key, value = line.split(':', 1)
            headers[key.strip()] = value.strip()

        # Parse body
        body = '\r\n'.join(lines[body_start:]) if body_start < len(lines) else ''
        # Parse query parameters
        parsed_url = urlparse(path)
        query_params = parse_qs(parsed_url.query)

        return {
            'method': method,
            'path': parsed_url.path,
            'query_params': query_params,
            'headers': headers,
            'body': body,
            'version': version
        }

    def create_response(self, status_code, headers, body):
        """Create HTTP response"""
        status_messages = {
            200: 'OK',
            201: 'Created',
            400: 'Bad Request',
            404: 'Not Found',
            405: 'Method Not Allowed',
            500: 'Internal Server Error'
        }

        status_message = status_messages.get(status_code, 'Unknown')
        response = f'HTTP/1.1 {status_code} {status_message}\r\n'

        # Add default headers
        default_headers = {
            'Server': 'Custom-HTTP-Server/1.0',
            'Date': datetime.utcnow().strftime('%a, %d %b %Y %H:%M:%S GMT'),
            'Connection': 'close'
        }

        all_headers = {**default_headers, **headers}

        for key, value in all_headers.items():
            response += f'{key}: {value}\r\n'

        response += '\r\n'
        response += body

        return response.encode('utf-8')

    def handle_request(self, request):
        """Handle incoming request"""
        try:
            # Apply middleware
            for middleware in self.middleware:
                result = middleware(request)
                if result:  # Middleware can modify request or return early response
                    return result

            path = request['path']
            method = request['method']

            # Find matching route
            if path in self.routes and method in self.routes[path]:
                handler = self.routes[path][method]
                return handler(request)
            else:
                # Check for parameterized routes
                for route_path in self.routes:
                    if self.match_route(route_path, path):
                        if method in self.routes[route_path]:
                            handler = self.routes[route_path][method]
                            # Extract parameters
                            params = self.extract_params(route_path, path)
                            request['params'] = params
                            return handler(request)

                return self.create_response(
                    404,
                    {'Content-Type': 'application/json'},
                    json.dumps({'error': 'Route not found'})
                )

        except Exception as e:
            return self.create_response(
                500,
                {'Content-Type': 'application/json'},
                json.dumps({'error': 'Internal server error', 'message': str(e)})
            )

    def match_route(self, route_pattern, request_path):
        """Match route pattern with request path"""
        route_parts = route_pattern.strip('/').split('/')
        path_parts = request_path.strip('/').split('/')

        if len(route_parts) != len(path_parts):
            return False

        for route_part, path_part in zip(route_parts, path_parts):
            if route_part.startswith('<') and route_part.endswith('>'):
                continue  # Parameter placeholder
            elif route_part != path_part:
                return False

        return True

    def extract_params(self, route_pattern, request_path):
        """Extract parameters from route"""
        route_parts = route_pattern.strip('/').split('/')
        path_parts = request_path.strip('/').split('/')

        params = {}
        for route_part, path_part in zip(route_parts, path_parts):
            if route_part.startswith('<') and route_part.endswith('>'):
                param_name = route_part[1:-1]
                # Handle type conversion
                if ':' in param_name:
                    param_type, param_name = param_name.split(':', 1)
                    if param_type == 'int':
                        path_part = int(path_part)
                params[param_name] = path_part

        return params

    def handle_client(self, client_socket, address):
        """Handle individual client connection"""
        try:
            request_data = client_socket.recv(4096)
            if not request_data:
                return

            request = self.parse_request(request_data)
            response = self.handle_request(request)
            client_socket.send(response)

        except Exception as e:
            error_response = self.create_response(
                500,
                {'Content-Type': 'text/plain'},
                f'Server Error: {str(e)}'
            )
            client_socket.send(error_response)

        finally:
            client_socket.close()

    def start(self):
        """Start the HTTP server"""
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        server_socket.bind((self.host, self.port))
        server_socket.listen(5)

        print(f'Server running on http://{self.host}:{self.port}')

        try:
            while True:
                client_socket, address = server_socket.accept()
                client_thread = threading.Thread(
                    target=self.handle_client,
                    args=(client_socket, address)
                )
                client_thread.start()

        except KeyboardInterrupt:
            print('\nShutting down server...')

        finally:
            server_socket.close()

# Usage Example
def create_api_server():
    server = HTTPServer('localhost', 8080)

    # In-memory data store
    users = []
    next_id = 1

    # Middleware for logging
    def logging_middleware(request):
        print(f"{datetime.now()} - {request['method']} {request['path']}")
        return None  # Continue processing

    # Middleware for CORS
    def cors_middleware(request):
        return None  # Could add CORS headers here

    server.add_middleware(logging_middleware)
    server.add_middleware(cors_middleware)

    @server.route('/', 'GET')
    def home(request):
        return server.create_response(
            200,
        {'Content-Type': 'application/json'},
        json.dumps({'message': 'Welcome to Custom HTTP Server API'})
    )

@server.route('/api/users', 'GET')
def get_users(request):
    return server.create_response(
        200,
        {'Content-Type': 'application/json'},
        json.dumps(users)
    )

@server.route('/api/users', 'POST')
def create_user(request):
    nonlocal next_id

    try:
        data = json.loads(request['body'])
        user = {
            'id': next_id,
            'name': data.get('name'),
            'email': data.get('email')
        }
        users.append(user)
        next_id += 1

        return server.create_response(
            201,
            {'Content-Type': 'application/json'},
            json.dumps(user)
        )

    except json.JSONDecodeError:
        return server.create_response(
            400,
            {'Content-Type': 'application/json'},
            json.dumps({'error': 'Invalid JSON'})
        )

@server.route('/api/users/<int:user_id>', 'GET')
def get_user(request):
    user_id = request['params']['user_id']
    user = next((u for u in users if u['id'] == user_id), None)

    if user:
        return server.create_response(
            200,
                {'Content-Type': 'application/json'},
                json.dumps(user)
            )
        else:
            return server.create_response(
                404,
                {'Content-Type': 'application/json'},
                json.dumps({'error': 'User not found'})
            )

    @server.route('/api/users/<int:user_id>', 'DELETE')
    def delete_user(request):
        user_id = request['params']['user_id']
        user_index = next((i for i, u in enumerate(users) if u['id'] == user_id), None)

        if user_index is not None:
            users.pop(user_index)
            return server.create_response(204, {}, '')
        else:
            return server.create_response(
                404,
                {'Content-Type': 'application/json'},
                json.dumps({'error': 'User not found'})
            )

    return server

# Run the server
if __name__ == '__main__':
    server = create_api_server()
    server.start()
```

Testing the Custom HTTP Server

python

```
# test_server.py
import requests
import json

def test_api():
    base_url = 'http://localhost:8080'

    # Test home endpoint
    response = requests.get(f'{base_url}/')
    print(f'GET /: {response.status_code} - {response.json()}')

    # Test create user
    user_data = {'name': 'John Doe', 'email': 'john@example.com'}
    response = requests.post(f'{base_url}/api/users', json=user_data)
    print(f'POST /api/users: {response.status_code} - {response.json()}')
    user_id = response.json()['id']

    # Test get users
    response = requests.get(f'{base_url}/api/users')
    print(f'GET /api/users: {response.status_code} - {response.json()}')

    # Test get specific user
    response = requests.get(f'{base_url}/api/users/{user_id}')
    print(f'GET /api/users/{user_id}: {response.status_code} - {response.json()}')

    # Test delete user
    response = requests.delete(f'{base_url}/api/users/{user_id}')
    print(f'DELETE /api/users/{user_id}: {response.status_code}')

if __name__ == '__main__':
    test_api()
```

Best Practices and Production Considerations

Security

1. **Input Validation**: Always validate and sanitize user input
2. **Authentication**: Implement proper authentication mechanisms
3. **HTTPS**: Use SSL/TLS in production
4. **Rate Limiting**: Prevent abuse with rate limiting
5. **CORS**: Configure Cross-Origin Resource Sharing properly

Performance

1. **Caching**: Implement caching strategies

2. **Database Optimization**: Use proper indexing and queries
3. **Connection Pooling**: Manage database connections efficiently
4. **Load Balancing**: Distribute traffic across multiple servers

Monitoring and Logging

1. **Structured Logging**: Use consistent log formats
2. **Error Tracking**: Monitor and track errors
3. **Performance Metrics**: Track response times and throughput
4. **Health Checks**: Implement health check endpoints

Deployment

1. **Environment Variables**: Use environment variables for configuration
2. **Docker**: Containerize applications
3. **Process Managers**: Use tools like Gunicorn for production
4. **Reverse Proxy**: Use Nginx or Apache as reverse proxy

This comprehensive guide covers the fundamentals of Python web development, Flask framework usage, REST API design, and building HTTP servers from scratch. Each section builds upon the previous ones, providing practical examples and best practices for real-world applications.