Laborheft: Implementierung eines REST-Webservers mit Python

Übersicht

In diesem Labor wirst du lernen, wie du einen einfachen REST-Webserver in Python mit dem Modul `http.server` implementierst. Du wirst schrittweise einen Webserver erstellen, der verschiedene HTTP-Methoden unterstützt und JSON-Antworten zurückgibt.

Ziele

- Verstehen, wie ein HTTP-Server in Python funktioniert.
- Implementierung eines REST-Webservers, der GET- und POST-Anfragen verarbeitet.
- Testen des Webservers mit verschiedenen Werkzeugen.

Voraussetzungen

- Grundkenntnisse in Python.
- Installation von Python 3.x.

Schritt 1: Einrichten eines einfachen HTTP-Servers

1. Erstelle ein neues Python-Skript mit dem Namen `simple_http_server.py`.

2. Füge den folgenden Code ein:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print(f"Serving at port {PORT}")
    httpd.serve_forever()
```

3. Starte den Server, indem du das Skript ausführst:

```
python simple_http_server.py
```

4. Öffne einen Webbrowser und navigiere zu `http://localhost:8000`. Du solltest eine einfache Datei- oder Verzeichnisansicht sehen.

Schritt 2: Erstellen eines REST-Webservers

1. Erstelle ein neues Python-Skript mit dem Namen `rest_server.py`.

2. Füge den folgenden Code ein, um einen REST-Webserver zu implementieren:

```
from http.server import BaseHTTPRequestHandler, HTTPServer
import json

class RESTRequestHandler(BaseHTTPRequestHandler):
    def _set_response(self):
        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()

    def do_GET(self):
        self._set_response()
        url = self.path
        if url == "/exampleURL":
            exampleClass.exampleFunction()
        response = {'message': 'GET request received'}
        self.wfile.write(json.dumps(response).encode('utf-8'))

    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
        post_data = self.rfile.read(content_length)
        response = {'message': 'POST request received', 'data':
post_data.decode('utf-8')}
        self._set_response()
        self.wfile.write(json.dumps(response).encode('utf-8'))

def run(server_class=HTTPServer, handler_class=RESTRequestHandler,
port=8000):
    server_address = ('', port)
    httpd = server_class(server_address, handler_class)
    print(f'Starting httpd on port {port}...')
    httpd.serve_forever()

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

3. Starte den Server, indem du das Skript ausführst:

```
python rest_server.py
```

Dieser Code setzt einen einfachen HTTP-Server auf, der Anfragen vom Typ `GET` und `POST` verarbeitet. Der Server gibt JSON-Antworten zurück und kann auf Port 8000 betrieben werden. Du kannst diesen Server anpassen, indem du die Methoden in `RESTRequestHandler` änderst oder erweiterst, um zusätzliche Funktionalitäten hinzuzufügen.

Funktionen

`RESTRequestHandler`

Die `RESTRequestHandler`-Klasse erbt von `BaseHTTPRequestHandler` und überschreibt Methoden, um `GET`- und `POST`-Anfragen zu verarbeiten.

`_set_response(self)`

- **Beschreibung:** Setzt die HTTP-Antwort-Header und den Statuscode auf `200 OK`.
- **Details:**
  - `self.send_response(200)`: Sendet einen Statuscode `200 OK`.
  - `self.send_header('Content-type', 'application/json')`: Setzt den `Content-Type` der Antwort auf `application/json`.
  - `self.end_headers()`: Beendet die Header-Sektion der Antwort.

`do_GET(self)`

- **Beschreibung:** Verarbeitet `GET`-Anfragen.
- **Details:**
  - Ruft `_set_response()` auf, um die Antwort-Header zu setzen.
  - Prüft welche funktion aufgerufen werden soll.
  - Erzeugt eine JSON-Antwort mit der Nachricht `'GET request received'`.
  - Sendet die JSON-Antwort an den Client.

`do_POST(self)`

- **Beschreibung:** Verarbeitet `POST`-Anfragen.
- **Details:**
  - Liest die Länge der Daten aus den HTTP-Headern (`Content-Length`).
  - Liest die tatsächlichen `POST`-Daten vom Client.
  - Erzeugt eine JSON-Antwort, die die Nachricht `'POST request received'` sowie die empfangenen Daten enthält.
  - Ruft `_set_response()` auf, um die Antwort-Header zu setzen.
  - Sendet die JSON-Antwort an den Client.

`run(server_class=HTTPServer, handler_class=RESTRequestHandler, port=8000)`

- **Beschreibung:** Startet den HTTP-Server.
- **Details:**
  - `server_address = ('', port)`: Setzt die Serveradresse auf den angegebenen Port (Standard: 8000) und akzeptiert Verbindungen von allen IP-Adressen.
  - `httpd = server_class(server_address, handler_class)`: Erstellt eine Instanz des HTTP-Servers mit der angegebenen Adresse und Handler-Klasse.
  - `print(f'Starting httpd on port {port}...')`: Gibt eine Nachricht aus, dass der Server gestartet wird.
  - `httpd.serve_forever()`: Startet den Server und lässt ihn ununterbrochen laufen.

`if __name__ == '__main__':`

- **Beschreibung:** Stellt sicher, dass `run()` nur ausgeführt wird, wenn das Skript direkt ausgeführt wird (nicht importiert).
- **Details:**
  - Ruft die `run()`-Funktion auf, um den Server zu starten.

Schritt 3: Testen des REST-Webservers

1. **GET-Anfrage testen**:
   - Verwende `curl` oder einen Webbrowser, um eine GET-Anfrage an den Server zu senden:

     ```
     curl http://localhost:8000
     ```

   - Du solltest eine JSON-Antwort erhalten: `{"message": "GET request received"}`
2. **POST-Anfrage testen**:
   - Verwende `curl`, um eine POST-Anfrage mit Daten an den Server zu senden:

     ```
     curl -X POST -d "Hello, Server" http://localhost:8000
     ```

   - Du solltest eine JSON-Antwort erhalten, die die gesendeten Daten enthält: `{"message": "POST request received", "data": "Hello, Server"}`

Schritt 4: Erweiterung des Servers

1. **Weitere HTTP-Methoden implementieren**:
   - Erweitere den `RESTRequestHandler`, um PUT- und DELETE-Anfragen zu verarbeiten.

```
def do_PUT(self):
    self._set_response()
    response = {'message': 'PUT request received'}
    self.wfile.write(json.dumps(response).encode('utf-8'))

def do_DELETE(self):
    self._set_response()
    response = {'message': 'DELETE request received'}
    self.wfile.write(json.dumps(response).encode('utf-8'))
```

2. **Server neu starten und testen**:
   - Führe das Skript erneut aus und teste die neuen Methoden mit `curl`:

```
curl -X PUT http://localhost:8000
curl -X DELETE http://localhost:8000
```

Schritt 5: Weiterführende Aufgaben

1. **Verbesserung des Routings**:
   - Implementiere verbessertes Routing.
2. **Authentifizierung hinzufügen**:
   - Implementiere eine einfache Authentifizierungsmethode für den Server.
3. **Datenbankanbindung**:
   - Verbinde den Server mit einer SQLite-Datenbank und speichere Daten von POST-Anfragen.
4. **Deployment des Servers**:
   - Recherchiere, wie du den Server auf einem Remote-Server oder in der Cloud bereitstellen kannst (z.B. mit `gunicorn` oder `nginx`).

Zusammenfassung

In diesem Labor hast du einen einfachen REST-Webserver mit Python und `http.server` implementiert. Du hast gelernt, wie man verschiedene HTTP-Methoden verarbeitet und den Server testet. Mit diesen Grundlagen kannst du nun komplexere Webserver oder APIs entwickeln.

Ressourcen

- Python-Dokumentation zu http.server
- REST-API-Tutorial
- Python Packaging and Deployment