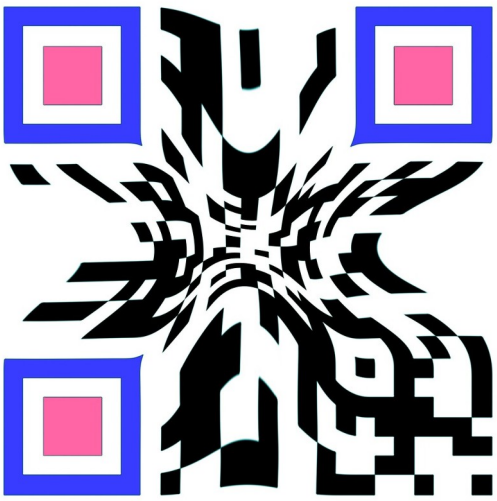
Softwaredokumentation

des containerisierten REST-API-Frameworks für Kommunikation zwischen multiplen Clients mit integriertem Last-Test-Modul



Anleitung und Dokumentation zu Architektur, Installation, Integrierbarkeit. Schnittstellen, Umsetzung, Testing und dem Einsatz des REST-Frameworks

Alexander Knüppel **Knueppelalex@web.de**

Navigator

1. Einleitung

Diese Dokumentation soll einen Überblick über die Funktionen und Aufgaben der Anwendung geben, sowie Anleitung bieten.
Es wurde versucht einerseits so S.M.A.R.T. als möglich und so detailliert als nötig vorzugehen, um eine Ausreichenden und Praktikablen Umgang mit dem erstellten Framework zu gewährleisten.

Die Dokumentation kann und soll nicht alle Lösungen für alle Anwendungs- und Problemfälle welche bei Benutzung auftreten können abdecken.

Da beim planen, erstellen und testen der Software und ihrer Komponenten weitestgehend Wert auf Robustheit und gleichzeitig Flexibilität gelegt wurde sind die in dieser Dokumentation beschriebenen Anweisungen und Umsetzungen das Ergebnis von ausgiebigen Recherchen, intensiver Planungen und ständigem überprüfen der Ergebnisse um jegliches Problem nach bestem Wissen zu eliminieren.

Wen betrifft diese Dokumentation?
Die Dokumentation soll in erster Linie an diejenigen gerichtet sein, welche mit meiner bereitgestellten Grundlage weitere Anwendungen und Umsetzungen entwickeln, weshalb diese Dokumentation sich eher an die Programmierer und Systemintegratoren, sowie Systemadministratoren richtet.

2. Architekturübersicht

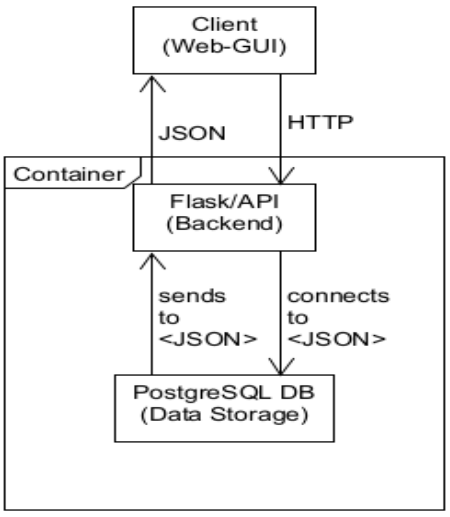


Diagramm 1: Struktur der Anwendung

Architekturübersicht der Anwendung

Die Anwendung ist eine RESTful API, welche mit Flask entwickelt wurde und eine PostgreSQL-Datenbank zur Speicherung von Testdaten verwendet. Die Architektur umfasst mehrere miteinander verknüpfte Schichten und Komponenten, um die Funktionalität eines Frameworks zum Datenaustausch mittels REST-Prinzipien bereitzustellen.
Nachfolgende Komponenten wurden in diese Architektur implementiert:

2.1 Frontend

- **Benutzeroberfläche**: in der aktuellen Implementierung wurde kein direktes Frontend erstellt, da alle Funktionen per HTTP-Anfragen über die API abgewickelt werden. Clients (z.B. Webanwendungen oder Server) interagieren in der Regel über Netzwerkanbindung mit der API und der dahinter liegenden Datenbank.

2.2 Backend

- **Flask-Anwendung**: Die Hauptanwendung wird durch Flask als Kommunikations-Server bereitgestellt. Folgenden Komponenten bilden dieses Framework:
  - ***com_server.py***: Startet die Flask-Anwendung und konfiguriert Umgebungsvariablen.
  - ***api_routes.py***:
    Erstellt, mittels des Flaskpakets „Blueprints“, Routen (API-Endpunkte) in welchen die Logik zur Verarbeitung von Anfragen (CRUD-Operationen für Benutzer) sowie deren Fehlerbehandlung definiert werden.
  - ***__init__.py***: Initialisiert die Flask-App und registriert die in der der api_routes erstellten „Blueprints“. Referenziert das Verzeichnis „*app*“ im Projektordner als Python-Paket.

2.3 Datenbank

- **PostgreSQL**: Die Datenbank speichert Testdaten der Tabelle „*users*“.
  Die Verbindung zur Datenbank erfolgt über einen Verbindungspool in *api_routes.py*.
- ***db_parameter.sql***: Skript zur Initialisierung der Datenbank mit Tabellen und Beispieldaten. Dient zum leichteren definieren und ändern der benötigten Tabellen.

2.4 Containerisierung

- **Docker**: Die Anwendung wird in Docker-Containern initialisiert, um eine einfache Bereitstellung und Skalierung zu ermöglichen.
  - ***Dockerfile***: Definiert das Image für die Flask-Anwendung, da auf keine vorhandenes Docker-Image zurückgegriffen werden kann. Parametrisiert Containerumgebung, verwendete Ports, integriert Anwendungen und führt bei Initialisierung der Container

einen auszuführender Befehl („*python com_server.py*“)
aus, welcher die Anwendung im Container startet.

- ***compose.yaml***: Konfiguriert die Docker-Container für die Flask-Anwendung und die PostgreSQL-Datenbank.

2.5 Tests

- **Unit-Tests**: Tests zur Überprüfung der API-Routen und deren Funktionalität.
- **Integrationstests**: Tests zur Überprüfung der Datenbankverbindung und CRUD-Funktionalität.
- **Lasttests**: Tests zur Überprüfung der Leistungsfähigkeit der API unter verschiedenen Lastbedingungen mit Locust.

3. Installation und Konfiguration

Dieses Kapitel beschreibt die erforderlichen Schritte zur Installation und Konfiguration der Anwendung, um sicherzustellen, dass sie ordnungsgemäß ausgeführt werden kann.

3.1.1Voraussetzungen Linux

Linux

Betriebssystem: Fedora-Workstation oder eine andere Linux-Distribution.
Docker: Notwendig für die Containerisierung der Anwendung.
Docker Compose: Erforderlich zur Verwaltung mehrerer Container.
Git: Referenzierung und Iterierung von Projekten
Python 3.x: Die Anwendung benötigt Python in der Version 3.x.
pip: Der Python-Paketmanager zur Installation der benötigten Abhängigkeiten.

3.2 Installation(Fedora)

1 Installation von Docker und Docker Compose

Docker-Installation sollte in Fedora über den Paketmanager erfolgen.
Bitte hierbei unbedingt die offizielle Anleitung auf https://docs.docker.com/engine/install/ beachten!

```
user@hostname:~$ sudo dnf install docker user@hostname:~$ sudo systemctl start docker
user@hostname:~$ sudo systemctl enable docker
user@hostname:~$ sudo dnf install docker-compose
```

2 Klonen des Repositorys

Das Repository der Anwendung mittels Git oder Github klonen:

```
user@hostname:~$ git clone <repository-url>
user@hostname:~$ cd <repository-name>
```

3 Installation der Abhängigkeiten

erforderliche Python-Pakete mittels Pip installieren:

```
user@hostname:~$ pip install -r requirements.txt
```

4 Starten der Docker-Container

Die Anwendung und die PostgreSQL-Datenbank werden automatisch beim Starten der Container initialisiert:

```
user@hostname:~$ docker-compose up --build
```

3.3 Konfiguration

1. **Umgebungsvariablen**
    - Die Umgebungsvariablen in *docker-compose.yml* sollten überprüft werden, um sicherzustellen, dass sie korrekt konfiguriert sind (z.B. Datenbankname, Benutzername, Passwort).
2. **Ports**
    - Die Portkonfiguration in *docker-compose.yml* sollte kontrolliert werden, um sicherzustellen, dass die API auf dem gewünschten Port (standardmäßig 5000) verfügbar ist.
3. **Datenbankverbindung**
    - Die Verbindung zur PostgreSQL-Datenbank in *api_routes.py* sollte auf korrekte Parameter überprüft werden.

3.4 Test der Installation

1. **Überprüfung des Containerstatus**
   Der Status aller laufenden Container kann mit folgendem Befehl überprüft werden:

   ```
   user@hostname:~$ docker ps
   ```

2. **Testen der API-Endpunkte**
   Um sicherzustellen, dass die API-Endpunkte erreichbar sind, kann ein cURL-Befehl verwendet werden:

   ```
   user@hostname:~$ curl -i -X GET http://localhost:5000/api/users
   ```

3.1.2 Voraussetzungen Windows

Windows

Betriebssystem: Windows 10 oder höher.
Docker oder Docker Desktop: Notwendig für die Containerisierung der Anwendung.
Docker Compose: Erforderlich zur Verwaltung mehrerer Container. In D. Desktop enthalten
Git: Referenzierung und Iterierung von Projekten
Python 3.x: Die Anwendung benötigt Python in der Version 3.x.
pip: Der Python-Paketmanager zur Installation der benötigten Abhängigkeiten.

3.2 Installation (Windows)

1.Docker und Docker Compose installieren

- Docker Desktop von der offiziellen Docker-Website : https://docs.docker.com/desktop/setup/install/windows-install/ herunterladen und installieren. Unbedingt die Installationsanweisung auch für „WSL 2“ oder „Hyper-V“ beachten und ggf. eine(!) der beiden installieren/nutzen.
- Sicher stellen, dass Docker Desktop nach der Installation gestartet wird und die Container-Ressourcen konfiguriert sind.

2.Klonen des Repositorys
Öffne die Eingabeaufforderung oder PowerShell und klone das Repository mit Git:

```
C:\> git clone <repository-url> C:\> cd <repository-name>
```

3.Installation der Abhängigkeiten

Stelle sicher, dass Python und pip installiert sind. Installiere die erforderlichen Python-Pakete:

```
C:\> pip install -r requirements.txt
```

4.Starten der Docker-Container

Die Anwendung und die PostgreSQL-Datenbank werden automatisch beim Starten der Container initialisiert. Der Befehl muss im Verzeichnis in welchem sich die *„compose.yaml“*-Datei der Anwendung befindet ausgeführt werden. :

```
C:\> docker-compose up --build
```

3.3 Konfiguration

1. **Umgebungsvariablen**
   - Überprüfe die Umgebungsvariablen in `docker-compose.yml`, um sicherzustellen, dass sie korrekt konfiguriert sind (z.B. Datenbankname, Benutzername, Passwort).
2. **Ports**
   - Kontrolliere die Portkonfiguration in `docker-compose.yml`, um sicherzustellen, dass die API auf dem gewünschten Port (standardmäßig 5000) verfügbar ist.
3. **Datenbankverbindung**
   - Überprüfe die Verbindung zur PostgreSQL-Datenbank in `api_routes.py` auf korrekte Parameter.

3.4 Test der Installation

1. **Überprüfung des Containerstatus**
   Der Status aller laufenden Container kann mit folgendem Befehl überprüft werden:
   ```
   C:\> docker ps
   ```

1.

2. **Testen der API-Endpunkte**
   Um sicherzustellen, dass die API-Endpunkte erreichbar sind, kann ein cURL-Befehl verwendet werden:

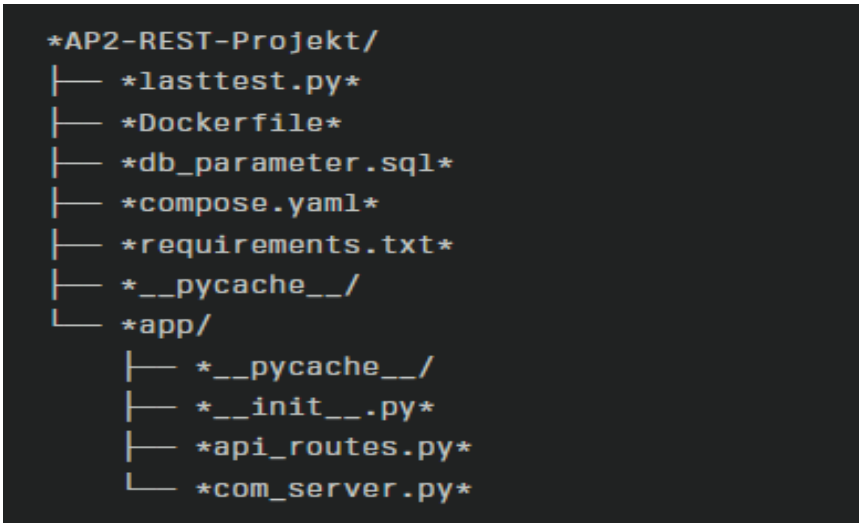
```
C:\> curl -i -X GET http://localhost:5000/api/users
```

4. Verzeichnisstruktur

Die Verzeichnisstruktur der Anwendung ist so gestaltet, dass sie eine klare Trennung der verschiedenen Komponenten und deren Funktionalitäten ermöglicht. Nachfolgend wird die Struktur detailliert beschrieben:

Abbildung 2:

Darstellung der Strukturierung des Projektordners



4.1 Hauptverzeichnis

- ***AP2-REST-Projekt/***: Das Hauptverzeichnis der Anwendung, das alle erforderlichen Dateien und Unterverzeichnisse enthält.

4.2 Wichtige Dateien und Ordner

- ***lasttest.py***: Skript zur Durchführung von Lasttests mit Locust, um die Leistungsfähigkeit der API unter verschiedenen Bedingungen zu überprüfen.
- ***Dockerfile***: Definiert das Docker-Image für die Flask-Anwendung, einschließlich der Installation von Abhängigkeiten und der Konfiguration der Containerumgebung.

- ***db_parameter.sql***: SQL-Skript zur Initialisierung der PostgreSQL-Datenbank mit Tabellen und Beispieldaten.
- ***compose.yaml***: Konfigurationsdatei für Docker Compose, die die Container für die Flask-Anwendung und die PostgreSQL-Datenbank definiert.
- ***requirements.txt***: Enthält eine Liste aller Python-Pakete, die für die Ausführung der Anwendung erforderlich sind.
- ***__pycache__/***: Verzeichnis, das von Python automatisch erstellt wird, um kompilierte Bytecode-Dateien zu speichern.

4.3 app/ Verzeichnis

- ***app/***: Enthält den Quellcode der Anwendung.
    - ***__init__.py***: Initialisiert das app/-Verzeichnis als Python-Paket und konfiguriert die Flask-App.
    - ***api_routes.py***: Definiert die API-Endpunkte und deren Logik zur Verarbeitung von Anfragen (CRUD-Operationen für Benutzer).
    - ***com_server.py***: Startet die Flask-Anwendung und konfiguriert Umgebungsvariablen.

5. API-Dokumentation

API-Endpunkte: Auflistung aller API-Endpunkte mit Beschreibung, erwarteten Eingaben und Ausgaben.

Statuscodes: Auflistung und Definition der Statuscodes (dies könnte das Kapitel sein, das du bereits erstellt hast).

Die API-Dokumentation soll eine umfassende Übersicht über die verfügbaren Endpunkte, deren Funktionalität und der erforderlichen Parameter bieten und soll Entwicklern als Referenz dienen, um die API effektiv zu nutzen.

5.1 Einführung

Die API ermöglicht den Zugriff auf die hinterlegten Datensätze aus der Datenbank und unterstützt grundlegende CRUD-Operationen (Create, Read, Update, Delete).

5.2 Authentifizierung

Zum aktuellen Stand der Entwicklung dieser Anwendung ist keine Authentifizierung implementiert, außer den nativ in Datenbank, Flask und Python integrierten/möglichen Lösungen, da es sich bei dem Projekt eher um eine Machbarkeitsstudie handelt welche ein grundlegendes Framework

anbietet, auf welchem man weitere Funktionalitäten und Anwendungen aufbauen, bzw. an dieses anbinden muss. Die Priorisierung liegt klar auf der Flexibilität und Kompatibilität mit der aktuell im Aufbau befindlichen AAS-Architektur des H2Go-Projekts und eventuellen zukünftig noch zu integrierenden Komponenten.

5.3 API-Endpunkte

GET /api/users

- **Beschreibung**: Gibt eine Liste aller Benutzer zurück.
- **HTTP-Methode**: GET
- **Antwortformat**: JSON
- **Statuscodes**:
  - 200: Erfolgreiche Anfrage
  - 500: Interner Serverfehler

GET /api/users/{user_id}

- **Beschreibung**: Gibt die Details eines spezifischen Benutzers zurück.
- **HTTP-Methode**: GET
- **Pfadparameter**:
  - `user_id`: Die ID des Benutzers.
- **Antwortformat**: JSON
- **Statuscodes**:
  - 200: Benutzer gefunden
  - 404: Benutzer nicht gefunden

POST /api/users

- **Beschreibung**: Erstellt einen neuen Benutzer.
- **HTTP-Methode**: POST
- **Anforderungsdaten** (JSON):

Abbildung 3: Ausgegebenen JSON-Struktur

```
{
    "username": "string",
    "phone": "string",
    "ip": "string"
}
```

- **Antwortformat**: JSON
- **Statuscodes**:
  - 201: "id": username
  - 400: Benutzer konnte nicht erstellt werden, bitte wiederholen
  - 400: Benutzer konnte nicht erstellt werden, bitte wiederholen

Nicht implementierte Endpunkte (als Beispiel) :

PUT /api/users/{user_id}

- **Beschreibung**: Aktualisiert die Informationen eines bestehenden Benutzers.
- **HTTP-Methode**: PUT
- **Pfadparameter**:
  - `user_id`: Die ID des Benutzers.
- **Anforderungsdaten** (JSON):
  - **Antwortformat**: JSON
  - **Statuscodes**:
    - 200: Benutzer erfolgreich aktualisiert
    - 404: Benutzer nicht gefunden

DELETE /api/users/{user_id}

- **Beschreibung**: Löscht einen bestehenden Benutzer.
- **HTTP-Methode**: DELETE
- **Pfadparameter**:
  - `user_id`: Die ID des Benutzers.
- **Antwortformat**: JSON
- **Statuscodes**:
  - 204: Benutzer erfolgreich gelöscht
  - 404: Benutzer nicht gefunden

5.4 Beispiele für Anfragen

Die Endpunkte sind ebenfalls über cURL-Befehle erreichbar und könne Anfragen beantworten:

➔ Abrufen aller Benutzer:

```
curl -i -X GET http://localhost:5000/api/users
```

➔ Abrufen eines spezifischen Benutzers:

```
curl -i -X GET http://localhost:5000/api/users/1
```

➔ Erstellen eines neuen Benutzers:

```
curl -i -X POST http://localhost:5000/api/users \
  -H "Content-Type: application/json" \
  -d '{"username": "testuser", "phone": "1234567890", "ip": "192.168.1.10"}'
```

5.5 Fehlermeldungen

Die implementierten Fehlermeldungen und deren Bedeutung:

- **400 error**: Benutzer konnte nicht erstellt werden, bitte wiederholen
- **400 error**: Benutzername bereits vergeben
- **404 error**: User nicht gefunden.
- **500 error**: <Fehlermeldung>
- **500 error**: Ein unerwarteter Fehler ist aufgetreten.

6. Tests

6.1 Teststrategie

Das entworfene Set an Tests folgt einer klaren und zeitsparenden Strategie, die Tests möglichst gleichzeitig aus.- bzw. durchzuführen. Um effizient und dennoch ausreichend exakt zu sein. Deshalb wurde entschieden das vorab überlegte Set:

1. **Unit-Tests**
2. **Integrationstests**
3. **Lasttests**
4. **Zusatztest zur Testroute**

Teilweise oder komplett ineinander zu integrieren. Dies lag nahe, da sich aufgrund der simplen aber effektiven Strukturen der verwendeten Frameworks, Komponenten und der Sprache Python bei beinahe jedem ausführen der Anwendung die Testfelder in nahezu allen Bereichen überschneiden. Das endgültig festgelegte Testset beschränkt sich somit auf:

1. **Unit-Tests**
2. **Integrationstests**
3. **Lasttests**

6.2 Testfälle und Umsetzung

Unit-Tests

Test der API-Routen:
1.1: Prüfung, ob die GET-Anfrage an /api/users eine 200-Statusantwort zurückgibt und die erwarteten Daten enthält.
1.2: Prüfung, ob die GET-Anfrage an /api/users/<user_id> bei einem vorhandenen Benutzer eine 200-Statusantwort zurückgibt und die korrekten Benutzerdaten liefert.
1.3: Prüfung, ob die GET-Anfrage an /api/users/<user_id> bei einem nicht vorhandenen Benutzer eine 404-Statusantwort zurückgibt.
1.4: Prüfung, ob die POST-Anfrage an /api/users einen neuen Benutzer erfolgreich erstellt (200 oder 201) und die richtige ID zurückgibt.
1.5: (*Zusatztest zur integrierten Testroute)*
Datenformat-Ausgabe überprüfen auf einer Testroute per Browser und Konsole.
Überprüfen per Webbrowser und cURL-Anfrage auf der Test-Route /test, um sicherzustellen, dass diese das erwartete JSON-Format korrekt zurückgibt.

Umsetzung:

1.1 GET-Anfrage an /api/users

Prüfung, ob die GET-Anfrage eine 200-Statusantwort zurückgibt und die erwarteten Daten enthält:

```
user@hostname:~$ curl -i -X GET http://localhost:5000/api/users
```

(*-i: Zeigt die HTTP-Header der Antwort an.)*

Erwartete Antwort: Status 200 und eine Liste von Benutzern im JSON-Format.

1.2 GET-Anfrage an /api/users/<user_id>

Prüfung, ob die GET-Anfrage bei einem vorhandenen Benutzer eine 200-Statusantwort zurückgibt:

```
user@hostname:~$ curl -i -X GET http://localhost:5000/api/users/1
```

Erwartete Antwort: Status: 200 und die Benutzerdaten im JSON-Format.

1.3 GET-Anfrage an /api/users/<user_id> (nicht vorhandener Benutzer)

Prüfung, ob die GET-Anfrage bei einem nicht vorhandenen Benutzer eine 404-Statusantwort zurückgibt:

```
user@hostname:~$ curl -i -X GET http://localhost:5000/api/users/1234567890
```

(ID 1234567890 garantiert das dieser User nicht vorhanden ist, da der Zahlenraum der durch lasttest.py generierten User nur maximal 5 Stellen haben kann)

Erwartete Antwort: Status 404 und eine Fehlermeldung im JSON-Format.

1.4 POST-Anfrage an /api/users

Prüfung, ob die POST-Anfrage einen neuen Benutzer erfolgreich erstellt (200 oder 201) und die richtige ID zurückgibt:

```
user@hostname:~$ curl -i -X POST http://localhost:5000/api/users \
    -H "Content-Type: application/json" \
    -d '{"username": "testuser", "phone": "1234567890", "ip": "192.168.1.10"}'
```

(-H "Content-Type: application/json": Setzt den Header für den Inhaltstyp auf JSON.
-d: Überträgt die JSON-Daten)

Erwartete Antwort: Status 201 (oder 200) und eine JSON-Antwort mit der ID des neu erstellten Benutzers.

1.5 Zusatztest zur Testroute

Datenformat überprüfen auf einer Testroute per Browser und Konsole:
Überprüfen per Webbrowser und curl-Anfrage der /test-Route, um sicherzustellen, dass sie das erwarteten JSON-Antwortformat korrekt zurückgibt.

Umsetzung:

Überprüfung per Webbrowser:
http://localhost:5000/test

Erwartete Antwort:
Eine JSON-Antwort im Format: {"message": "Hallo Testsubjekt"}

Überprüfung per cURL-Anfrage:
Führe den folgenden cURL-Befehl im Terminal aus:

user@hostname:~$ curl -i -X GET http://localhost:5000/test

Erwartete Antwort: Eine JSON-Antwort im Format: { "message": "Hallo Testsubjekt"}

2. Testvorgabe:

Integrationstests

2.1: Datenbankverbindung:
Testen der Verbindung zur PostgreSQL-Datenbank, um sicherzustellen, dass diese korrekt konfiguriert ist und Anfragen verarbeitet werden können.

Umsetzung:
Ausführen des erstellten Testskriptes „*test_db_connection.py*“ in der bash:
user@hostname:~$ python test_db_connection.py

Erwartete Antwort: Erfolgsmeldung, die bestätigt, dass die Datenbankverbindung hergestellt werden konnte.

2.2: CRUD-Funktionalität:
Simulieren der Erstellung eines Benutzers über die API und prüfen, ob dieser in der Datenbank vorhanden ist.
Ausführen des erstellten Testskriptes „*test_crud_functionality.py*“ in der bash:

user@hostname:~$ python test_crud_functionality.py

Erwartete Antwort: Bestätigung, dass der Benutzer erfolgreich erstellt wurde und in der Datenbank gefunden werden kann.

3. Testvorgabe:

Lasttests

3.1: Leistungsüberprüfung:

Lasttests mit Locust durchführen, um sicherzustellen, dass die API unter verschiedenen Lastbedingungen stabil bleibt (1, 2, 5, 20, 50, 100 gleichzeitigen Usern).

Umsetzung:
Starte Locust mit dem Lasttest-Skript:
user@hostname:~$ locust -f lasttest.py --host=http://localhost:5000

Öffnen des Locust-internen WebGUI im Webbrowser *(Default-Adresse und -Port: http://localhost:8089)*, um die Tests zu konfigurieren und zu starten.

Erwartetes Ergebniss: Stabilität der API unter den angegebenen Lastbedingungen ohne signifikante Verzögerungen (unter 200 ms Reaktionszeit) oder auftretende Fehler.
Ergebnisse:
Status: Analyse der Antwortzeiten und Fehlerquoten während des Tests.

3.2: Ramp-Up:

Lasttest mit 50 Usern und erhöhtem Ramp-Up-Wert, um die Stabilität zu ermitteln.

Umsetzung

Zusätzliches Testset mit dem selben Lasttest-Skript und 50 Usern, aber erhöhtem Ramp-Up-Wert (default ist 1) durchführen, um die Stabilität zu prüfen.
Hierzu wird in den Startparametern der Web-GUI ein höherer Ramp-Up-Wert eingetragen und gestartet.

Erwartetes Ergebniss:
Die API sollte auch unter diesen Bedingungen stabil bleiben.

3.3: Errorhandling:

mittels eines angepassten Locust-Skriptes werden Fehler erzeugt um zu ermitteln ob das implementierte Errorhandling der Anwendung ausreicht.

Umsetzung:

Modifizieren des Locust-Skripts, um absichtlich fehlerhafte Anfragen zu senden.

Erwartete Antwort:

Die API sollte angemessen auf Fehler reagieren (z.B. Status 400) und entsprechende Fehlermeldungen zurückgeben. Dies sollte jedoch keine Verbindungsabbrüche, oder gar eine Einfrieren oder das Abstürzen der Anwendung verursachen.

7. Fehlerbehandlung

Die Fehlerbehandlung wurde weitestgehend mittels Try-Except und definierter Statuscodes umgesetzt, um die Stabilität und Transparenz der Prozesse im Server und der Datenbank zu gewährleisten.

Beispiel einer umgesetzten try-except implementierung in der get_all_users-Route:

```
@api_blueprint.route("/api/users", methods=["GET"])
      def get_all_users():
            conn = get_db_connection()
            try:
                  with conn.cursor() as cursor:
                        cursor.execute("SELECT * FROM users;")
                        users = cursor.fetchall()
                  return jsonify(users)
            except Exception as e:
                  return jsonify({"error": str(e)}), 500
            finally:
                  release_db_connection(conn)
```

Detailbeschreibung der Umsetzung:

Die Fehlerbehandlung in der entwickelten API erfolgt über strukturierte Ausnahmen und spezifische Rückgabewerte, die es ermöglichen, sowohl erfolgreiche als auch fehlerhafte Anfragen angemessen zu verarbeiten. Die Implementierung umfasst drei Haupt-Routen: das Abrufen aller Benutzer, das Abrufen eines spezifischen Benutzers sowie das Hinzufügen neuer Benutzer. Jede dieser Routen enthält spezifische Mechanismen zur Handhabung von Fehlern, die während der Ausführung auftreten können.

Erfolgreiche Anfragen

Für erfolgreiche Anfragen werden entsprechende HTTP-Statuscodes zurückgegeben. Bei der Abfrage aller Benutzer wird ein Statuscode von 200 OK verwendet, während das Hinzufügen eines neuen Benutzers mit dem Statuscode 201 Created bestätigt wird. Diese Rückmeldungen sind konform mit den RESTful-Prinzipien und bieten den Clients klare Informationen über den Erfolg ihrer Anfragen.

Fehlerbehandlung

Die API implementiert eine umfassende Fehlerbehandlung, die in den try-except-Blöcken jeder Route untergebracht ist. Im Falle eines Fehlers wird ein Statuscode von 500 Internal Server Error zurückgegeben, begleitet von einer detaillierten Fehlermeldung. Dies ermöglicht es Entwicklern und Systemadministratoren, Probleme schnell zu identifizieren und zu beheben.

Für spezifische Fehler wie das Nichtvorhandensein eines Benutzers wird ein Statuscode von 404 Not Found zurückgegeben, während bei einem Konflikt aufgrund eines bereits vergebenen Benutzernamens ein Statuscode von 400 Bad Request verwendet wird. Diese differenzierte Handhabung von Fehlern trägt zur Robustheit der API bei und verbessert die Benutzererfahrung.

Beurteilung der Implementierung

Die implementierte Fehlerbehandlung ist grundsätzlich ausreichend für eine Anwendung dieser Art. Die Verwendung von spezifischen Statuscodes ermöglicht es den Clients, auf unterschiedliche Fehlerszenarien angemessen zu reagieren. Die klare Trennung zwischen verschiedenen Arten von Fehlern (z.B. Client-Fehler vs. Server-Fehler) ist ein positiver Aspekt dieser Implementierung.

Allerdings gibt es einige Schwachstellen und Verbesserungspotenziale:

Fehlende Validierung von Eingabedaten: Derzeit wird nicht überprüft, ob die vom Client gesendeten Daten vollständig und korrekt sind, bevor sie in die Datenbank eingefügt werden. Eine Validierung könnte dazu beitragen, häufige Fehlerquellen frühzeitig zu erkennen und entsprechende Rückmeldungen zu geben.

Allgemeine Fehlermeldungen: Während die Rückgabe einer allgemeinen Fehlermeldung ("Ein unerwarteter Fehler ist aufgetreten") hilfreich ist, könnte es sinnvoll sein, diese Meldungen weiter zu differenzieren oder zu anonymisieren, um sicherzustellen, dass keine sensiblen Informationen über die interne Logik der Anwendung preisgegeben werden.

Logging: Eine umfassende Protokollierung von Fehlern wäre vorteilhaft, um eine Nachverfolgbarkeit und Analyse im Falle von Problemen zu gewährleisten. Dies könnte helfen, Muster in den Fehlern zu erkennen und die Stabilität der Anwendung langfristig zu verbessern.

Testabdeckung: Um sicherzustellen, dass die Fehlerbehandlung unter verschiedenen Bedingungen funktioniert, sollte eine umfassende Testabdeckung implementiert werden. Unit-Tests und Integrationstests könnten dazu beitragen, potenzielle Schwachstellen in der Fehlerbehandlung frühzeitig zu identifizieren.

Insgesamt bietet die aktuelle Implementierung eine solide Grundlage für die Fehlerbehandlung in der API, jedoch sollten die genannten Schwachstellen adressiert werden, um die Robustheit und Benutzerfreundlichkeit weiter zu erhöhen.

Performanceverbesserungen:

Eine Verbesserung der Performance wurde erreicht, indem man in der a*pi_routes.py* ein Verbindungs-Pooling implementierte, welches das aufrechterhalten der Verbindung zur Datenbank erlaubt. Somit muss nicht bei jeder Anfrage erneut verbunden werden. Dies hilft die Auslastung zu minimieren weil dadurch der Overhead reduziert und zusätzliche Ressourcen frei werden. Zudem managed und limitiert Pooling die gleichzeitigen offenen Verbindungen was Überlastung der Datenbank verhindern soll.

Das Pooling wurde wie Folgt implementiert:

```
connection_pool = pool.SimpleConnectionPool(1, 20,
            dbname="REST_DB",
            user="postgres",
            password="test1234",
            host="db",
            port="5432")


    def get_db_connection():
            return connection_pool.getconn()


    def release_db_connection(conn):
            connection_pool.putconn(conn)
```

Weitere Fehler sind in bisherigen Ausführungen von Komponenten oder Anwendung nicht Aufgetreten oder wurden behoben. Zumeist waren diese jedoch eher auf mangelnde Erfahrung oder Unachtsamkeiten zurückzuführen und nicht auf Fehler in der Logik oder der allgemeinen Umsetzung.

8. Anwendungsfälle

Die erstellte und im Dokument behandelte Anwendung wurde mit der Intention erstellt, ein kleines und flexibles Framework anzubieten welches simpel um- und einsetzbar ist, aber dennoch leistungsstark genug um es ggf. auch für die Realisierung einer Schnittstelle zwischen mehreren Servern und einer Datenbank oder einem Backup-System (oder mehreren) einzusetzen. Diese API soll automatisierte „Kommunikation“ und Datenaustausch selbstverwaltet ermöglichen bzw. vereinfachen. Aufgrund der Grundprämissen „Flexibilität“, „Robustheit“, „Kompatibilität“, „gute Verständlichkeit“ und „Zukunftssicherheit“ wurde Wert auf Standardisierte Schnittstellen, Protokolle, Formate, Frameworks und Programmiersprachen gelegt. Alle diese genannten „Werkzeuge“ sind meines Wissens nach gut dokumentiert, weit Verbreitet und bieten eine breite Basis an Unterstützung durch deren Verbreitung und Quelloffenheit.

9. Fazit

REST-API-Framework

Auf einen Blick:

Schnittstelle: REST-API

integrierte Kommunikation: HTTP (HTTPS nicht integriert, aber möglich)
integriertes Austauschformate: JSON

Webserver: Flask

benutzte Sprache: Python

Lasttest Framework: Locust

benutzte Sprache: Python
Kommunikation: HTTP/HTTPS (und einige andere)

Containerisierung: Docker

- Open Source
- ressourceneffizienter als Virtualisierung
- Linux nativ
- Super Flexibel

Sprachen: Python

- Breite Supportbasis,
- Linux nativ
- klare und leichte Syntax

Datenbank: PostgreSQL
unterstützte Datentypen: JSON, XML, PostGIS (Geodaten)
unterstützte Betriebssysteme: Linux, Windows, MacOS etc.
- Open Source
- Breite Basis, Große Community

Ausblick

Denkbar wäre mit diesem Framework, was auch die Grundidee war, eine Backup-System einzurichten welches flexible automatisierte Datenverteilungen realisiert, was wiederum ein eigenes automatisiertes Synchronisationssystem ermöglichen kann. Welches eine Art immer aktuelle Master-Instanz bildet welche je nach Bedarf und verfügbarer Ressourcen alle Teilnehmer aktuell hält und eine Echtzeitbackup bietet. Quasi eine art Cloud oderP2P-Netzwerk.
Natürlich lassen sich auch andere Umsetzungen vorstellen, vom Privaten Bereich bis hin zu Industrie.

10. Anhang

10.1 Referenzen

1. Docker

- **Link**: www.docker.com
    - Offizielle Homepage des Docker-Projekts, das Entwicklern hilft, Container-Anwendungen zu erstellen, zu teilen und auszuführen.
- **Link**: Docker Docs

    - Offizielle Dokumentation von Docker mit Ressourcen, Handbüchern und Anleitungen zur Containerisierung von Anwendungen.
- **Link**: Docker Official Images
    - Übersicht über die offiziellen Docker-Images, die bewährte Praktiken fördern und regelmäßig aktualisiert werden.

2. Flask

- **Link**: flask.palletsprojects.com
    - Offizielle Homepage von Flask, einem Mikro-Webframework für Python, das die Entwicklung von Webanwendungen erleichtert.

3. PostgreSQL

- **Link**: www.postgresql.org
    - Offizielle Homepage von PostgreSQL, einer leistungsstarken Open-Source-Datenbank.

4. Python

- **Link**: www.python.org
    - Offizielle Homepage von Python, einer weit verbreiteten Programmiersprache für die Entwicklung von Anwendungen.

5. Docker Compose

- **Link**: docs.docker.com/compose
    - Offizielle Dokumentation zu Docker Compose, das es ermöglicht, Multi-Container-Anwendungen zu definieren und auszuführen.

6. Locust

- **Link**: locust.io
    - Offizielle Homepage von Locust, einem Lasttest-Tool für Webanwendungen.

10.2 Anhang Statuscode-Dokumentation

Erfolgreiche Anfragen

GET /api/users
Statuscode: **200**
Statusmessage: "OK"

Trigger: Erfolgreiche Abfrage aller Benutzer.

GET /api/users/int:user_id
Statuscode: **200**
Statusmessage: "User-Daten"
Trigger: Erfolgreiche Abfrage eines spezifischen Benutzers.
Statuscode: **404**
Statusmessage: "error": "User nicht gefunden"
Trigger: Benutzer mit der angegebenen ID existiert nicht.

POST /api/users
Statuscode: **201**
Statusmessage: "id": user_id
Trigger: Benutzer erfolgreich erstellt, ID zurückgegeben.

Fehlerhafte Anfragen:

GET /api/users
Statuscode: **500**
Statusmessage: "error": "Ein unerwarteter Fehler ist aufgetreten: <Fehlermeldung>"
Trigger: Fehler bei der Datenbankabfrage oder Verbindung.

GET /api/users/int:user_id
Statuscode: **500**
Statusmessage: "error": "<Fehlermeldung>"
Trigger: Fehler bei der Datenbankabfrage oder Verbindung.

POST /api/users
Statuscode: **400**
Statusmessage: "error": "Benutzer konnte nicht erstellt werden, bitte wiederholen"
Trigger: Allgemeiner Fehler beim Erstellen des Benutzers (z. B. fehlende Daten).
Statuscode: **400**
Statusmessage: "error": "Benutzername bereits vergeben"
Trigger: Versuch, einen Benutzer mit einem bereits existierenden Benutzernamen zu erstellen.
Statuscode: **500**
Statusmessage: "error": "Ein unerwarteter Fehler ist aufgetreten: <Fehlermeldung>"
Trigger: Fehler bei der Datenbankoperation oder Verbindung.
Fehler außerhalb von fehlerhaften Anfragen

Für alle Routen:

Statuscode: **500**

Statusmessage: "error": "Ein unerwarteter Fehler ist aufgetreten: <Fehlermeldung>"
Trigger: Allgemeine Serverfehler, die nicht spezifisch auf die Anfrage zurückzuführen sind (z. B. Datenbankverbindungsprobleme).