Pflichtenheft

zur Umsetzung eines containerisierten Frameworks zur Kommunikation zwischen multiplen Clients mit integriertem Lasttestmodul

Alexander Knüppel **Version: 1.0**

Inhaltsverzeichnis

25.11.2024 **Alexander Knüppel**

Einleitung:

Dieses Pflichtenheft beschreibt die technischen Anforderungen und Spezifikationen, welche in der beschriebenen Anwendung umgesetzt werden sollen.
Ziel des Projekts ist es, ein flexibles und skalierbares Framework zu entwickeln, welches die Kommunikation zwischen multiplen Clients und einer containerisierten Datenbank ermöglicht.

Die Zielgruppe umfasst Entwickler und Tester, die an der Implementierung beteiligt sind. Das Projekt beinhaltet die Erstellung eines Flask-Servers mit REST-API und die Integration eines Lasttestmoduls.

Dieses Dokument dient als Leitfaden für die Umsetzung des Projekts.

Ausgangssituation

Das Forschungsprojekt "H2GO – virtuelle Referenzfabrik" zielt auf die Digitalisierung einer automatisierten Brennstoffzellenproduktion mittels Verwaltungsschalen nach Industrie-4.0-Standards. Mehrere Fraunhofer-Institute arbeiten an der virtuellen Abbildung der Prozesse, verteilt auf vier Standorte.
Ein zentraler Registry-Server verwaltet die Daten der AAS.
Umgesetzt wird dieser Server mittels des BaSyx-AAS-Registry-Server- Frameworks, welches eine REST-Schnittstelle bereitstellt.
Aktuell weist dieser Server jedoch keine relevante Ausfallsicherheit auf, ein Ausfall könnte die gesamte verteilte Produktion gefährden.

Zielsetzung

Um dieses Risiko zu minimieren, kann eine dezentrale und automatisierte Echtzeitlösung im Sinne eines Redundant Distributed File Systems (RDFS) mittels verteilter Backup-Registry-Server sinnvolle Sicherheit bieten.
Die Herausforderung besteht darin, eine Lösung zu entwickeln, die interoperabel, grundlegend, anpassbar und leicht verständlich ist, um den Anforderungen des Projekts und der Vielzahl der beteiligten Personen gerecht zu werden.
Zusätzlich soll eine Testumgebung integriert werden, um die Belastbarkeit der Anwendung messbar zu machen und potenzielle Fehler oder Engpässe frühzeitig zu identifizieren.
Aufgrund des derzeitigen nur teil- und testweise umgesetzten Architektur des o.g. AAS-Registry-Servers wird das REST-Framework in einem grundlegend funktionale und kompatiblen, aber nicht spezifisch in die derzeitige AAS-Architektur einbindbaren Zustand erstellt. Dieser Ansatz der Umsetzung soll gewährleisten das zukünftige am H2Go-AAS-Registry-Server bzw. dessen Architektur zu implementierende Komponenten, Anwendungen etc. möglichst nicht durch das

REST-Framework limitiert werden und umgekehrt das Framework einfach und flexibel auch im weiteren Verlauf des H2Go-Projektes integriert werden kann.

Erwartete Ergebnisse

Am Ende des Projekts wird ein funktionsfähiger Flask-Server implementiert sein, der über eine REST-API die Kommunikation zwischen den Clients (Registry-Server, Backup-Server, Lasttest) und dem containerisierten Backup-Registry-Server ermöglicht. Zudem wird ein Lasttestmodul integriert, um die Belastbarkeit des Systems zu überprüfen.
Um bereits erste Funktionstests der Anwendung durchführen zu können wird eine Datenbankanwendung als Dummy im Framework integriert.

Systemübersicht

Systemarchitektur

Das System besteht aus mehreren containerisierten Komponenten: einem Flask-Server, einer Datenbank und einem Lasttestmodul. Diese Komponenten kommunizieren über eine REST-API und sind so konzipiert, dass sie in einer Docker-Umgebung betrieben werden.
Zudem sind folgenden Komponenten notwendigerweise ausserhalb der Containerisierung angesiedelt: die Main-Anwendung welche das Framework startet, sowie dem Lasttestmodul welches Lasttests bereit stellt.

Hauptkomponenten

1. Main-Anwendung: Startpunkt der Anwendung, erstellt und/oder konfiguriert die nachfolgenden Module.
2. Flask-Server: Der zentrale Bestandteil des Systems, der als REST-API fungiert und die Kommunikation zwischen den Clients und der Datenbank ermöglicht.
3. Datenbank: Eine Datenbankanwendung wird als Dummy integriert, um erste Funktionstests durchzuführen und als Platzhalter für zukünftige Datenbankimplementierungen zu dienen.
4. Lasttestmodul (Locust): Dieses Modul wird verwendet, um die Belastbarkeit des Systems zu überprüfen und potenzielle Engpässe zu identifizieren.

Technologien

Das System nutzt Docker zur Containerisierung der Anwendungsteile sowie Flask für die Entwicklung des Servers. Die Kommunikation erfolgt über eine REST-API. Locust wird verwendet, um Lasttests durchzuführen und die Performance des Systems zu evaluieren.

Funktionale Anforderungen

Allgemeine Anforderungen

1. Das System muss in der Lage sein, Anfragen über eine REST-API zu empfangen und zu verarbeiten.

2. Die Anwendung muss eine benutzerfreundliche Schnittstelle bieten.

3. Das System soll grundsätzlich sicherstellen, dass sensible Daten geschützt sind.

Spezifische Funktionen

1. Main-Anwendung:
- Die Main-Anwendung muss in der Lage sein, alle erforderlichen Module zu starten und zu konfigurieren.

2. Flask-Server:
- Der Flask-Server muss folgende Endpunkte bereitstellen:
    - `GET /api/data`: Gibt aktuelle Daten aus der Datenbank zurück.
    - `POST /api/data`: Ermöglicht das Hinzufügen neuer Daten zur Datenbank.

3. Datenbank:
- Die Dummy-Datenbank muss einfache CRUD-Operationen (Create, Read, Update, Delete) unterstützen.
- die Datenbank sollte leicht austauschbar und Änderungen sollten leicht umsetzbar sein.

4. Lasttestmodul (Locust):
- Das Lasttestmodul muss in der Lage sein, simulierte Benutzeranfragen an den Flask-Server zu senden und die Performance zu messen.

Interaktionen

- Der Flask-Server empfängt API-Anfragen von Clients (z.B. Lasttest, Registry-Server, sonstige Stakeholder) und verarbeitet diese durch Interaktion mit der Datenbank.

Protokolle und Datenformate

Die Kommunikation zwischen den Komponenten erfolgt über das HTTP-Protokoll. Die API

verwendet JSON (siehe Begriffserläuterung 1.) als Datenformat für Anfragen und Antworten, da es leichtgewichtig, menschenlesbar und von vielen Programmiersprachen unterstützt wird. Diese Wahl ermöglicht eine effiziente Datenübertragung und eine einfache Integration in verschiedene Clients.

Nicht-funktionale Anforderungen

1. Performance:
- Das System sollte mindestens 100 gleichzeitige Benutzeranfragen verarbeiten können.

2. Zuverlässigkeit:
- Das System muss eine Verfügbarkeit von 99,9 % gewährleisten.
- Im Falle eines Ausfalls sollte eine parallele Instanziierung einer neuen Anwendung, welche gut eingebunden werden kann, für nur minimalste Ausfallzeiten sorgen.

3. Sicherheit:
- Ein Authentifizierungsmechanismus soll in der Datenbank implementiert werden und den zugriff durch nicht qualifizierte User unterbinden.

4. Benutzerfreundlichkeit:
- Die Benutzeroberfläche muss intuitiv gestaltet sein und eine einfache Navigation ermöglichen.
- Dokumentation zur Nutzung der API muss bereitgestellt werden.

5. Wartbarkeit:
- Der Code muss gut dokumentiert sein, um zukünftige Wartungsarbeiten zu erleichtern.
- Das System sollte modular aufgebaut sein, um Änderungen an einzelnen Komponenten ohne Auswirkungen auf andere Teile des Systems zu ermöglichen.

6. Zukunftsicherheit:
- die Anwendung soll flexibel gestaltet werden, um zukünftige Implementierungen in ihrer Umgebung konfliktfrei zu akzeptieren.
- die Anwendung soll interoperabel sein, um eine breite Kompatibilität mit anderen Systemen zu gewährleisten.

7. Portierbarkeit:
- Die Anwendung soll auf einer breiten Auswahl an Systemen ausführbar sein, einschließlich gängiger Betriebssysteme wie Windows und Linux.
- Die Anwendung soll so entwickelt werden, dass sie mit Standardprogrammierschnittstellen (APIs) kompatibel ist, um eine einfache Portierung auf verschiedene Plattformen zu ermöglichen.

Schnittstellen

API-Schnittstellen

Die Anwendung stellt eine REST-API zur Verfügung, die folgende Endpunkte umfasst:

1. GET /api/data:

Beschreibung: Gibt aktuelle Daten aus der Datenbank zurück.

- Eingabe: Keine
- Ausgabe: JSON-Objekt mit den aktuellen Daten

2. POST /api/data:

Beschreibung: Ermöglicht das Hinzufügen neuer Daten zur Datenbank.

- Eingabe: JSON-Objekt mit den zu speichernden Daten
- Ausgabe: Bestätigungsnachricht + Statuscode oder Fehlerstatuscode

3. Die Kommunikation erfolgt im JSON-Format. Beispiel für eine typische Anfrage an den POST-Endpunkt:

```json
{"name": "Beispielname",
  "wert": 42}
```

Locust-Schnittstelle

Das Lasttestmodul (Locust) kommuniziert direkt mit der REST-API des Flask-Servers. Die Tests werden über eine Web-GUI gestartet, die es ermöglicht, die Anzahl der simulierten Benutzer und deren Verhalten zu konfigurieren.

1. Web-GUI:

- Die Web-GUI von Locust wird unter `http://localhost:8089` bereitgestellt. Eine entsprechende Anleitung hierfür wird als Meldung einschließlich eines Links zum Localhost in der Konsole ausgegeben.
- Über diese Oberfläche können Benutzerparameter wie die Anzahl der Benutzer und die Ramp-Up-Rate (siehe Begriffserläuterung 2.) festgelegt werden.

2. Interaktion:

- Locust sendet HTTP-Anfragen an die Endpunkte der Flask-API (z.B. `GET /api/data`, `POST /api/data`), um Lasttests durchzuführen und die Performance der Anwendung zu messen.

- Die Ergebnisse der Tests, einschließlich Antwortzeiten und Fehlerraten, werden in der Web-GUI in Echtzeit angezeigt.

Interne Schnittstellen

Der Flask-Server kommuniziert mit der Dummy-Datenbank über SQL-Abfragen, um CRUD-Operationen durchzuführen.

Testanforderungen

Testarten

1. Unit-Tests:

- Überprüfung der Kernfunktionen des Flask-Servers und der Datenbankinteraktionen.

2. Integrationstests:

- Sicherstellung der korrekten Kommunikation zwischen dem Flask-Server und der Dummy-Datenbank.

3. Lasttests:

- Durchführung von Tests mit Locust, um die Performance der Anwendung bei 100 gleichzeitigen Benutzern zu evaluieren.

Testumgebung

Die Tests werden in einer Docker-Umgebung durchgeführt, um Konsistenz zu gewährleisten.

Testwerkzeuge

- pytest: Wird für Unit- und Integrationstests verwendet.
- Locust: Wird für Lasttests eingesetzt.

Testkriterien

- Alle Unit-Tests müssen erfolgreich abgeschlossen werden.
- Integrationstests dürfen keine Fehler zurückgeben.
- Die Anwendung sollte bei Lasttests eine Antwortzeit von unter 500 Millisekunden bei 100 gleichzeitigen Benutzern aufweisen.

Rollen und Verantwortlichkeiten

1. **Projektleiter (Alexander Knüppel)**:
- Gesamtverantwortung für die Planung, Durchführung und Überwachung des Projekts.
- Koordination aller Aktivitäten und Zeitmanagement.
- Kommunikation mit den Projektbetreuern.

2. **Entwickler (Alexander Knüppel):**
- Implementierung des Flask-Servers, der API-Schnittstellen und der Dummy-Datenbankanwendung.
- Durchführung von Unit- und Integrationstests.

3. **Tester (Alexander Knüppel):**
- Durchführung von Lasttests mit Locust zur Evaluierung der Performance.
- Sicherstellung, dass alle funktionalen und nicht-funktionalen Anforderungen erfüllt sind.

4. **Projektbetreuer (Bumin Hatiboglu):**
- Beratung während des Projekts.
- Unterstützung bei der Klärung von Fragen und Herausforderungen.
- Teilnahme an der Endabnahme des Projekts.

5. **Softwarebetreuer (Sven Liekfeldt):**
- Unterstützung bei technischen Fragen und Herausforderungen.
- Feedback zur Softwarearchitektur und Implementierung.
- Teilnahme an der Endabnahme des Projekts.

6. **Endabnahme (Sven Liekfeldt, Bumin Hatiboglu, Alexander Knüppel)**

- Vorstellung der erstellten Lösung und dazugehöriger Dokumente durch Alexander Knüppel
- Beurteilung der Umsetzung erfolgt durch Projekt- und Softwarebetreuer
- Ein Abnahmeprotokoll wird erstellt, um die Ergebnisse der Endabnahme festzuhalten.

Abnahmekriterien

Kompatibilität zum Registry-Server:

Die Anwendung muss erfolgreich eine Kommunikation mittels REST-API gewährleisten um die Kompatibilität mit der Schnittstelle des Registry-Servers zu demonstrieren. Da dieser Registry-Server zum Abnahmezeitpunkt nicht zur Verfügung steht, gilt die erfolgreiche Kommunikation zwischen Lasttestmodul und Kommunikations-Server als Erfolgskriterium.
Dies ist das Hauptkriterium für die Abnahme.

Anpassbarkeit und Erweiterbarkeit:

Die Anwendung muss so gestaltet sein, dass sie anpassbar ist und zukünftige Erweiterungen problemlos integriert werden können. Dies stellt sicher, dass die Lösung eine breite Basis für zukünftige Entwicklungen bietet.

Einhaltung der Projektzeit:

Das Projekt muss innerhalb des festgelegten Zeitrahmens von maximal 80 Stunden abgeschlossen werden.

Verwendung von Open-Source-Lösungen:

Sofern möglich, müssen Open-Source-Lösungen verwendet werden, um Kosteneffizienz sowie Überprüf- und Wartbarkeit zu gewährleisten.

Sicherheitskonzept:

Obwohl ein umfassendes Sicherheitskonzept nicht vollständig implementiert sein muss, sollte die Anwendung grundlegende Sicherheitsmechanismen bieten und anpassbar sein für zukünftige Sicherheitsimplementierungen.

Dokumentation:

Alle erforderlichen Dokumentationen (z.B. API-Dokumentation) müssen erstellt und bereitgestellt werden.

25.11.2024 **Alexander Knüppel**

Abnahmeprotokoll

AP2-Projektarbeit: Containerisiertes Framework zur automatisierten Kommunikation zwischen multiplen Clients mit integriertem Lasttestmodul

Anwesende: **Datum: xx.11.2024**

- Alexander Knüppel (Projektleiter)
- Sven Liekfeldt (Softwarebetreuer)
- Bumin Hatiboglu (Projektbetreuer)

Vorstellung der Lösung:

Umsetzung eines Projektarbeit im Rahmen der AP2. Umgesetzt wurde eine Framework welches Kommunikation mittels REST-API und einer containerisierten Datenbank ermöglicht, um eine Umsetzbarkeit als Kommunikationsstruktur für eine Backup-Server-Architektur zu darzustellen und zu erleichtern.

Bewertung der Umsetzung:

1. Kompatibilität zum Registry-Server:
Die Anwendung gewährleistet eine erfolgreiche Kommunikation mittels REST-API, um die Kompatibilität mit der Schnittstelle des Registry-Servers zu demonstrieren. Da der Registry-Server zum Abnahmezeitpunkt nicht verfügbar ist, wurde die erfolgreiche Kommunikation zwischen dem Lasttestmodul und dem Kommunikations-Server als Erfolgskriterium herangezogen.

Entscheidung:

__ erfüllt **__ nicht erfüllt**

2. Anpassbarkeit und Erweiterbarkeit:
Die Anwendung wurde so gestaltet, dass sie anpassbar ist und zukünftige Erweiterungen problemlos integriert werden können.

Entscheidung:

__ erfüllt **__ nicht erfüllt**

25.11.2024 **Alexander Knüppel**

Einhaltung der Projektzeit:
Das Projekt wurde innerhalb des festgelegten Zeitrahmens von maximal 80 Stunden abgeschlossen.

Entscheidung:

__ erfüllt **__ nicht erfüllt**

Verwendung von Open-Source-Lösungen:
Es wurden Open-Source-Lösungen verwendet, um Kosteneffizienz sowie Überprüf- und Wartbarkeit zu gewährleisten.

Entscheidung:

__ erfüllt **__ nicht erfüllt**

Sicherheitskonzept:
Die Anwendung bietet grundlegende Sicherheitsmechanismen und ist anpassbar für zukünftige Sicherheitsimplementierungen, auch wenn ein umfassendes Sicherheitskonzept nicht vollständig implementiert sein muss.

Entscheidung:

__ erfüllt **__ nicht erfüllt**

Dokumentation:
Alle erforderlichen Dokumentationen (z.B. API-Dokumentation) wurden erstellt und bereitgestellt

Entscheidung:

__ erfüllt **__ nicht erfüllt**

Unterschrift:

Alexander Knüppel Sven Liekfeldt Bumin Hatiboglu

Begriffserläuterungen

1. JSON:
= Java Script Objekt Notation

2. Ramp-Up Rate:

Die Ramp-Up Rate bezeichnet die Geschwindigkeit, mit der die Anzahl der simulierten Benutzeranfragen während eines Lasttests erhöht wird. Sie ermöglicht es, schrittweise mehr Last auf das System zu bringen, um dessen Reaktion unter steigender Belastung zu beobachten. Durch eine kontrollierte Ramp-Up Rate können Tester potenzielle Engpässe oder Leistungsprobleme frühzeitig identifizieren und sicherstellen, dass das System nicht überlastet wird.