

**Projekt:** Verteilte Systeme Labor - API Harvester

**Dozenten:** Marius Erlen, Benjamin Salchow

**Modul:** Verteilte Systeme Labor

**Gruppenmitglieder:** Michael Schick, Marcel Hasselberg

**Studienjahrgang:** INF2022

**Inhaltsverzeichnis:**

[1. Einleitung 2](#_Toc177409610)

[1.1 Erklärung des Projekts 2](#_Toc177409611)

[1.2 Zielsetzung 2](#_Toc177409612)

[2. Architektur 3](#_Toc177409613)

[2.1 Anforderungen 3](#_Toc177409614)

[2.1.1 Funktional 3](#_Toc177409615)

[2.1.2 Nichtfunktional 3](#_Toc177409616)

[2.2 Allgemeine Architektur 4](#_Toc177409617)

[2.3 Auswahl der Technologien 4](#_Toc177409618)

[2.3.1 Python/Flask 4](#_Toc177409619)

[2.3.2 React 4](#_Toc177409620)

[2.3.3 Nginx 5](#_Toc177409621)

[2.3.4 PostgreSQL 5](#_Toc177409622)

[2.3.4 InfluxDB 5](#_Toc177409623)

[2.3.5 Redis 5](#_Toc177409624)

[2.3.6 Ofelia 5](#_Toc177409625)

[3. Umsetzung 6](#_Toc177409626)

[4.1 Komponentensicht 6](#_Toc177409627)

[3.2 Probleme während der Umsetzung 7](#_Toc177409628)

[3.2.1 Scheduler 7](#_Toc177409629)

[3.2.2 Systemleistung 8](#_Toc177409630)

[3.2.3 Programmcode zwischen den Containern aufteilen 8](#_Toc177409631)

[4. Reflexion 9](#_Toc177409632)

[4.1 Zukunftsausblick für spätere Projekte 9](#_Toc177409633)

[4.2 Größte Herausforderungen 9](#_Toc177409634)

[5. Quellen 10](#_Toc177409635)

# Einleitung

## 1.1 Erklärung des Projekts

Die Anwendung, welche im Umfang dieses Projektes entwickelt wurde, ermöglicht es Benutzern, verschiedenste API-Endpunkte zu Abonnieren. So können Benutzer beispielsweise Wetter oder Finanz API’s abonnieren und die Daten, welche zyklisch abgefragt werden, später einsehen. Die Anwendung bietet eine Benutzerregistrierung und einen Login-Bereich, über den sich Nutzer anmelden können. Nach dem Login können Benutzer aus einer Auswahl von APIs wählen und diese dann Abonnieren, sodass sie in gewünschten Intervallen abgefragt werden. Die Benutzeroberfläche ist dabei intuitiv gestaltet, sodass das Abonnieren und das Verwalten der Abonnements einfach möglich ist. Das System ist zudem flexibel aufgebaut, sodass neue APIs durch die Entwickler problemlos integriert werden können, um die Anwendung kontinuierlich zu erweitern.

## 1.2 Zielsetzung

Das Ziel dieses Projekts ist der Aufbau eines Systems, das den modernen Anforderungen eines verteilten Systems gerecht wird. Dabei spielt die Containerisierung der verschiedenen Komponenten mithilfe von Docker und Docker Compose eine zentrale Rolle, um die Anwendung flexibel und portabel zu gestalten. Zudem wird besonderer Wert auf Lastverteilung durch Load Balancer gelegt, um eine optimale Skalierung zu ermöglichen und Engpässe sowie Single Points of Failure zu vermeiden. Die Skalierung der Datenbanken wurde in diesem Fall bewusst außen vorgelassen.

# Architektur

## 2.1 Anforderungen

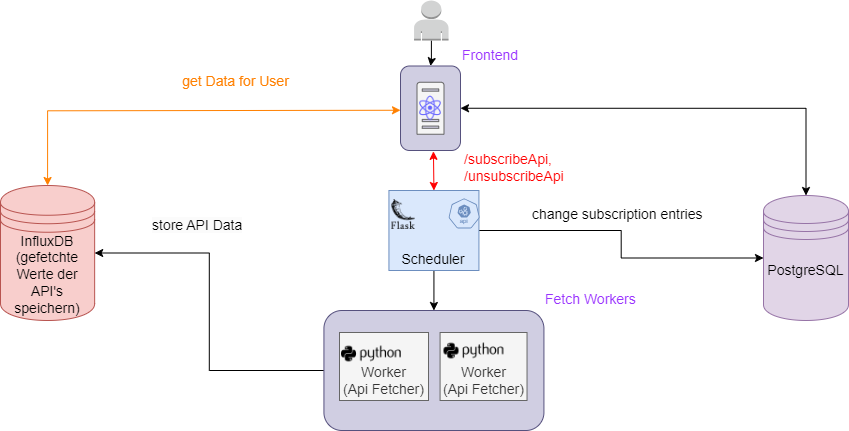
### 2.1.1 Funktional

|  |  |  |
| --- | --- | --- |
| Optional: | Anforderung | Beschreibung |
| Nein | Registrierungsmöglichkeit | Benutzer können sich registrieren und ein Admin Benutzer wird automatisch erstellt. |
| Nein | Rollenbasiertes Zugriffssystem | Implementierung eines Systems, das unterschiedliche Zugriffsrechte je nach Benutzerrolle gewährt |
| Ja | Premiumfunktionen für bestimmte Rollen | z.B. nur Premiumnutzer können alle verfügbaren Apis fetchen |
| Nein | Subscriben/Unsubscriben von APIS | Benutzer können API’s, um Daten zyklisch zu fetchen, abonnieren und auch wieder deabonnieren |
| Nein | Datenanzeige | Benutzer können die gesammelten Daten einsehen |
| Nein | Worker auslasten | Nach dem ein Container der API’s Abfragt voll ausgelastet ist, sollen weitere Aufgaben an den nächsten freien Container ausgelagert werden |

### 2.1.2 Nichtfunktional

|  |  |  |
| --- | --- | --- |
| Anforderung | Beschreibung | Messwert |
| Vollständig responsives Design | Gewährleistung, dass die Anwendung auf verschiedenen Geräten und Bildschirmgrößen optimal dargestellt wird. | - Anwendung mit den Entwicklertools des Browsers auf drei unterschiedlichen virtuellen Gerätetypen getestet |
| Betreibbarkeit in einem Container | Möglichkeit, die Anwendung in einem Container-Umgebung zu betreiben, um die Portabilität sowie die Skalierbarkeit zu verbessern. | - Auf mindestens 2 Plattformen getestet |
| Zuverlässigkeit | Sicherstellung, dass die Anwendung robust ist und potenzielle Fehler für den Benutzer ersichtlich sind | - Verfügbarkeit von min. 95% - Fehler werden ausgegeben |
| Dokumentation und Wartbarkeit des Codes | Bereitstellung einer ausführlichen Dokumentation und gut strukturierter Codebasis, um die Wartbarkeit zu gewährleisten. | - Stukturierter und kommentierter Code - Dokumentation |

## 2.2 Allgemeine Architektur



## 2.3 Auswahl der Technologien

### 2.3.1 Python/Flask

In diesem Projekt wurde Flask als Webframework ausgewählt. “Flask ist ein Python-Microframework [...]. Es wurde mit Blick auf die Geschwindigkeit entwickelt, sodass Sie sehr schnell mit der Entwicklung von Webanwendungen beginnen können.”[[1]](#footnote-2) Gerade das minimalistische Konzept, was es uns ermöglicht nachfolgende Komponenten sehr flexibel einzubauen war einer der Hauptgründe weshalb Flask gewählt wurde. Die bereits vorhandene Expertise im Team hat zudem den Einarbeitungsaufwand minimiert und zu erheblichen Zeitersparnissen geführt.

### 2.3.2 React

React wurde als Frontend-Framework ausgewählt und musste sich hierbei hauptsächlich gegen Angular beweisen. React ermöglicht die Verwendung von JavaScript oder Typescript zur Erstellung von Benutzeroberflächen. Außerdem erleichtert die Nutzung von Komponenten in React den Aufbau eines einheitlichen User Interfaces und fördert die Wiederverwendbarkeit von Logik, was Zeit einspart. Die weit verbreitete Nutzung von React garantiert umfangreiche Unterstützung durch eine große, aktive Community und eine Vielzahl von nützlichen Libraries und Tools. Für das Design der Anwendung wurde Bootstrap verwendet.

### 2.3.3 Nginx

Nginx wird in diesem Projekt für das Load-Balancing sowohl der Scheduler als auch des Frontends eingesetzt. Es verwendet das Round-Robin-Verfahren, um eingehende Anfragen gleichmäßig auf die aktiven Serverinstanzen zu verteilen, was die Lastenverteilung optimiert und die Verfügbarkeit erhöht.

### 2.3.4 PostgreSQL

PostgreSQL übernimmt die Verwaltung der Nutzerinformationen, die Speicherung der verfügbaren API-Endpunkte und die Verwaltung der Abonnements. Die Entscheidung für PostgreSQL basiert auf der vorhandenen Erfahrung im Team sowie den Vorteilen der Open-Source-Lizenz und der großen aktiven Community.

### 2.3.4 InfluxDB

InfluxDB wird verwendet, um die von den APIs abgefragten Werte zu speichern. Die Datenbank ist besonders auf die Speicherung und Abfrage von Zeitreihendaten ausgelegt, was sie ideal für diese Aufgabe macht. So benutzt Influx ein Zeilenorientiertes Datenspeicherungsformat „which provides a number of benefits for working with time series data. This includes improved data compression which makes it more cost effective when storing large amounts of data.“[[2]](#footnote-3)

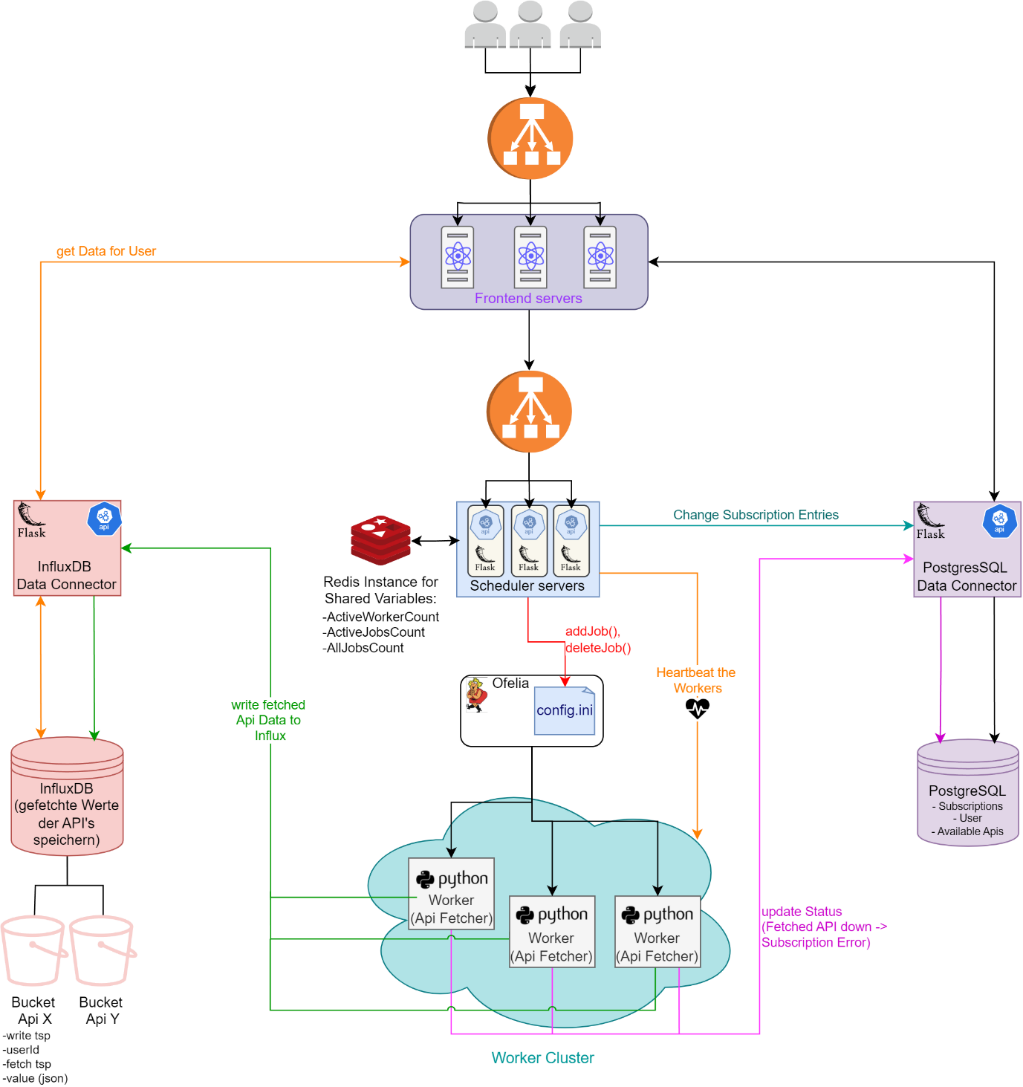
### 2.3.5 Redis

Redis wird eingesetzt, um die Konsistenz des Systems zu gewährleisten, indem es Daten zwischen den Containern verwaltet. Es wird insbesondere für die replizierten Scheduler verwendet, um gemeinsame Werte wie die Anzahl aktiver API-Abfrageinstanzen zu speichern und darauf zuzugreifen.

### 2.3.6 Ofelia

Als leichtgewichtiges Scheduling-Tool ermöglicht Ofelia die einfache Planung und Ausführung von Jobs innerhalb von Docker-Containern. Es bietet eine effiziente Möglichkeit zur Verwaltung und Verteilung wiederkehrender Aufgaben. Mit Ofelia ist es möglich die API-Abfrage Jobs zentral zu orchestrieren.

# Umsetzung

4.1 Komponentensicht

**Frontend**Das Frontend, welches auf Port 8080 läuft, wurde mithilfe von React und Bootstrap designt und implementiert. Um die Ausfallsicherheit zu gewährleisten und eine hohe Anzahl von Anfragen zu bewältigen, ist ein Load-Balancer vor die Frontend-Container geschaltet, welche in beliebiger Anzahl horizontal skaliert werden können. Dieser Nginx Load-Balancer verteilt die Anfragen mithilfe des Round-Robin-Verfahrens gleichmäßig. Über das replicas-Attribut in der docker-compose-Datei kann hierbei die Anzahl der Frontend-Replikate einfach geändert werden. Die neuen Replikate werden automatisch in den Load-Balancer integriert und von diesem berücksichtigt.

**Datenconnectoren (für InfluxDB/PostgresSQL)**

Die Daten-Connectoren bieten eine REST-API-Schnittstelle für die anderen Systemkomponenten an, sodass diese CRUD-Operationen auf den Datenbanken auslösen können. Um Sicherheitsaspekte zu berücksichtigen ist in diesen ein Schutzmechanismus eingebaut: Wenn ein Nutzer Daten benötigt, wird ein gültiger JWT-Token verlangt. Für eine Systemkomponente erfolgt die Authentifizierung hingegen über einen API-Key.

**Scheduler**Nachdem der User eine API über die GUI abonniert hat, nimmt der Scheduler diese Anfrage entgegen. Vor diesem Scheduler ist ähnlich wie im Frontend ein Load-Balancer geschaltet, der die Last auf mehrere Scheduler verteilen kann.  
Bei einer Aboanfrage überprüft der Scheduler, ob der Benutzer berechtigt ist diese Operation durchzuführen. Bei einem erfolgreichen Abonnier Vorgang macht der Scheduler einen Eintrag in der Datenbank und zudem einen Eintrag in der   
Ofelia-Konfigurationsdatei. Jene Einträge in der Konfigurationsdatei, werden dann von Ofelia ausgewertet, sodass die API-Abfrageaufgaben an die Worker Container ausgelagert werden können. Beim Eintragen in die Konfigurationsdatei stellt der Scheduler sicher, dass die Worker-Container vollständig ausgelastet werden, bevor ein neuer Container verwendet wird. Diese Vorgehensweise wurde bereits im Hinblick auf ein zukünftiges Autoscaling der Worker implementiert. Die Abfrageaufgaben werden zudem automatisch neu auf die Worker verteilt, sobald ein Benutzer eine API deabonniert oder ein Workercontainer keinen Heartbeat mehr an den Scheduler sendet. Durch dieses Konzept ist eine hohe Ausfallsicherheit gegeben und eine hohe Auslastung der Worker sichergestellt.

**Worker**Den Workern werden durch Ofelia zyklische API-Abfrageaufträge gegeben, welche dann von diesen ausgeführt werden. Die Anzahl der Worker bleibt während der Laufzeit immer konstant, kann jedoch vorher in einer Konfiguration festgelegt werden. Anschließend schreiben die Worker die gefetchten Daten über den Dataconnector in die Influx-Datenbank.

## 3.2 Probleme während der Umsetzung

### 3.2.1 Scheduler

Problem: Am Beginn des Projekts hatten wir den Plan das wir bei den Worker Containern ein Autoscalling einbauen, welches im Scheduler umgesetzt werden sollte. Hierbei hätte der Scheduler dann die Aufgabe gehabt diese Worker Container automatisch je nach Auslastung horizontal zu skalieren. In einem ersten PoC hat dies auch funktioniert, indem ein Docker CLI Container mit in eine sehr kleine Umgebung aufgenommen wurde. In der aktuellen Umgebung macht dies aber keinen Sinn. Nachdem der Scheduler an den CLI Container einen docker compose up --scale foo=x Befehl schicken würde, versucht dieser CLI Container die ganze Umgebung neu zu bauen. Dies würde erstmals viel zu lange dauern und auf Grund von bestimmten Abhängigkeiten zwischen den Containern auch nicht funktionieren.

Lösung: Fixe Anzahl an Worker Containern verwenden und die Aufgaben dann auf diesen verteilen.

### 3.2.2 Systemleistung

Problem: Das System kann aufgrund der hohen Anzahl an Containern viele Ressourcen vom Hostrechner verbrauchen. Wenn man keine von den replizierbaren Containern (Frontend, Scheduler, Worker) skaliert sind bereits 11 Container im Einsatz. Wenn man hier nun aber eine Skalierung vornimmt, hat die Umgebung im Standardfall 5 Worker-, 3 Frontend- und 3 Scheduler Container hat, sodass bereits 19 Container laufen würden. So kann der Hostrechner schnell an seine Leistungsgrenzen kommen. Vor allem wenn auf diesem parallel noch am System entwickelt wird, kommt es schnell vor, dass er an seine Leistungsgrenzen kommt.

Lösung: Die Ressourcen die Docker verwenden darf limitieren. Unter Windows (was in unserem Fall das Entwicklungsbetriebssystem ist), muss man hierfür die Ressourcen limitieren, welche die WSL oder HyperV verwenden darf.[[3]](#footnote-4)

### 3.2.3 Programmcode zwischen den Containern aufteilen

Problem: Um die Codequalität zu gewährleisten und dem DRY-Prinzip (Don't Repeat Yourself)[[4]](#footnote-5) zu folgen, sollte gemeinsamer Programmcode oder wiederkehrende Konstanten und Datenstrukturen zwischen den Containern geteilt werden. Dies funktionierte jedoch nicht, da Docker Compose standardmäßig den Build-Kontext so festlegt, dass nur Dateien und Verzeichnisse, die sich im gleichen Ordner wie das Dockerfile oder darunter befinden, verwendet werden können.

Lösung: Der Build-Kontext in der Docker-Compose-Datei wird auf das Root-Verzeichnis des Projekts gesetzt. Dadurch können die Dockerfiles weiterhin in den jeweiligen Verzeichnissen der einzelnen Komponenten liegen, indem man im Compose-File den Pfad zu den Dockerfiles zusätzlich zum Build-Kontext explizit angibt.

# Reflexion

## 4.1 Zukunftsausblick für spätere Projekte

Rückblickend würden wir in zukünftigen Projekten den Fokus zunächst auf die minimalen Funktionen legen, um möglichst schnell ein lauffähiges MVP (Minimum Viable Product) zu entwickeln. In diesem Projekt haben wir uns bereits früh viele Gedanken über Rollen, Rechte und zusätzliche Features, wie die Darstellung von Diagrammen gemacht. Dennoch sollte das Projekt so entwickelt werden, dass man sich keine Einschränkungen für die Weiterentwicklung schafft. Es ist wichtig, genügend Raum für Erweiterungen zu lassen und das Grundgerüst so aufzubauen, dass zusätzliche Funktionen später problemlos integriert werden können.

## 4.2 Größte Herausforderungen

Eine große Herausforderung waren die Architekturentscheidungen. Zum einen, weil wir ein Projekt dieser Art noch nie umgesetzt haben und zum anderen, weil wir teilweise unterschiedliche Standpunkte hatten. Um diese Herausforderung zu bewältigen, führten wir ausführliche Diskussionen und zeichneten Architekturdiagramme, um unsere unterschiedlichen Perspektiven zu visualisieren und ein gemeinsames Verständnis zu entwickeln. Zusätzlich testeten und implementierten wir Proof-of-Concepts einzelner Komponenten, um die bestmögliche Lösung zu finden.

# Quellen

[1] **Website The story: Flask vs. Django:**Veröffentlicht: 2024 [Zugriff: 14.09.2024]  
URL: <https://thestory.is/de/journal/flask-vs-django/>

[2] **Website influxdata: A Guide to the New InfluxDB Database Engine:**[Zugriff: 16.09.2024]URL: <https://www.influxdata.com/influxdb-storage>

[3] **Website docker.docs: Docker Desktop WSL 2 backend on Windows:**[Zugriff: 16.09.2024]URL: <https://docs.docker.com/desktop/wsl/>

[4] **Cremer, Marcel: Das DRY-Prinzip (Don't repeat yourself):**Veröffentlicht: 2023 [Zugriff: 16.09.2024]  
URL: https://www.influxdata.com/influxdb-storage

1. https://thestory.is/de/journal/flask-vs-django/ [↑](#footnote-ref-2)
2. https://www.influxdata.com/influxdb-storage [↑](#footnote-ref-3)
3. https://docs.docker.com/desktop/wsl/ [↑](#footnote-ref-4)
4. https://marcel-cremer.de/de/glossar/dry/ [↑](#footnote-ref-5)