



Diplomarbeit

Höhere Technische Bundeslehranstalt Leonding
Abteilung für Medientechnik

LeoCode

Eingereicht von: **Christian Donnabauer, 5AHTIM**

Datum: **April 21, 2021**

Betreuer: **Prof. Ing. Mag. Dr. Thomas Stütz**

Declaration of Academic Honesty

Hereby, I declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

This paper has neither been previously submitted to another authority nor has it been published yet.

Leonding, April 21, 2021

Christian Donnabauer

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorgelegte Diplomarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Gedanken, die aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Leonding, am 21. April 2021

Christian Donnabauer

Gender Declaration

In this diploma thesis the language form of the generic masculine is used for reasons of better readability. All personal designations are therefore to be understood as genderneutral.

Gender Erklärung

Aus Gründen der besseren Lesbarkeit wird in dieser Diplomarbeit die Sprachform des generischen Maskulinums verwendet. Alle personenbezogenen Bezeichnungen sind somit geschlechtsneutral zu verstehen.

Abstract

In the course of Christian Donnabauer's diploma thesis, an online learning platform is developed, which makes it easier for students of lower grades to get started with programming. By means of a task pool created by teachers, students can additionally prepare for tests. An automated server-side check of the submitted programs and code fragments provides users with immediate feedback on the correctness of their submissions. Independent practice is thus possible. The teaching staff is relieved of routine tasks in order to be able to provide better individual support for the students. Since the tasks are created by the teaching staff, integration into the current curriculum and adaptation to the respective learning progress is possible.

Zusammenfassung

Im Zuge der Diplomarbeit von Christian Donnabauer wird eine Online-Lernplattform entwickelt, welche Schülern der niederen Jahrgänge den Einstieg in die Programmierung erleichtert. Mittels einem von Lehrern erstellten Aufgabenpools können Schüler sich zusätzlich auf Tests vorbereiten. Durch eine automatisierte serverseitige Überprüfung der abgegebenen Programme und Codefragmente erhalten die Benutzer sofort ein Feedback über die Korrektheit ihrer Abgaben. Ein selbstständiges Üben ist damit möglich. Das Lehrpersonal wird von Routineaufgaben entlastet, um so die SchülerInnen besser individuell betreuen zu können. Da die Aufgaben vom Lehrpersonal erstellt werden, ist eine Integration in den aktuellen Lehrplan und eine Anpassung an den jeweiligen Lernfortschritt möglich.

Danksagung

Ich bedanke mich recht herzlich bei meinem Diplomarbeitsbetreuer, Prof. Stütz, der mir bei der Umsetzung der Diplomarbeit tatkräftig zur Seite stand. Welches Problem sich auch auftat, er konnte mich stets mit guten Ratschlägen unterstützen. Bereits bei der Themenfindung konnte er mir maßgeblich helfen.

Besonders möchte ich mich außerdem bei Hr. Prof. Aberger sowie bei Halil Bahar bedanken. Die mit Prof. Stütz gemeinsam erarbeiteten Lösungsansätze für die verschiedenen Problemstellungen, halfen mir dabei, diese immer so effizient wie möglich zu lösen. Ein großes Dankeschön geht außerdem an Halil, für die Hilfe bei der Implementierung der Status-Ansicht einer Abgabe.

Contents

1 Ausgangssituation und Zielsetzung	4
1.1 Ausgangssituation	4
1.2 Ist-Zustand	4
1.3 Beschreibung der Problemstellung	4
1.4 Marktanalyse	5
1.4.1 Codingame	5
1.4.2 Katacoda	5
1.4.3 EduTools	6
1.5 Aufgabenstellung	6
1.5.1 Gesamtkonzept	6
1.5.2 Aufgabenbereiche der vorliegenden Arbeit	6
1.5.3 Funktionale Anforderungen	6
1.5.4 Nichtfunktionale Anforderungen	7
1.6 Ziele	7
2 Systemarchitektur	9
2.1 Verschiedene Versionen	9
2.1.1 Version 1: Erster Prototyp	9
2.1.2 Version 2: Jenkins	11
2.1.3 Version 3: Jenkinsfile Runner	13
2.1.4 Version 4: Apache Kafka	14
2.2 Aktuelle Komponenten	15
2.2.1 Angular Frontend	15
2.2.2 Quarkus Backend	17
2.2.3 Kafka Broker	19
2.2.4 Testing-API	20
2.2.5 Jenkins und Jenkinsfile Runner	22
2.3 Mögliche zukünftige Erweiterungen	24
3 Schnittstellendefinition:	26
3.1 Beispiel	26
3.1.1 Erstellen eines Übungsbeispiels	26

3.1.2	Testen einer Abgabe	26
3.2	Rest-Endpoints	29
3.2.1	Create Example	29
3.2.2	Get Example By ID	30
3.2.3	List All Examples	31
3.2.4	Create Submission	32
3.2.5	Portfolio	33
3.3	Submission Status Endpoint	34
4	Ausgewählte Aspekte	36
4.1	Messaging mit Apache Kafka	36
4.1.1	Event Streaming	36
4.1.2	Was ist Apache Kafka?	36
4.1.3	Grundbegriffe	37
4.1.4	Beispiel anhand zweier Quarkus Instanzen	37
5	Resümee	44
6	Anhang	45
6.1	Besprechungsprotokolle	45
7	Quellenverzeichnis	55

Chapter 1

Ausgangssituation und Zielsetzung

1.1 Ausgangssituation

Die HTL Leonding ist eine Höhere Technische Lehranstalt im oberösterreichischen Zentralraum mit ca. 1000 Schülern und den Fachabteilungen Medientechnik, Informatik, Medizintechnik sowie Elektronik.

1.2 Ist-Zustand

In allen Zweigen spielt das Programmieren eine mehr oder weniger große Rolle, besonders in den Zweigen Medientechnik bzw. Informatik. Momentan nutzt das Lehrpersonal verschiedenste Plattformen wie zum Beispiel Github-Classroom oder Moodle, um die Abgaben der Schüler gesammelt zu erhalten.

1.3 Beschreibung der Problemstellung

Programmieren ist ein sehr anspruchsvoller Gegenstand. Viele Schüler haben Probleme nur durch die Übungen im Rahmen des Unterrichts eine positive Note zu erreichen. Üben die Schüler außerhalb des Unterrichts, wäre ein rasches Feedback zur Korrektheit der erstellten Programme oder Hinweise zu den Fehlern wünschenswert. Auch zusätzlich Übungsbeispiele werden oftmals von den Schülern verlangt.

Ein Nachteil der Notenfindung durch primär punktuelle Leistungsfeststellungen (Tests) ist es, die Bemühungen der Schüler in der Vorbereitung (zum Test) nicht in genügendem Maße berücksichtigen zu können. Es wäre wünschenswert, die durch die Schüler geübten Beispiele in einem sogenannten Portfolio dokumentieren zu können und so dem Lehrpersonal eine zusätzliche Beurteilungsgrundlage zur Verfügung zu stellen.

Haben die Lehrer Übungsabgaben erhalten, so müssen diese auf Korrektheit und der genauen Umsetzung überprüft werden. Diese Überprüfung ist sehr zeitintensiv und kann je nach Aufgabenstellung selbst für einzelne Abgaben stundenlang dauern. Die Lehrer müssen allerdings Abgaben von ganzen Klassen überprüfen, was in den unteren Jahrgängen teilweise bis zu 30 Schüler bzw. Abgaben sein können.

1.4 Marktanalyse

Ziel der Marktanalyse ist es festzustellen, ob bereits fertige Plattformen/Tools zur Verfügung stehen, um autonomes Üben im Bereich Programmieren zu ermöglichen. Die berücksichtigten Kriterien sind:

- Kosten: Wie teuer ist es, diese Plattform zu verwenden?
- Erstellung von eigenen Übungsbeispielen: Können Lehrer eigene Übungsbeispiele erstellen? Wie detailliert können Beispiele konfiguriert werden?
- Datenschutz: Werden die persönlichen Daten der Schüler vertraulich behandelt?
- Portfoliofunktion: Wird eine Dokumentation der gesamten Übungsleistung unterstützt?
- Prüfungsfunktion: Ist es möglich einfache Überprüfungen wie zum Beispiel Lernzielkontrollen durchzuführen?
- Einfachheit der Bedienung: Wie komplex ist die Nutzeroberfläche aufgebaut? Ist eine eigene Entwicklungsumgebung notwendig?

1.4.1 Codingame

[12] Die Website codingame.com ermöglicht es Nutzern anhand von kleineren Programmieraufgaben spielerisch gegeneinander anzutreten. Die Verwendung ist zwar kostenlos und eine Portfoliofunktion sowie ein Online Editor sind gegeben, allerdings müssen Übungsbeispiele durch Nutzung der sogenannten tech.io Plattform extern erstellt werden. Des Weiteren ist keine genaue Information bzgl. Datenschutz oder die Möglichkeit von Prüfungen gegeben.

1.4.2 Katacoda

[10] Katacoda ist eine vom Verlag O'Reilly entwickelte Plattform, welche verschiedenste sogenannten Szenarios anbietet. Ein Szenario entspricht im Prinzip ein Übungsbeispiel, welche in verschiedenste Ausführungen angeboten wird. Grundsätzlich bietet Katacoda kostenlos die Portfoliofunktion zbd Erstellung eigener Szenarios. Allerdings bleibt der Datenschutz Aspekt fraglich. Diese Plattform ist allerdings mehr für Cloud Lösungen als Programmiersprachen per se gedacht.

1.4.3 EduTools

[6] EduTools ist eine IDE Erweiterung aus dem Hause Jetbrains. Diese Erweiterung kann direkt in der IDE verwendet werden und bietet Beispiele für Beginner durch die Stepik Plattform. Stepik kann genutzt werden um selbst erstellte Beispiele mit Schülern oder anderen Lehrern zu teilen. Auch wenn diese Erweiterung kostenlos ist, ist sie auf die Nutzung von Jetbrains Produkten angewiesen. Eine Portfoliofunktion wird nicht unterstützt.

Aus der Marktanalyse geht vor, dass zwar bereits verschiedene Lösungen bestehen, diese allerdings alle schwer auf den Lehrplan anzupassen sind.

1.5 Aufgabenstellung

1.5.1 Gesamtkonzept

Nach Durchführung der Marktanalyse wurde beschlossen, dass es sinnvoll ist, ein modulares System unter dem Namen LeoLearn zur erstellen. Das System LeoLearn beinhaltet mehrere Module, welche in der folgenden Abbildung erkennbar sind. Diese verschiedenen Module werden durch eine gemeinsame Benutzeroberfläche dargestellt.

1.5.2 Aufgabenbereiche der vorliegenden Arbeit

Im Rahmen der vorliegenden Diplomarbeit wird der Teilbereich LeoCode erstellt.

- Erstellung des Softwaresystems (Konzeption, Programmierung)
 - Backend: automatisierte Tests durchführen und Feedback generieren
 - Frontend: möglichst intuitive Programmierumgebung bzw. Upload Funktion für Programmbeispiele
- Operating - Erstellen eines lauffähigen Systems
 - Leichtgewichtige Virtualisierung (Docker)
 - Continuous Integration (CI) mit Automatisierungsserver (Jenkins)

LeoCode ist so konzipiert, dass das Training verschiedenster Programmiersprachen möglich ist. In der vorliegenden Arbeit wird jedoch nur eine Programmiersprache (Java) prototyphaft berücksichtigt.

1.5.3 Funktionale Anforderungen

Zu entwickeln ist eine Plattform welche... :

- ... es Schülern ermöglicht, Code-Beispiele zu programmieren
- ... es Schülern nach Abgabe ihrer Programmcodes Feedback, Lösungshinweise bzw. die Lösung bereitstellt

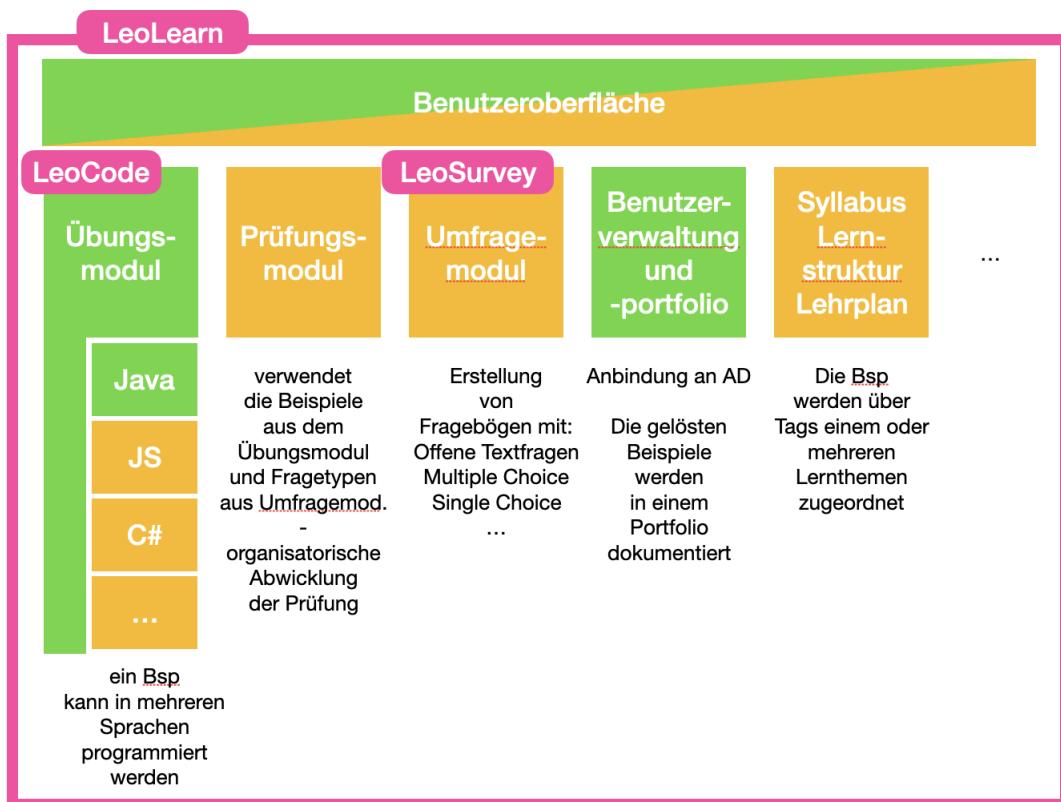


Figure 1.1: Struktur LeoLearn

- ... es dem Lehrpersonal ermöglicht, Beispiele inklusive Tests zu erstellen

1.5.4 Nichtfunktionale Anforderungen

Das Projekt LeoCode ist als Langzeitprojekt konzipiert und wird in späteren Ausbaustufen um andere Sprachen und eventuell weitere Funktionalitäten erweitert.

1.6 Ziele

- Schüler können Beispiele bearbeiten, welche ihrem Können entsprechen und werden so durch kleine Erfolgsergebnisse motiviert, selbstständig weiter zu üben.
- Das Lehrpersonal spart bei der Erstellung, der Korrektur und beim Feedback viel Zeit. Diese Zeitersparnis kann genutzt werden, um die individuelle Förderung der Schüler zu verstärken.
- Durch das zentrale Beispielsrepository wird ein defacto-Standard für den Unterricht geschaffen. Das Lernniveau der parallelen Klassen wird so vereinheitlicht.

- Schüler können außerhalb des Unterrichts zusätzliche Übungen machen.
- Neulehrer werden beim Gestalten ihres Unterrichts unterstützt, da sie auf das Übungsbeispiel-Repository zugreifen können.

Chapter 2

Systemarchitektur

2.1 Verschiedene Versionen

Im Laufe der Realisierung der vorliegende Arbeit, wurde die Systemarchitektur schrittweise verbessert, weshalb verschiedene Versionen entstanden sind. Um den Verlauf nachzuvollziehen, werden im folgenden Kapitel verschiedene Versionen sowie Begründungen zu Änderungen angeführt.

2.1.1 Version 1: Erster Prototyp

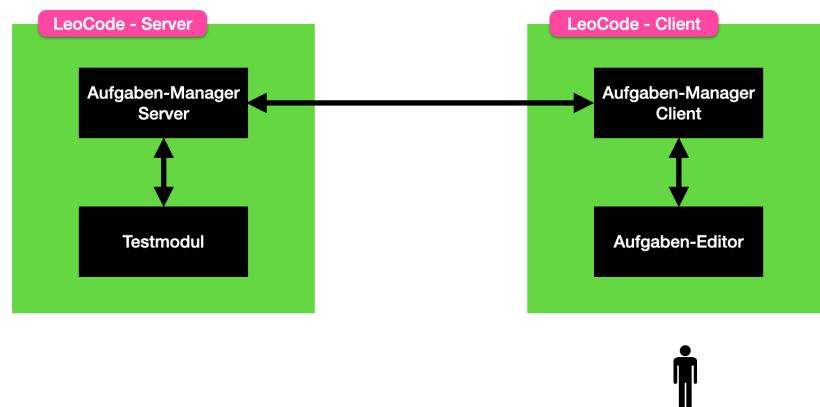


Figure 2.1: Kommunikation der Komponenten LeoCode V1

Bereits beim Projekt-Kickoff-Meeting wurde beschlossen, Angular als Frontend und eine JakartaEE microprofile Implementierung (Quarkus) als Backend zu verwenden. Sowohl Angular, als auch der JakartaEE-Standard (ehemalig Java EE) mit Quarkus als Implementierung, sind in der Industrie weit verbreitet und kostenfrei zu nutzen. Als

Datenbank wurde Postgres ausgewählt, die sich durch ihre Stabilität und ebenfalls freie Verfügbarkeit auszeichnet. Als besonderer Vorteil von Quarkus ist die einfache Integration in Cloud Architekturen zu erwähnen.

Die folgende Graphik stellt die geplante Systemarchitektur zu Projektbeginn dar:

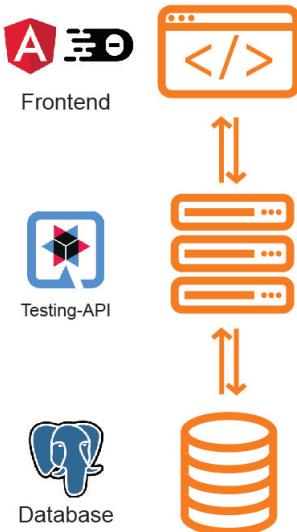


Figure 2.2: Systemarchitektur LeoCode V1

Die Testing-API ist ein Quarkus Server, welcher sowohl für das User-Management, als auch für das Testen von Projekten zuständig ist. Nutzer müssen die Files (Java-Sourcecode und Unittests) per Multipart-Form einzeln hochladen und können danach mittels erneuten Rest Request die Abgaben testen lassen. Die abgegebenen Java-Projekte werden in einem Ordner am Quarkus Server zwischengespeichert, wonach die automatisierten Tests (jUnit) mittels Maven durchgeführt werden.

2.1.2 Version 2: Jenkins

Um das Erweitern für anderer Programmiersprachen zu erleichtern, wurde die Testing-API in einem generellen Backend Server und einem Testing-API Server aufgeteilt.

Zu den Aufgaben des allgemeinen Backend Server zählen unter anderem das User-Management und vor allem die Kommunikation mit der Datenbank. Der Testing-API Server hingegen ist lediglich für das Testen von (Java-)Programmcodes zuständig. Somit wird eine Trennung des LeoLern-Managementsystems und dem Evaluationssystem für die Programmcodes erreicht. Wird eine zusätzliche Testing-API (das Evaluationssystem) in einer anderen Programmiersprache zB C# erstellt, können Aufgaben auch in dieser Programmiersprache erstellt werden.

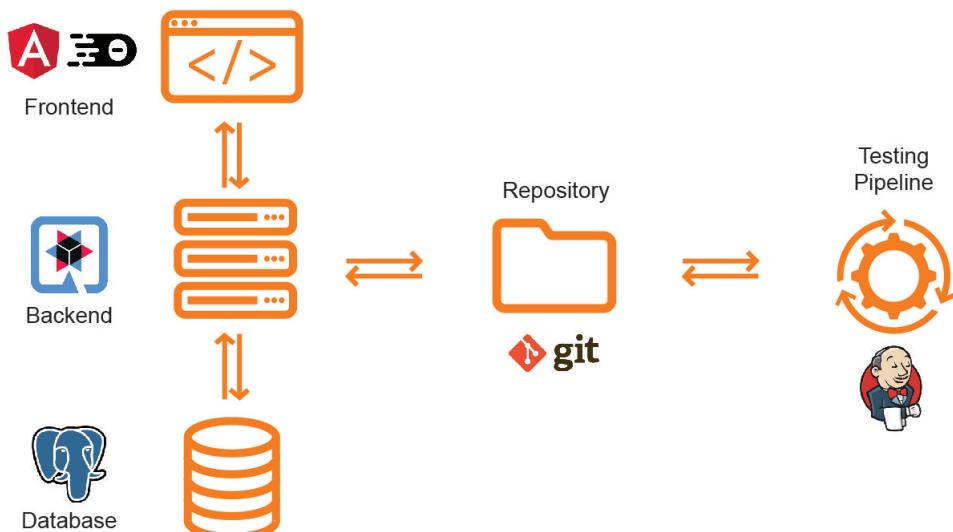


Figure 2.3: Systemarchitektur LeoCode V2

Damit die abgegebenen Java-Projekte zur Testing-API gelangen, wird erneut auf eine Rest-Schnittstelle zurückgegriffen. Lädt der Benutzer sein Projekt hoch, werden die nötigen Files erneut per Multipart-Form Fileupload zum Testing-API Server mittels eines Rest Clients des Backend Servers weitergeleitet.

Um die Vorteile eines Automatisierungsservers zu nutzen und die Abgaben in einer gesonderten Laufzeitumgebung zu testen, wurde nun außerdem Jenkins verwendet. Die genauen Vorteile eines Jenkins werden im Kapitel **Jenkins und Jenkinsfile Runner** genauer thematisiert.

Damit die Build-Prozesse automatisch gestartet werden konnten, wurde ein Git Repository verwendet. In diesem Git Repository wird eine Webhook ausgeführt, dh ein Jenkins-Build wird gestartet sobald das Java Projekt durch die Testing-API auf das Remote Repository gepusht wird.

Der eingezeichnete Theia Editor ist eine IDE, welche innerhalb des Browsers läuft. Nutzer sparen sich also jegliches Setup (zum Beispiel das Installieren einer IDE, JDK, ...) bei der Nutzung des auf Visual Studio Code basierenden Editors. Der Editor Theia soll es ermöglichen, die Übungen in einem Online-Editor erstellen bzw. testen zu können um so (besonders bei den ersten Aufgaben) auf eine IDE verzichten zu können.

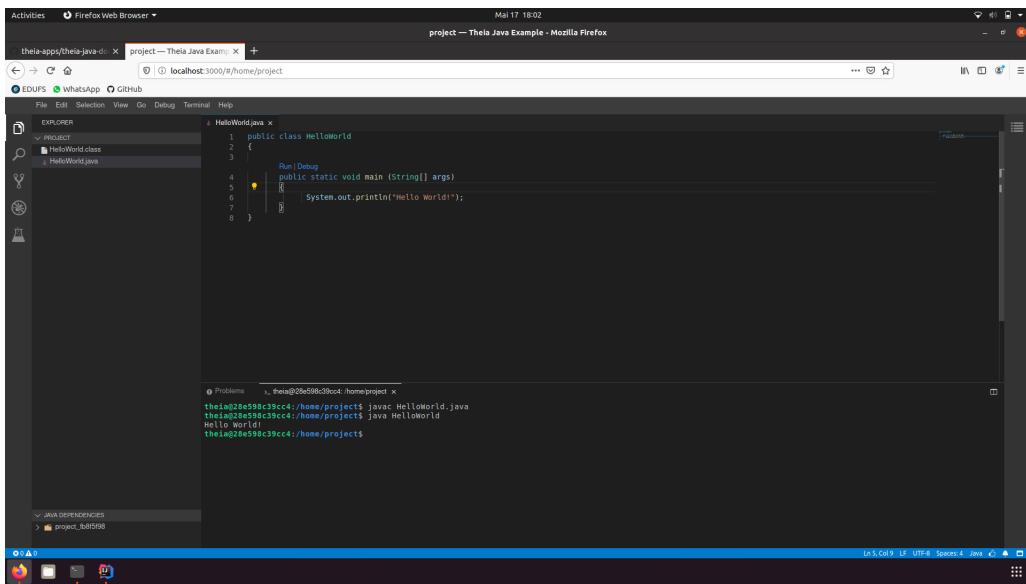


Figure 2.4: Online Editor Theia

2.1.3 Version 3: Jenkinsfile Runner

Da der Lösungsansatz, einen Jenkins-Build mittels eines Git Repositories zu starten, nicht für genügend Abgaben geeignet ist, wurde die Systemarchitektur erneut umgestellt und der Jenkinsfile Runner verwendet. Wie der Name bereits vermuten lässt, ermöglicht dieser das Starten von Jenkins-Pipelines. So bleiben die Vorteile eines Automatisierungsservers erhalten, ohne ein Git-Repository zu verwenden.

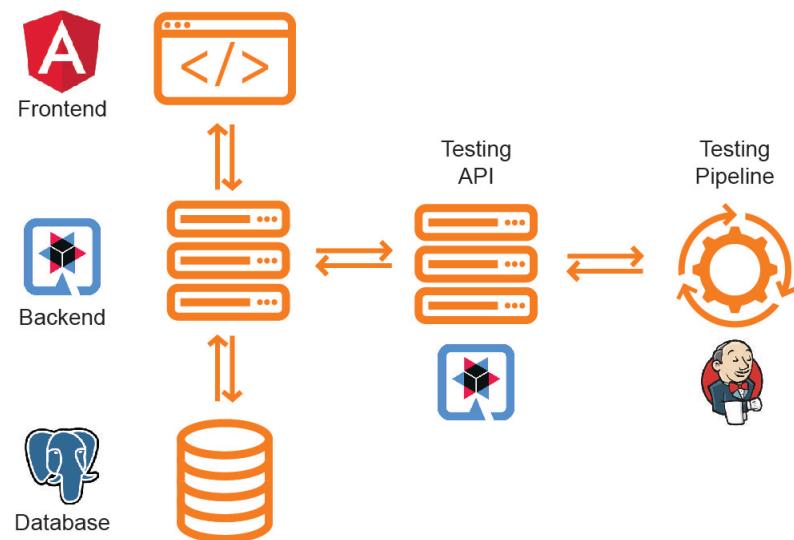


Figure 2.5: Systemarchitektur LeoCode V3

Wie zuvor bereits erwähnt sind detailliertere Informationen dazu im Kapitel **Jenkins und Jenkinsfile Runner** zu finden.

2.1.4 Version 4: Apache Kafka

Auch wenn der Testprozess bereits funktionierte, fehlte noch eine genügend performante Rückmeldung für den Benutzer. Um dies zu ermöglichen und gleichzeitig den Backend Server von der Testing-API zu entkoppeln, wurde entschieden, ein Messaging System zur Kommunikation zwischen Testing-API und Backend Server zu verwenden. Als Messaging System wird in der vorliegenden Arbeit Apache Kafka eingesetzt.

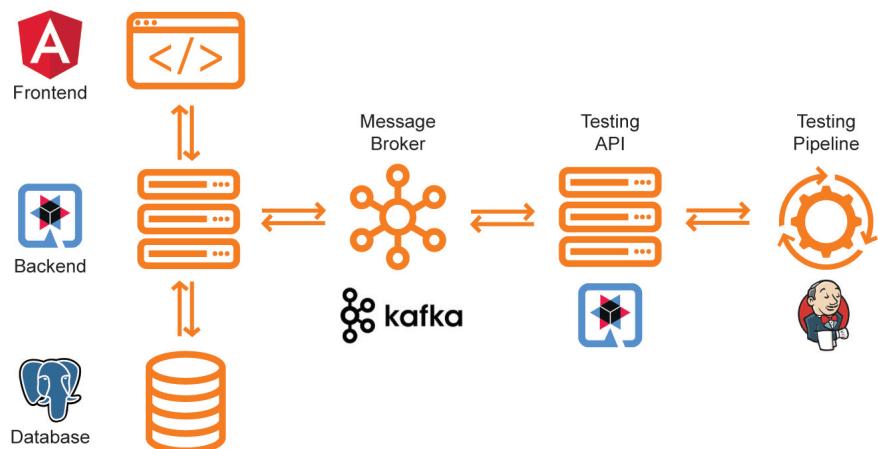


Figure 2.6: Systemarchitektur LeoCode V4

2.2 Aktuelle Komponenten

Das aktuell bestehende System (*Version 4: Apache Kafka*) besitzt eine Vielzahl an verschiedenen Komponenten. Deren Aufbau, Funktionsweise, sowie Zusammenhänge werden im folgenden Kapitel genauer behandelt.

2.2.1 Angular Frontend

Folgendes Kapitel wurde unter Verwendung der Quellen [2] bearbeitet.

Das Hauptgewicht der vorliegenden Arbeit liegt an der Entwicklung des durchaus komplexen Backends. Das Frontend wurde daher nur mit der grundlegenden Funktionalität ausgestattet. Dennoch wurde unter Verwendung von Angular Material ein recht ansehnlicher Prototyp erstellt.

Angular Material ist eine UI-Komponentenbibliothek, welche Entwickler beim Erstellen von gestalterisch ansprechenden Projekten hilft. In Kombination mit Bootstrap können so, sehr effizient, attraktive Webseiten entwickelt werden. Der Angular Client der vorliegenden Arbeit bietet unter anderem folgende Schlüsselfunktionen:

- Das Erstellen von Aufgabenstellungen:

The screenshot shows the LeoCode application interface. On the left is a sidebar with a 'Menu' button at the top, followed by 'Examples', 'Test Code', 'Create', and 'Profile'. The main area has a blue header bar with the text 'LeoCode'. Below the header is a form for creating a new example. The fields are as follows:

- Name of the Author: T. Stuetz
- Name of the Example: Hello World
- Type: Maven
- Short Description of the Example: The aim of the following Example is to print Hello World
- Instruction File: instruction.md (with a 'Browse...' button)
- Zip file (containing a pom.xml, correct Solution & Unittests): HelloWorld.zip (with a 'Browse...' button)
- Jenkinsfile: Jenkinsfile (with a 'Browse...' button)
- Whitelist: (empty input field)
- Blacklist: (empty input field)

At the bottom of the form is a blue 'Create Example' button.

Figure 2.7: Erstellen einer Aufgabe

- Die Auflistung der Aufgabenstellungen:

Name	Description	Type
Hello World	The aim of the following Example is to print Hello World	MAVEN
Fakultaet	The aim of the following example is to create a console program, which calculates the fakultaet	MAVEN

Items per page: 50 < > 1 - 2 of 2

[Create Example](#)

Figure 2.8: Auflistung der Aufgabenstellungen

- Die Detailansicht einer Aufgabenstellung:

ID	Name	Description	Type	Files
6	Hello World	The goal of this example is to create a Maven Project, which prints Hello World!	MAVEN	instruction.md pom.xml HelloWorld.java HelloWorldTest.java Jenkinsfile

[Test](#)

Figure 2.9: Detailansicht der Aufgabenstellung

- Das Erstellen von Abgaben:

Username: i160174

Your Sourcecode:

[Test Your Code](#)

Figure 2.10: Erstellen einer Abgabe

- Die Statusansicht einer Abgabe:

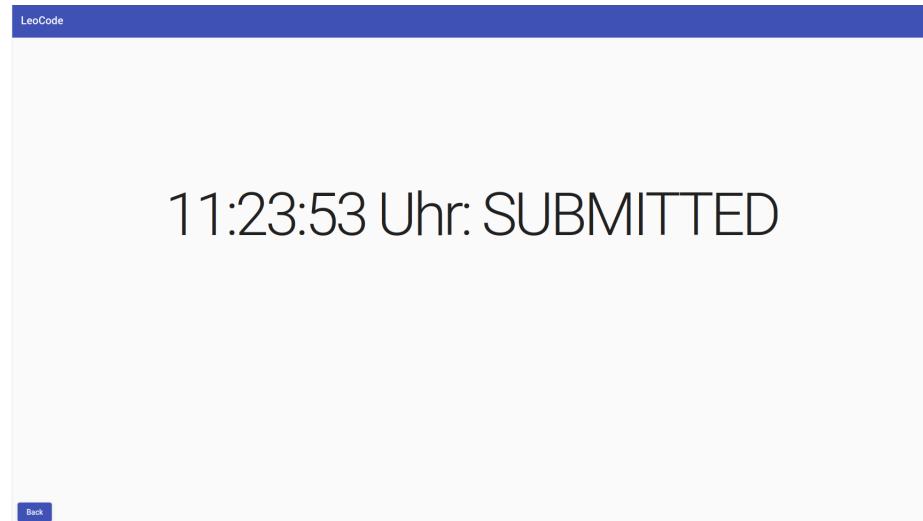


Figure 2.11: Statusansicht einer Abgabe

- Der Verlauf von Abgaben je Nutzer:

Menu	LeoCode			
	ID	Status	Result	Date
Examples	13	CORRECT	Finished: SUCCESS	11:25 17/04/2021
Create	14	FAILED	Finished: FAILURE	11:27 17/04/2021

Items per page: 50 < > 1 – 2 of 2

Figure 2.12: Abgabe Verlauf eines Nutzers

Als Schnittstellen zum Backend dienen hauptsächlich Rest-Endpoints, allerdings wurde zur Einsicht des aktuellen Abgabe Status auch mit Server Sent Events gearbeitet.

2.2.2 Quarkus Backend

Der allgemeine Backend Server spielt vor allem folgende entscheidende Rollen:

1. **Schnittstelle zum Frontend:** Wie bereits erwähnt, dienen mehrere Rest Endpoints als Schnittstellen zum Angular Client. Um mittels Angular Client nun zum Beispiel eine Aufgabe zu erstellen, muss dieser einen POST Request mit Multipart-Form Payload auf den zuständigen Endpoint stellen. Ist dieser Request erfolgreich, wird ein Json Objekt zurückgeliefert welches in etwa so aussieht:

```

1  {
2    "id": 6,
3    "author": "Lehrer 1",
4    "description": "The goal of this example is to create a Maven
      Project, which prints Hello World! ",
5    "name": "Hello World",
6    "type": "MAVEN"
7  }

```

Dieses Objekt spiegelt eine Instanz der DAO-Klasse des Backends wieder. Das sogenannte Data Access Object (DAO) Pattern wird allgemein dafür verwendet, dass der Applikations-Layer (zum Beispiel jene Objekte, welche im Angular Client verwendet werden) vom Persistenz-Layer (zum Beispiel die JPA-Objekte des Backends) getrennt werden kann.

Die eigentliche Example Klasse enthält vor allem noch JPA relevante Daten, die allerdings vom Angular Frontend nicht benötigt werden, weshalb letztlich auf die Verwendung vom DAO-Pattern zurückgegriffen wurde.

2. **Kommunikation mit der Datenbank:** Um mit der Postgres Datenbank zu kommunizieren wurde Hibernate ORM mit Panache in Kombination mit dem so genannten Repository Pattern verwendet. Das Repository Pattern ermöglicht, ähnlich zum DAO Pattern, Lese- und Schreibvorgänge auf logischer Ebene zu definieren. Im Vergleich zum DAO Pattern, liegt dieses allerdings auf einen höheren Level, näher zur Geschäftslogik. Im konkreten Fall der vorliegenden Arbeit, also in der Kombination mit Panache, erlaubt es die Panache spezifischen Methoden von den Objekt Klassen auszulagern.



Figure 2.13: Funktionsweise Repository Pattern

Hibernate ORM ist die am meisten verbreitetste JPA Implementierung. Auch wenn dies zwar detailliertes, konfigurierbares und komplexes OR-Mapping erlaubt, ist die Verwendung nicht wirklich einfach bzw. trivial. Panache hilft dabei, eben diese Problematik zu vermeiden, indem Entities und Zugriffe möglichst einfach und ohne Boilerplate-Code erstellt werden können.

Die Ausarbeitung dieses Kapitel wurde mithilfe der Quellen [4] [5] [1] [14] gemacht

3. **Kommunikation mittels Apache Kafka:** Nach Abgabe eines Projektes müssen zunächst die Abgegebenen Dateien extrahiert, kategorisiert und persistiert werden. Anschließend wird mittels Apache Kafka ein Submission Objekt vom Backend Server an die Testing-API gesandt. Dieses Submission Objekt enthält essentielle Daten wie zum Beispiel den Dateipfad zu den Projektdaten oder dessen aktuellen Status.

Grundsätzlich hatte der Backend Server auch die Rolle der Nutzerverwaltung, es wurde allerdings beschlossen die Nutzerverwaltung auf einen Schulserver auszulagern, da dies sonst den Umfang der vorliegenden Arbeit sprengen würde und die Anmeldung ohnehin über die Schulaccounts erfolgen soll.

2.2.3 Kafka Broker

Der Kafka Broker ist für die Kommunikation zwischen Backend und Testing-API Server zuständig. Außerdem entkoppelt er die beiden Server voneinander. Mittels Kafka zu kommunizieren ist außerdem höchst performant und ausfallsicher. Was Apache Kafka allerdings genau ist und für was es eingesetzt wird, wird im Kapitel *Messaging mit Apache Kafka* näher behandelt.

Wie zuvor bereits erwähnt, senden der Backend sowie der Testing-API Server Submission Objekte, per Kafka. Ein mögliches Submission Objekt, welches vom Backend Server gesendet wird, sieht in etwa so aus (der Lesbarkeit halber als Json formatiert):

```

1 {
2   "id": 7,
3   "pathToProject": "../../projects-in-queue/project-under-test-7.zip",
4   "author": "it160174",
5   "result": "",
6   "status": CREATED,
7   "lastTimeChanged": "2021-04-18T00:47:04.014190",
8   "example": 6
9 }
```

Im Anschluss an den Testprozess sendet wiederum die Testing-API das angepasste Submission Objekt zurück, welches danach vom Backend ausgewertet und an den Nutzer weitergeleitet wird. Beispiel für ein Submission Objekt nach dem Durchlaufen der Tests:

```

1 {
2   "id": 7,
3   "pathToProject": "../../projects-in-queue/project-under-test-7.zip",
4   "author": "it160174",
5   "result": "Finished: SUCCESS",
6   "status": CORRECT,
7   "lastTimeChanged": "2021-04-18T00:48:11.724037",
8   "example": 6
9 }
```

2.2.4 Testing-API

Damit Abgaben von Schüler getestet werden können, wird die Testing-API benötigt. Dieser Quarkus Server ist für das notwendige Test-Setup zuständig. Außerdem überprüft er nach korrektem Setup die Blacklist bzw. Whitelist Funktion und startet im Anschluss daran, wenn nötig, die Jenkins Pipeline. Nach Abschluss eines Test-Durchlaufes wertet der Server außerdem die Ergebnisse aus und sendet diese anschließend an den allgemeinen Backend Server. Nach diesem recht groben Überblick, folgt nun eine etwas ausführlichere Beschreibung, der einzelnen Aufgaben:

1. **Erstellung des Testsetups:** Nach Eintreffen eines Submission Objektes wird zu Beginn überprüft, ob das notwendige Testsetup besteht. Ist dies nicht der Fall, wird dieses erstellt oder eine entsprechende Fehlermeldung zurückgesendet. Das notwendige Testsetup umfasst unter anderem das zu testende Projekt, sowie ein gesondertes Dateiverzeichnis, in welchem das Projekt anschließend weiterbearbeitet werden kann.
2. **Entpacken des zu testenden Projektes:** Um Speicherplatz zu sparen werden die zu testenden Projekte gezipped in ein Dateiverzeichnis, auf welches beide Server Zugriff haben, zwischengespeichert. Dementsprechend muss die Testing-API um ein Projekt zu Testen zunächst die Dateien entpacken. Die Dateien werden anschließend im während des Setups erstellten Testverzeichnis abgespeichert. [8]
3. **Überprüfung der Keywords:** Die Lehrperson kann optional Keywords vorgeben, welche verwendet werden müssen (Whitelist) oder Keywords, welche nicht verwendet werden dürfen (Blacklist). Die Benennung der beiden Listen wurde unter Umständen etwas unglücklich gewählt. Es empfiehlt sich diese in Zukunft umzubenennen, beispielsweise in Allow- bzw. Blocklist.

Im Grunde können beide Funktionalitäten sehr ähnlich überprüft werden. In dem der Schülercode nach Keywords abgesucht wird, wird je nach Blacklist oder Whitelist Keyword, das Auffinden bzw. nicht Auffinden des jeweiligen Keywords, als Fehler bzw. als keiner gewertet. Hat das zu testende Projekt diese Überprüfung nicht bestanden, werden die nächsten Schritte übersprungen und direkt das entsprechend veränderte Submission Objekt an das Backend gesandt. Bei Blacklist-Fehlern wird die Zeilennummer zurückgegeben, in der das Keyword verwendet wurde, während bei Whitelist Fehlern das fehlende Keyword angegeben wird. Im Fall eines Blacklist Fehlers, würde das zurückgesandte Submission Objekt etwa so aussehen:

```

1  {
2    "id": 7,
3    "pathToProject": "../../projects-in-queue/project-under-test-7.
4      zip",
5    "author": "it160174",
6    "result"= "Blacklist Error: package has been used at line 1!",
7    "status"= FAILED,
8    "lastTimeChanged"= "2021-04-18T11:20:41.228739",
9    "example"= 6
9  }

```

4. **Erstellen der Projektstruktur:** Da im Backend nur einzelne Files in der Datenbank abgespeichert werden, enthalten die bei der Abgabe eines Schülers erstellten Zip-Archive, noch nicht die Projektstruktur eines klassischen Maven Projektes. Dementsprechend muss die Testing-API vor dem Testen das Projekt in die richtige Struktur bringen.

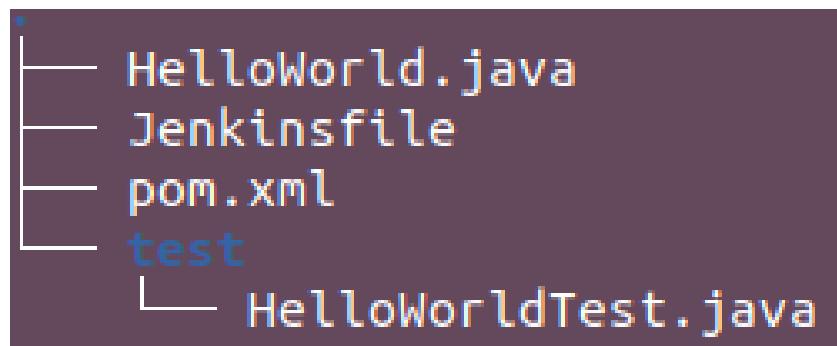


Figure 2.14: von Backend erstellte Zip-File Projektstruktur

Grundsätzlich reicht der Dateityp um zwischen pom.xml, Jenkinsfile und Java-Files zu unterscheiden. Um allerdings in der Testing-API zwischen Test- bzw. Code-Files unterscheiden zu können, werden die Testfiles vom Backend in einem Unterordner "test" platziert. Daraus kann die Testing-API dann ableiten, von welchem Typ eine Datei genau ist und an welchen Platz sie im generierten Maven Projekt sein soll.

5. **Starten der Tests:** Um nun den eigentlichen Testlauf zu starten, wird mittels Jenkinsfile Runner und dem vom Lehrer angegebenen Jenkinsfile eine Jenkins-Pipeline aufgebaut. Detailliertere Informationen dazu sind im Kapitel **Jenkins und Jenkinsfile Runner** zu finden.
6. **Evaluieren des Ergebnisses:** Um nun das Ergebnis des Testvorgangs zu bestimmen, muss der Konsolen Output der Jenkins Pipeline analysiert werden. Das danach zurückgelieferte Submission Objekt enthält im Attribut "result", eine detaillierte Information warum ein Testlauf evtl. fehlgeschlagen ist. Zusätzlich be-

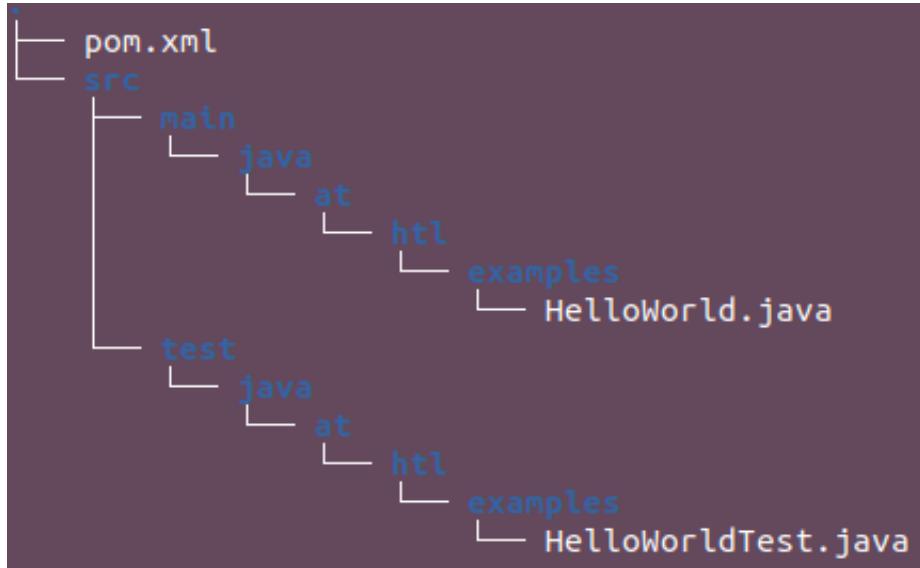


Figure 2.15: Beispiel für Maven Projektstruktur

sitzt dieses Objekt außerdem noch das "status" Attribut, welches unter anderem für die Statusansicht des Angular Client als SSE benötigt wird.

2.2.5 Jenkins und Jenkinsfile Runner

Um die Abgaben von Schülern zu testen, wird ein Jenkins Automationsserver verwendet. Typischerweise wird Jenkins für Continous Integration verwendet. Das Ziel von Continous Integration ist die Verbesserung einer bestimmten Software, durch die kontinuierliche Integration bzw. Testung von neuer oder veränderter Funktionalität. Dabei muss jede Softwareversion lauffähig sein. In der Praxis wird diese automatisierte Integration erst durch ein Versionskontrollsystem, wie zum Beispiel Git ermöglicht. Verzeichnet dieses System eine Änderung des Sourcecodes, also zum Beispiel einen neuen Commit, wird automatisch ein Jenkins-Build getriggered, welche das System inklusive der Änderungen testet. Macht so ein Entwickler beispielsweise einen Fehler, kann er diesen gleich im Jenkins-Build sehen, was das System insgesamt weniger fehleranfällig macht.

Im vorliegenden Fall besteht allerdings kein Versionierungssystem um die Abgaben von Schüler zu testen, weshalb der sogenannte Jenkinsfile Runner verwendet wird. Der Jenkinsfile Runner verpackt die Jenkins Pipeline Execution Engine als CLI Tool. Wie der Name unschwer vermuten lässt ermöglicht dieser es also Jenkinsfiles von der Command Line auszuführen.

Um nun die Abgaben von Schüler so zu testen, wurde ein lokaler Jenkins aufgesetzt, welcher anschließend von der Testing-API aus mit dem Jenkinsfile Runner die Projekte testet. Da auf dem lokalen Jenkins das sogenannte Docker Pipeline Plugin installiert wurde, können innerhalb eines Jenkinsfile Docker-Container als Laufzeitumgebungen

verwendet werden. Ein Beispiel für ein Jenkinsfile, welches ein Maven Projekt testet, sieht wie folgt aus:

```
1 pipeline {
2     agent {
3         docker {
4             image 'maven:3-alpine'
5         }
6     }
7     stages {
8         stage('Test') {
9             steps {
10                 sh 'mvn test'
11             }
12         }
13     }
14 }
```

Da nun Docker-Container zum Testen verwendet werden, kann Leocode in Zukunft mit Leichtigkeit um weitere Programmiersprachen erweitert werden. Um eine neue Programmiersprache zu unterstützen, muss der Testing-API Server lediglich die notwendige Projektstruktur erstellen und das angegebene Jenkinsfile entsprechend verändert werden.

Dieses Kapitel wurde unter der Verwendung von den Quellen [] [9] ausgearbeitet.

2.3 Mögliche zukünftige Erweiterungen

Docker ist eine Technologie, welche das Erstellen von sogenannten Images ermöglicht. Ein Docker Image ist im Prinzip eine fertige Konfiguration von Programmen. Wird eine Instanz eines Images erstellt, spricht man von einem Docker Container. Im Prinzip erfüllen Docker Container ähnliche Funktionen wie Virtuelle Maschinen, allerdings benötigen Container weniger Speicherplatz und Rechenleistung. Außerdem laufen Docker Container nicht direkt auf dem Host System, sondern auf der sogenannten Docker Engine. Die Engine verwaltet die Zuteilung von Ressourcen des Hosts auf die verschiedenen Containern.

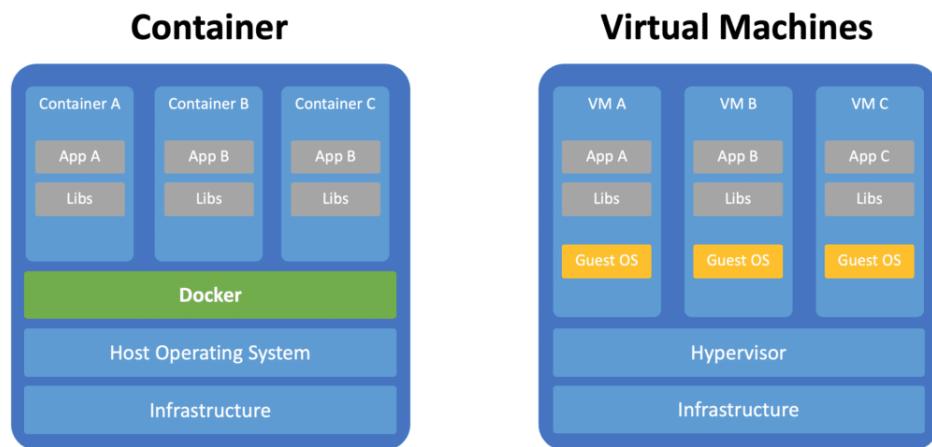


Figure 2.16: Docker vs. Virtuelle Maschinen

Aufgrund von schnelleren Startzeiten, sowie Ressourcenersparnissen wird Docker vor allem für Cloud Computing eingesetzt. Die zuvor genannten Vorteile erlauben eine flexiblere Skalierung, welche wiederum zu geringeren Wartezeiten für die Nutzer führt. Die vorliegende Software wurde allerdings bis jetzt noch nicht gedockert, was für ein Deployment in einer Cloud allerdings von essentieller Wichtigkeit ist.

Die Dockerisierung von LeoCode stellt aber nicht nur zeitlich eine Herausforderung da. Wie zuvor behandelt startet momentan die Testing-API mittels des Jenkinsfile Runners einen Jenkins Build. Da die Testing-API bei einer gedockerten Lösung allerdings selbst innerhalb eines Containers laufen würde, müsste diese einen neuen Container starten müssen. Der Jenkins-Container muss anschließend erneut einen Docker Container starten, damit Jenkins bei der Testung Docker verwenden kann.

Dieses Kapitel wurde mithilfe der Quellen [7] kreiert.

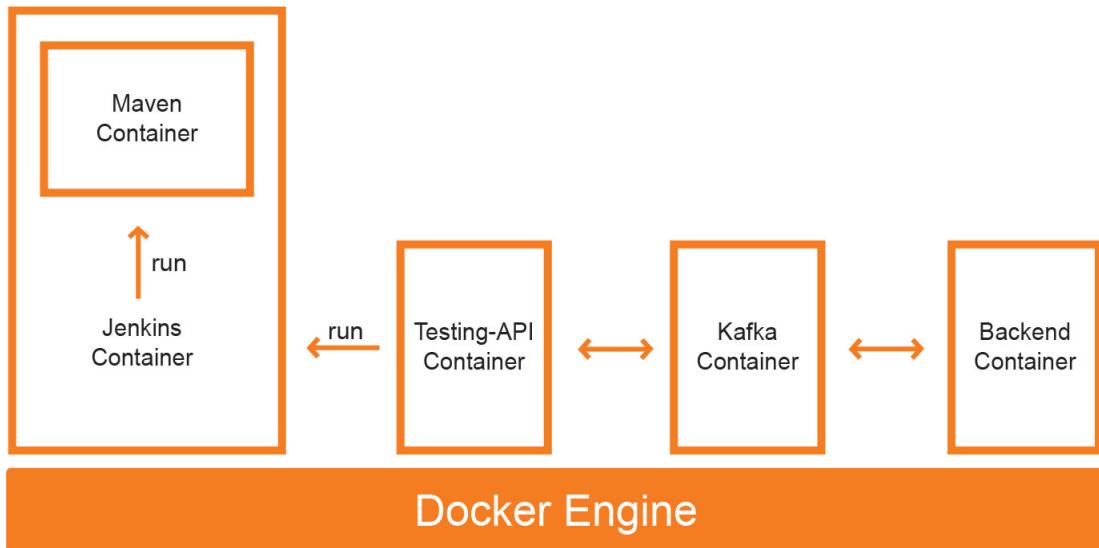


Figure 2.17: Docker in Docker Systemarchitektur

Die vorliegende Arbeit kann aber auch auf eine andere Art mittels Docker umgesetzt werden. Da eine Docker in Docker Variante hinsichtlich des Cloud Deployments oft zu Problemen führen kann, steht auch eine andere Variante zur Auswahl.

Bei dieser Variante wird für jede unterstützte Programmiersprache ein eigener Jenkinsfile-Runner Container erstellt. Dieser Container enthält neben dem Jenkinsfile Runner auch das Setup für die jeweilige Programmiersprache wie beispielsweise eine JDK im Fall von Java. Der Jenkinsfile Runner wird dabei innerhalb des Containers mittels eines Shellskriptes mit Endlosschleife gestartet und „darauf warten“, einen „Auftrag“ von der Testing-API zu erhalten. Da auf den jeweiligen Container ein passendes Setup besteht, kann auf die Verwendung des Docker-Pipeline Plugins verzichtet werden und so die damit verbundenen Komplikationen vermieden werden.

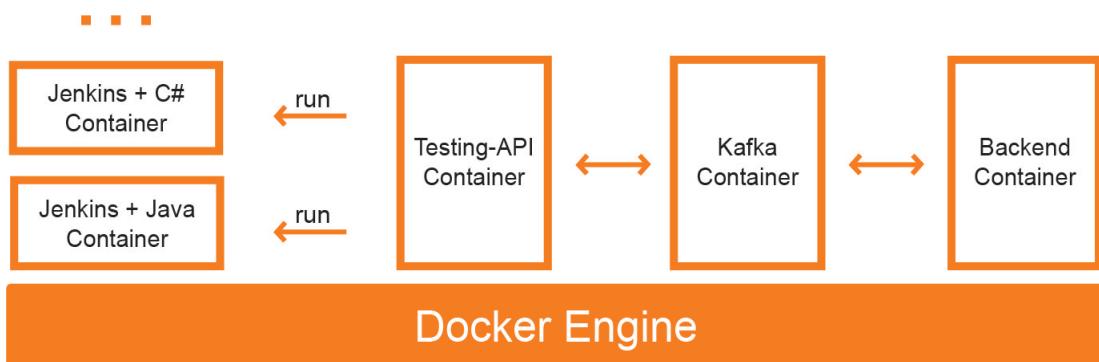


Figure 2.18: Docker Systemarchitektur

Chapter 3

Schnittstellendefinition:

3.1 Beispiel

Auch wenn die Funktionsweise der einzelnen Komponenten aus dem vorhergehenden Kapiteln bereits bekannt ist, folgt nun ein genaues Beispiel um die Kommunikation zwischen den Komponenten noch besser nachvollziehen zu können. Im nächsten Kapitel wird zuerst ein Übungsbeispiel angelegt und danach getestet.

3.1.1 Erstellen eines Übungsbeispiels

Um ein Übungsbeispiel zu erstellen wird lediglich ein POST Request an das Backend gesendet. Wie genau dieser aussieht ist im Kapitel (*Create Example*) zu finden. Das unter anderem konsumierte Zip-File wird entpackt und die einzelnen Files kategorisiert bzw. persistiert. Im Anschluss wird daraus ein Example Objekt erstellt, welches ebenfalls in der Datenbank abgespeichert wird.

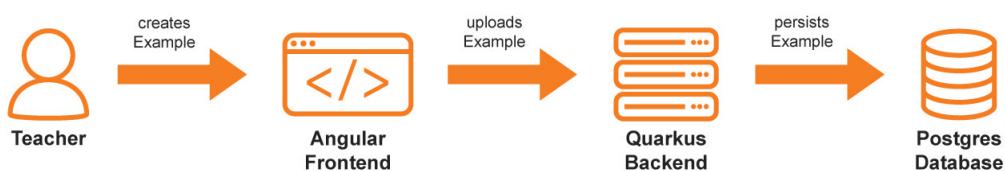


Figure 3.1: Teacher LeoCode: Erstellen eines Beispiels

3.1.2 Testen einer Abgabe

Das Testen eines Beispiels ist hingegen etwas komplexer. Um den Code abzugeben, senden Schüler mittels des Angular Frontends einen *Create Submission* Request. Anhand der Example-ID des empfangenen Submission Objektes, werden die zum Testen benötigten Files (pom.xml, Unitests, Jenkinsfile) aus der Datenbank abgefragt. An-

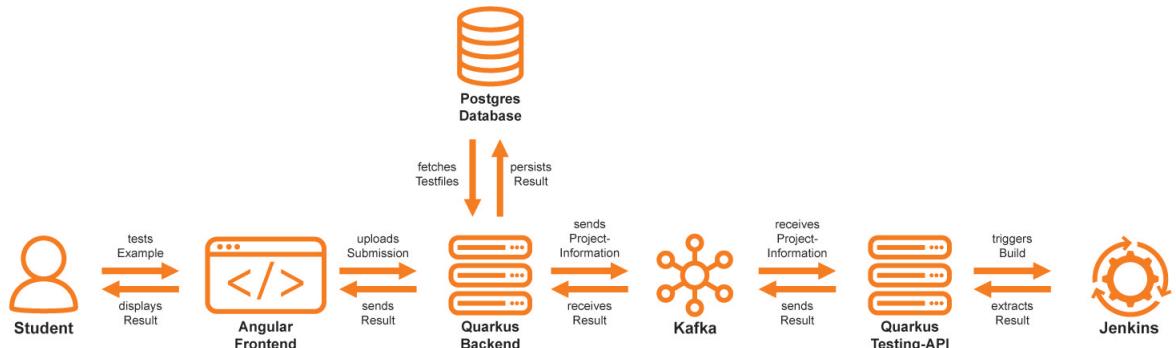


Figure 3.2: Student LeoCode: Abgabe eines Beispieles

schließend werden die Files in die folgende Struktur gebracht und als Zip-File komprimiert.

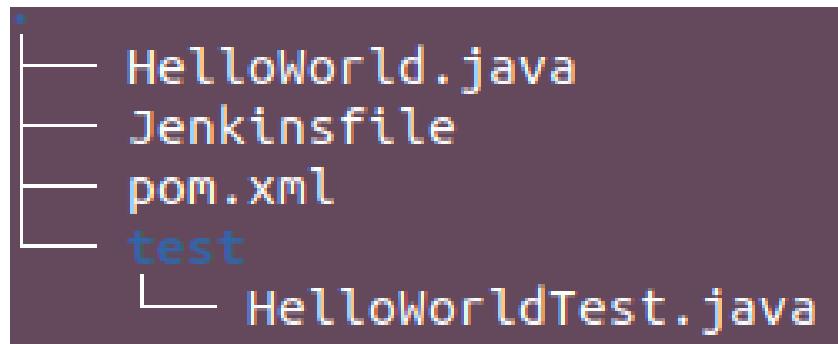


Figure 3.3: von Backend erstellte Zip-File Projektstruktur

Die erstellten Zip-Files werden danach in einem zuvor definierten Verzeichnis zwischengespeichert, auf welches sowohl Backend, als auch Testing-API Zugriff haben. Unmittelbar darauffolgend wird folgendes Submission Objekt mittels Kafka an die Testing-API gesendet.

```

1  {
2      "id": 7,
3      "pathToProject": "../../projects-in-queue/project-under-test-7.zip",
4      "author": "it160174",
5      "result": "",
6      "status": CREATED,
7      "lastTimeChanged": "2021-04-18T00:47:04.014190",
8      "example": 6
9  }
  
```

Die Testing-API startet nach Empfangen eines Submission Objektes den Testprozess. Der Testprozess beginnt mit dem notwendigen Setup der Files, sobald die Abgabe die Überprüfungen der Keywords (Blacklist bzw. Whitelist) übersteht. Anschließend wird mittels Jenkinsfile Runner eine Jenkins Pipeline gestartet (detailliertere Infos dazu sind im Kapitel **Jenkins und Jenkinsfile Runner** zu finden). Ist dieser Jenkins Build fertig, muss anhand des Build Outputs noch das Ergebnis evaluiert werden, wonach anschließend das angepasst Submission Objekt per Kafka zurückgesendet wird.

```
1  {
2      "id": 7,
3      "pathToProject": "../../projects-in-queue/project-under-test-7.zip",
4      "author": "it160174",
5      "result": "Finished: SUCCESS",
6      "status": CORRECT,
7      "lastTimeChanged": "2021-04-18T00:48:11.724037",
8      "example": 6
9  }
```

Empfängt der Backend Server eine fertig getestete Submission, speichert er diese in der Datenbank. Fragt ein Nutzer nun diese Submission mittels des **Submission Status Endpoint** ab, wird der Status der Submission als Server Sent Event zurückgeliefert.

3.2 Rest-Endpoints

3.2.1 Create Example

Um ein Code-Beispiel zu erstellen, muss ein POST Request an den Create Example Endpoint gestellt werden. Der unter dem Pfad **POST http://localhost:9090/example** erreichbare Endpoint konsumiert dabei einen Multipart Upload, welcher wie folgt auszusehen hat:

Name	Content-Type	Erklärung	Wert Beispiel
exampleName	text/plain	Der Benutzername des Beispiels	Hello World
username	text/plain	Der Name des Beispield-Author	T. Stuetz
description	text/plain	Eine Kurzbeschreibung des Beispiels	Das Ziel diese Beispiels ist es ...
exampleType	text/plain	Der Typ des Beispiels (momentan nur Maven unterstützt)	MAVEN
jenkinsfile	application/octet-stream	Das Jenkinsfile, welches zum Testen von Jenkins verwendet wird	pipeline { ... }
project	application/zip	Ein Zip File des gelösten Beispiels. Dieses Projekt muss eine Java Sourcecode, Unitests und eine pom.xml enthalten.	-
instruction	text/markdown	Die genaue Angabe des Beispiels	# UE1: Hello World ...
whitelist	text/plain	Wörter, welche verwendet werden müssen (optional)	StringBuilder
blacklist	text/plain	Wörter, welche nicht verwendet werden dürfen (optional)	equals

Das dabei zurückgelieferte Json Objekt enthält das erstellte Beispiel und könnte wie folgt aussehen:

```
1  {
2    "id": 6,
3    "author": "Lehrer 1",
4    "description": "The goal of this example is to create a Maven
      Project, which prints Hello World! ",
5    "name": "Hello World",
6    "type": "MAVEN",
7    "blacklist": [],
8    "whitelist": []
9 }
```

Im Angular Frontend wird dieser Request wie folgt abgebildet:

The screenshot shows a web application interface titled 'LeoCode'. On the left, there is a vertical 'Menu' sidebar with options: 'Examples', 'Test Code', 'Create', and 'Profile'. The main content area is titled 'Create Example'. It contains several input fields and file upload buttons:

- Name of the Author: T. Stuetz
- Name of the Example: Hello World
- Type: Maven
- Short Description of the Example: The aim of the following Example is to print Hello World
- Instruction File: Browse... Instruction.md
- Zip file (containing a pom.xml, correct Solution & Unittests): Browse... HelloWorld.zip
- Jenkinsfile: Browse... Jenkinsfile
- Whitelist: (empty)
- Blacklist: (empty)

At the bottom right of the form is a blue 'Create Example' button.

Figure 3.4: Create Example

3.2.2 Get Example By ID

Mittels GET Request an diesen Endpoint, können die Details eines bereits erstellten Beispiele abgefragt werden. Dieser Endpoint benötigt als einzigen Parameter, wie der Name bereits vermuten lässt, die **ID** eines Examples. Angenommen es besteht ein Beispiel mit der ID "6", erhält man durch den Request **GET http://localhost:9090/example/{id}** beispielsweise folgenden Response:

```

1   {
2     "id": 6,
3     "name": "Hello World",
4     "type": "MAVEN",
5     "blacklist": [
6       ],
7     "whitelist": [
8       ],
9     "files": [
10      {
11        "name": "instruction.md",
12        "fileType": "INSTRUCTION",
13        "content": "# This would be the Instruction File of an Example,
14          written by teachers"
15      }, # ... pom.xml, Jenkinsfile, Unitests
16    ]
17  }

```

Im Frontend wird dieser Endpoint für die Detailansicht eines Beispieles benötigt: Im Angular Frontend wird dieser Request wie folgt abgebildet:

ID	Name	Description	Type	Files
6	Hello World	The goal of this example is to create a Maven Project, which prints Hello World!	MAVEN	instruction.md pom.xml HelloWorld.java HelloWorldTest.java Jenkinsfile

Figure 3.5: Get Example By ID

3.2.3 List All Examples

Dieser Rest-Endpoint listet alle bestehenden Beispiele auf. Der unter **GET http://localhost:9090/examples** erreichbare Endpoint benötigt keine Parameter und liefert ein Array aus Beispielen. Ein möglicher Json Response würde in etwa so aussehen:

```

1  [
2    {
3      "id": 6,
4      "author": "T. Stuetz",
5      "blacklist": [],
6      "description": "The goal of this example is to create a Maven
          Project, which prints Hello World! ",
7      "name": "Hello World",
8      "type": "MAVEN",
9      "whitelist": []
10    },
11    {

```

```

12      "id": 12,
13      "author": "T. Stuetz",
14      "blacklist": [],
15      "description": "The aim of this project is to calculate the
16          Fakultaet of a given Number",
17      "name": "Fakultaet",
18      "type": "MAVEN",
19      "whitelist": []
20  }, ...

```

Dieser Endpoint wird beispielsweise im Frontend aufgerufen, um die verschiedenen Beispiele aufzulisten:

The screenshot shows a web application interface titled 'LeoCode'. On the left, there is a vertical sidebar with a 'Menu' section containing links for 'Examples', 'Test Code', 'Create', and 'Profile'. The main content area is titled 'Examples' and displays a table with two rows. The columns are 'Name', 'Description', and 'Type'. The first row has 'Hello World' in the Name column, 'The aim of the following Example is to print Hello World.' in the Description column, and 'MAVEN' in the Type column. The second row has 'Fakultaet' in the Name column, 'The aim of the following example is to create a console program, which calculates the fakultaet' in the Description column, and 'MAVEN' in the Type column. At the bottom of the table, there are buttons for 'Create Example' and navigation controls.

Name	Description	Type
Hello World	The aim of the following Example is to print Hello World.	MAVEN
Fakultaet	The aim of the following example is to create a console program, which calculates the fakultaet	MAVEN

Figure 3.6: Auflistung der Aufgabenstellungen

3.2.4 Create Submission

Testet ein Schüler ein Beispiel, wird ein sogenanntes Submission Objekt erstellt. Dieses Objekt entspricht also der Abgabe eines Schülers und wird unter dem Pfad **POST http://localhost:9090/submission** erstellt. Dieser Endpoint konsumiert ähnlich zum Create Example Endpoint einen Multipart Upload, welcher in diesem Fall wie folgt aussieht:

Name	Content-Type	Erklärung	Wert Beispiel
username	text/plain	Der Benutzername des Schülers	it160174
example	text/plain	Die ID des Beispiels, welches die Schüler zu lösen probieren.	6
code	text/x-java	Der Code des Schülers	package at.htl...

Als Response wird lediglich die ID der erstellten Submission zurückgeliefert, da für genauere Infos bezüglich des Abgabe Status der **Submission Status Endpoint** verwendet wird. Dieser Endpoint wird im Angular Frontend wie folgt verwendet:

Figure 3.7: Create Submission

3.2.5 Portfolio

Der Portfolio Endpoint gibt alle Submissions, die ein bestimmter Nutzer getätigt hat zurück. Der unter dem Pfad **GET http://localhost:9090/submission/history/{username}** erreichbare Endpoint benötigt als Parameter einen **Nutzernamen**, um den Submissionverlauf eines bestimmten Nutzers zurückzuliefern. Beispielsweise würde der Rückgabewert für den Nutzer "it160174" aus einem Submission Array bestehen, welches wie folgt aussehen könnte:

```

1  [
2    {
3      "id": 13,
4      "lastTimeChanged": "02:56 20-04-2021",
5      "result": "Finished: SUCCESS",
6      "status": "CORRECT"
7    },
8    {
9      "id": 20,
10     "lastTimeChanged": "03:09 20-04-2021",
11     "result": "Blacklist Error: package has been used at line 1!",
12     "status": "FAILED"
13   }
14 ]

```

Die Portfolio-Funktion sieht im Angular Frontend in etwa so aus:

Id	Status	Result	Date
13	CORRECT	Finished: SUCCESS	11:25 17-04-2021
14	FAILED	Blacklist Error: package has been used at line 1!	11:27 17-04-2021

Figure 3.8: Portfolio

3.3 Submission Status Endpoint

Um den aktuellen Status einer Abgabe darzustellen, wurden sogenannte Server Sent Events verwendet. Server Sent Events bieten die Möglichkeit, dass Clients, welche ein Verbindung zu einem Server aufgebaut haben von diesem auch benachrichtigt werden. In einem praktischen Beispiel heißt das, dass ein Client ohne eine Website neuzuladen Updates erhalten kann.

In der vorliegenden Arbeit wurde diese Technologie so eingesetzt, dass Schüler automatische Statusupdates ihrer Abgaben erhalten. Der Testprozess einer Abgabe eines Schülers dauert im Schnitt eine Minute. Ist kein Fehler währenddessen aufgetreten, befindet sich die Submission im Status "SUBMITTED". Ist die Abgabe fertig getestet, wird der Schüler automatisch benachrichtigt. Ist die Abgabe korrekt wird ebenfalls der Status "CORRECT" zurückgegeben, ist dies nicht der Fall, ist der Submission Status "FAILED". Der einzige Parameter der dafür benötigt wird ist die ID der jeweiligen Submission.

Dieser Endpoint verbirgt allerdings zusätzlich noch die Besonderheit, dass Clients, welche Submission abfragen, die bereits fertig getestet sind, den Status aus der Datenbank erhalten. Dies wird realisiert, indem der Backend Server zuerst die Datenbank nach der abgefragten Submission durchsucht. Ist der Status der Submission noch "SUBMITTED", merkt sich der Server die Submission und benachrichtigt den Client bei Änderungen. Ist der Status der Submission allerdings nicht "SUBMITTED" bedeutet das, dass der Testprozess bereits abgeschlossen ist. Unabhängig vom Testergebnis, wird dieses anschließend zurückgegeben.

Das Angular Frontend verwendet diesen Endpoint wie folgt:

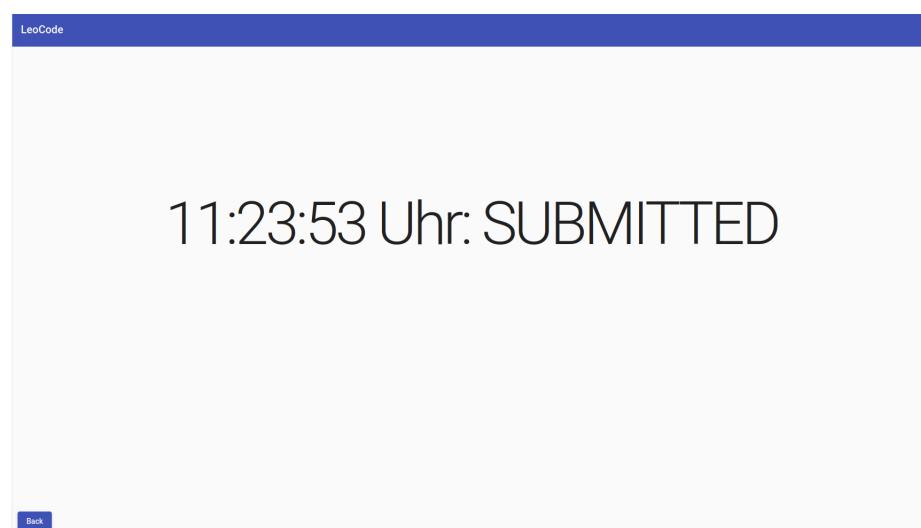


Figure 3.9: Submission Status

Wobei die Rohdaten des SSE (ohne Frontend) wie folgt zurückgeliefert werden:

```
data: 17:14:03 Uhr: SUBMITTED
```

```
data: 17:15:04 Uhr: SUCCESS
```

Figure 3.10: Server Sent Event Ouput

Chapter 4

Ausgewählte Aspekte

4.1 Messaging mit Apache Kafka

Folgende Ausarbeitung wurde mit den Quellen [3] [11] [13]

4.1.1 Event Streaming

Um zu verstehen, für welchen Einsatzzweck Apache Kafka geeignet ist, sollte man zuvor den Begriff des **Event-Streamings** besprechen. Event Streaming kann sehr gut mit dem Nervensystem des menschlichen Körpers verglichen werden. Wie der Name bereits vermuten lässt, dreht es sich hierbei um das Verarbeiten von Ereignissen. Solche Events können von einem Signal eines IOT Sensors bis hin zu Servern verschiedenste Technologien sein. Das Event Streaming beschäftigt sich damit diese Events oder Daten zu erfassen bzw. zu speichern und weiterzuverarbeiten. Die verschiedenen Prozesse laufen dabei nahezu in Echtzeit.

In der Praxis findet Event Streaming an den verschiedensten Plätzen Verwendung. Zum Beispiel in Banken zur Verarbeitung von Zahlungen oder in Krankenhäusern zur Überwachung der Gesundheitszustände von Patienten. Event Streaming spielt aber auch bei ereignisgesteuerten Architekturen und Microservices eine große Rolle, was auch auf das vorliegende Projekt zutrifft. Aber was ist die Aufgabe von Apache Kafka beim Event Streaming?

4.1.2 Was ist Apache Kafka?

Apache Kafka besteht grundsätzlich aus mehreren Servern und Clients die per TCP-Protokoll kommunizieren. Einige der Server, welche auch Broker genannt werden, sind dazu zuständig, dass die Daten persistiert werden. Andere Server wiederum sind dafür zuständig, dass die Daten der Clients zu den Brokern oder bei Bedarf von den Brokern weg gelangen. Diese Server-Architektur ist dabei sowohl skalierbar als auch ausfallsicher.

Die Apache Kafka Clients sind hingegen dafür zuständig, dass die verschiedenen Applikationen mit den Servern kommunizieren können. Apache bietet Clients in Java, Scala, Python sowie in vielen anderen Programmiersprachen und Rest Schnittstellen die von Nutzern verwendet bzw. implementiert werden können.

4.1.3 Grundbegriffe

- **Events und Messages:** Im Prinzip empfangen bzw. senden Server und Client Messages, welche aufgrund eines Events versendet werden. Diese Messages können außerdem Payloads zugewiesen werden, welche zum Beispiel Java Objekte oder sogar Json Daten sein können. Um die Definition etwas besser zu verstehen, ein kurzes Beispiel: Angenommen ein CO2-Sensor verzeichnet eine auffällig hohe Gaskonzentration, so wäre dies in diesem Beispiel ein Event. Dieses Event hat nun zu Folge, dass eine Message gesendet wird.
- **Broker:** Wie zuvor bereits erwähnt spielt der Broker eine essenzielle Rolle. Es kann einen oder mehrere Broker geben, die auf jeweils einen Server laufen. Sie sind dafür zuständig, dass alle Requests der Clients verarbeitet und die Daten repliziert werden. Die Ansammlung mehrerer Kafka Broker bezeichnet man auch als Cluster, welche eine höhere Verarbeitungsgeschwindigkeit der Anfragen ermöglichen.
- **Zookeeper:** Der sogenannte Zookeeper ist ein Service, welcher dafür zuständig ist, dass die Broker in einem Cluster gemanagt werden. Unter anderem sorgt der Zookeeper dafür, dass Konfigurationsdaten bestehen bleiben und der Status der Cluster Nodes, sowie die Kafka Topics, Partitionen, etc. im Auge behalten werden.
- **Producer bzw. Consumer:** Unter Produzent bzw. Konsument versteht man jene Komponente, welche Events sendet bzw. empfängt. Das Senden von Events wird auch publish genannt, wobei das Empfangen allgemein als subscribe bezeichnet wird.
- **Topics:** Topics kann man sich ähnlich wie eine Kategorie oder einen Ordner eines Filesystems vorstellen. Folgt man dem Filesystem Beispiel, so würden die Events den Files in jenem Ordner entsprechen. Diese Topics können jeweils mehrere Subscriber bzw. Producer besitzen, welche nur von Events die dem jeweiligen Topic zugeordnet sind benachrichtigt werden. Diese Topics sind in außerdem zusätzlich partitioniert, was bedeutet, dass ein Topic über verschiedene Kafka Broker verteilt wird.

4.1.4 Beispiel anhand zweier Quarkus Instanzen

Ziel des vorliegenden Beispieles ist es, dass zwei Quarkus Instanzen über Apache Kafka miteinander kommunizieren. Eine Instanz soll ein "Ping" senden, während der andere Server darauf mit einem "Pong" antwortet.

Um Zeit beim Setup zu sparen, wird Apache Kafka in diesem Beispiel mittels Docker

verwendet. Da zumindest ein Kafka Broker sowie Zookeeper benötigt werden, werden beide Services in einem gemeinsamen YAML-File definiert und anschließend mit der Hilfe von Docker-Compose gestartet. Eine mögliche Konfiguration sieht in etwa so aus:

```
version: '3'

services:

zookeeper:
  image: strimzi/kafka:0.19.0-kafka-2.5.0
  command: [
    "sh", "-c",
    "bin/zookeeper-server-start.sh config/zookeeper.properties"
  ]
  ports:
    - "2181:2181"
  environment:
    LOG_DIR: /tmp/logs

kafka:
  image: strimzi/kafka:0.19.0-kafka-2.5.0
  command: [
    "sh", "-c",
    "bin/kafka-server-start.sh config/server.properties --override listeners"
  ]
  depends_on:
    - zookeeper
  ports:
    - "9092:9092"
  environment:
    LOG_DIR: "/tmp/logs"
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
    KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
```

Für dieses Beispiel werden die Quarkus Extensions „Resteasy“, für einen simplen REST-Endpoint und „SmallRye Reactive Messaging“, zur Verwendung von Kafka, von der „Ping“-Instanz benötigt. Die „Pong“-Instanz hingegen benötigt lediglich die „SmallRye Reactive Messaging“-Extension. Die beiden Quarkus Projekte können entweder per Website, IDE Extension oder direkt mittels Maven auf der Konsole generiert werden.

Bash Befehle um Projekte mittels Maven zu Generieren:

- Ping-Server:

```
mvn io.quarkus:quarkus-maven-plugin:1.13.2.Final:create \
-DprojectGroupId=at.htl \
-DprojectArtifactId=kafka-messaging-demo-ping \
-DclassName="at.htl.PingResource" \
-Dpath="/ping" \
-Dextensions="resteasy,smallrye-reactive-messaging-kafka"
```

- Pong-Server:

```
mvn io.quarkus:quarkus-maven-plugin:1.13.2.Final:create \
-DprojectGroupId=at.htl \
-DprojectArtifactId=kafka-messaging-demo-pong \
-Dextensions="smallrye-reactive-messaging-kafka"
```

In beiden Projekten, wird standardmäßig ein Rest-Endpoint erstellt. Der bereits existierenden Rest-Endpoint des Ping Servers kann so umgeformt werden, dass ein "Ping" an die zweite Instanz abgeschickt wird. Um dies zu ermöglichen müssen, allerdings zunächst die **application.properties** Files verändert werden. Diese Files dienen zur Konfiguration der Quarkus Server bzw. des Kafka Clients.

Zuvor empfiehlt es sich einen Plan zur Konfiguration des Kafka-Servers zu erstellen: Wie viele Channels bzw. Topics werden benötigt? Welche Payloads werden versandt? In vorliegenden Beispiel sind jene Punkte allerdings relativ schnell geklärt. Zuerst zur einfacheren Frage, welche Payloads versandt werden. Da hierbei lediglich "Ping" bzw. "Pong" versandt werden soll, ist die Antwort auf diese Frage String-Payloads bzw. Messages.

Die Frage, wie viele Channels bzw. Topics benötigt werden, ist ebenfalls schnell geklärt. Grundsätzlich könnte man hierbei ein gemeinsames Topic erstellen, allerdings müssten dann die Nachrichten gefiltert werden, da sonst ein Endlosschleife entstehen würde. Um sich die Filterfunktion zu sparen würde es sich anbieten schlicht ein zweites Topic zu erstellen, ein Topic für Pings und ein Topic für Pongs.

Die beiden Properties-Files unterscheiden sich in Wirklichkeit nicht wirklich, sie sind schlicht genau das Gegenteil des jeweils anderen. Eine mögliche Konfiguration wäre beispielsweise:

- Ping-Server:

```
kafka.bootstrap.servers=localhost:9092

mp.messaging.outgoing.ping.connector=smallrye-kafka
mp.messaging.outgoing.ping.topic=pings
mp.messaging.outgoing.ping.value.serializer=org.apache.kafka.common.seria

mp.messaging.incoming.pong.connector=smallrye-kafka
mp.messaging.incoming.pong.topic=pongs
mp.messaging.incoming.pong.value.deserializer=org.apache.kafka.common.ser
mp.messaging.incoming.pong.health-readiness-enabled=false
```

- Pong-Server:

```
quarkus.http.port=8081
kafka.bootstrap.servers=localhost:9092

mp.messaging.outgoing.pong.connector=smallrye-kafka
mp.messaging.outgoing.pong.topic=pongs
mp.messaging.outgoing.pong.value.serializer=org.apache.kafka.common.seria

mp.messaging.incoming.ping.connector=smallrye-kafka
mp.messaging.incoming.ping.topic=pings
mp.messaging.incoming.ping.value.deserializer=org.apache.kafka.common.ser
mp.messaging.incoming.ping.health-readiness-enabled=false
```

In den vorliegenden Beispiel für ein application.properties File wird zu Beginn definiert, unter welcher Adresse der Kafka Broker erreichbar ist (im Pong Beispiel wird außerdem der Standart HTTP-Port des Quarkus Server verändert, da die beiden Instanzen sich sonst gegenseitig interferenzieren). Anschließend werden die Channels Konfiguriert.

mp.messaging.[outgoing—incoming].channel-name.property=value

Der Channel-Name kann grundsätzlich frei gewählt werden, muss allerdings mit den Werten der **@Incoming** bzw. **@Outgoing** Annotationen übereinstimmen. Wie zuvor erwähnt gibt es zwei verschieden Topics für "Ping" bzw. "Pong". Dieser ist ähnlich zum Channel-Name ebenfalls frei wählbar, muss aber mit der Konfiguration des jeweils anderen Servers übereinstimmen. Wäre dies nicht der Fall, würden die Messages unter einem falschen Topic gesendet und somit nicht empfangen werden. Zusätzlich muss noch ein Serialisierer bzw. Deserialisierer gesetzt werden, damit die Messages korrekt umgewandelt werden.

Man könnte hier beispielsweise auch eigene De-/Serialisierer angeben um zum Beispiel Json-Objekte per Kafka zu versenden.

Nach der korrekten Konfiguration können Messages wie folgt vom Ping-Server versendet werden:

```
15 package at.html;
16
17 import org.eclipse.microprofile.reactive.messaging.Channel;
18 import org.eclipse.microprofile.reactive.messaging.Emitter;
19 import org.jboss.logging.Logger;
20
21 import javax.inject.Inject;
22 import javax.ws.rs.GET;
23 import javax.ws.rs.Path;
24
25 @Path("/ping")
26 public class PingResource {
27
28     @Inject
29     Logger log;
30
31     @Inject
32     @Channel("ping")
33     Emitter<String> pingEmitter;
34
35     @GET
36     public void sendPing() {
37         log.info("sent Ping");
38         pingEmitter.send("ping");
39     }
40 }
```

Die **@Path** bzw. **@GET** Annotationen sind von Resteasy und erstellen einen **GET http://localhost:8080/ping** Endpoint. Bei einem Request wird schlicht die Methode **sendPing()** ausgeführt. Die **@Inject** Annotationen erstellen mittels Dependency Injection Objekte, während die **@Channel** Annotation sowie das **Emitter** Objekt zum Messaging benötigt werden.

Die **@Channel** Annotation gibt an, in welchen Channel die Message gesendet wird, wobei das **Emitter** Objekt dafür verwendet wird, dass Messages von imperativen Code gesendet werden können

Als Gegenstück dazu dient der PingListener des Pong Servers, welcher wie folgt aussieht:

```
41 package at.htl;
42
43 import org.eclipse.microprofile.reactive.messaging.Channel;
44 import org.eclipse.microprofile.reactive.messaging.Emitter;
45 import org.eclipse.microprofile.reactive.messaging.Incoming;
46 import org.jboss.logging.Logger;
47
48 import javax.inject.Inject;
49
50 public class PingListener {
51
52     @Inject
53     Logger log;
54
55     @Inject
56     @Channel("pong")
57     Emitter<String> emitter;
58
59     @Incoming("ping")
60     public void sendPong(String msg) {
61         log.info("received " +msg);
62         log.info("send pong");
63         emitter.send("pong");
64     }
65 }
```

Der einzige Unterschied zur PingResource ist, dass die sendPong() Methode nicht durch einen HTTP-Request ausgeführt wird, sondern durch eine Message im ping Channel. Nachdem in der vorliegenden Methode die empfangene Nachricht auf der Konsole ausgegeben wurde, wird ein "pong" gesendet.

Um die Message zu Empfangen, wird abschließend noch der PongListener am Ping-Server benötigt:

```

66 package at.htl;
67
68 import org.eclipse.microprofile.reactive.messaging.Incoming;
69 import org.jboss.logging.Logger;
70
71 import javax.inject.Inject;
72
73 public class PongListener {
74
75     @Inject
76     Logger log;
77
78     @Incoming("pong")
79     public void receivePong(String msg) {
80         log.info("received " + msg);
81     }
82 }
```

Diese Klasse höhrt lediglich auf Messages im pong Channel und gibt diese anschließend auf der Konsole aus.

Sind nun die Docker-Konfiguration sowie die beiden Quarkus Server gestartet, kann der **GET** Request an **http://localhost:8080/ping** gesendet werden. Daraufhin sendet der Ping-Server ein "ping" an die andere Quarkus Instanz, welche wiederrum ein "pong" erwidert.

```

christian@Blade: ~/school/da/kafka-messaging-demo/kafka-messaging-demo...
[2021-04-16 02:46:58,671] INFO [org.apa.kaf.cli.con.int.ConsumerCoordinator] (vert.x-kafka-consumer-thread-0) [Consumer clientId=kafka-consumer-pong, groupId=kafka-messaging-demo-ping] Setting offset for partition pongs-0 to the committed offset FetchPosition{offset=6, offsetEpoch=Optional.empty, currentLeader=LeaderAndEpoch{leader=Optional[localhost:9092 (id: 0 rack: null)], epoch=0}}
[2021-04-16 02:47:05,186] INFO [at.htl.PingResource] (executor-thread-1) sent Ping
[2021-04-16 02:47:05,288] INFO [at.htl.PongListener] (vert.x-eventloop-thread-0) received pong

christian@Blade: ~/school/da/kafka-messaging-demo/kafka-messaging-demo...
[2021-04-16 02:47:00,465] INFO [org.apa.kaf.cli.con.int.ConsumerCoordinator] (vert.x-kafka-consumer-thread-0) [Consumer clientId=kafka-consumer-ping, groupId=kafka-messaging-demo-pong] Setting offset for partition pings-0 to the committed offset FetchPosition{offset=61071, offsetEpoch=Optional.empty, currentLeader=LeaderAndEpoch{leader=Optional[localhost:9092 (id: 0 rack: null)], epoch=0}}
[2021-04-16 02:47:05,255] INFO [at.htl.PingListener] (vert.x-eventloop-thread-0) received ping
[2021-04-16 02:47:05,256] INFO [at.htl.PingListener] (vert.x-eventloop-thread-0) send pong
```

Figure 4.1: Server Konsolen Output (Ping-Server oben, Pong-Server unten)

Chapter 5

Resümee

Dadurch, dass mir Corona-bedingt die Kooperation mit einer Firma nicht zustande kam, hab ich die Diplomarbeit allein realisiert. Mit vollem Elan setzte ich mich bereits im Laufe des Sommersemesters der vierten Klasse an die Umsetzung der Diplomarbeit. Zu Beginn gelang es mir auch relativ schnell erste Prototypen zu erstellen, welche allerdings nach und nach immer komplexer wurden.

Im Laufe der Diplomarbeit habe ich nicht zuletzt wegen der alleinigen Umsetzung mit sehr viel verschiedenen Technologien gearbeitet. Dementsprechend habe ich extrem viel bei der Realisierung sowie bei der Systemplanung lernen können. Abschließend lässt sich sagen, dass sich die Monate der harten Arbeit gelohnt haben, da ein recht ansehnliches System entwickelt wurde.

Chapter 6

Anhang

6.1 Besprechungsprotokolle

6.1.1 15.04.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	15.04.2020 - 4.EH
Dauer	30min

Besprochene Themen:

- Thema für eine Diplomarbeit
- Rahmenbedingung/Aufgabenstellung

Vereinbarungen und Entscheidungen:

- Realisierung mit Online Editor? Welchen?
- Inspiration von verschiedenen Plattformen wie z.B.: KataCoda
- Antrag auf der Diplomarbeitsdatenbank stellen

6.1.2 29.04.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	29.04.2020 - 1.EH
Dauer	60min

Besprochene Themen:

- Überarbeitung des Diplomarbeitantrages
- Termine der Meilensteile fixiert
- Nächsten Schritte

Vereinbarungen und Entscheidungen:

- Schüler besitzen ein "Portfolio" (erledigte Aufgaben inklusive Ergebnisse)
- Beispiele erhalten Tags, zur Sortierung/Strukturierung, Schüler können Beispiele nach Themen sortieren bzw. auswählen
- Nächste Schritte:
 - Java Projekt auf Consolen Input bzw. Output überprüfen
 - Code nach "Verbotenen" Words überprüfen
 - Beispiel dafür zum Beispiel: Eine Schleife welche immer i ausgibt, Userabfrage wie oft eine Schleife i ausgeben soll
- Überarbeitung des Antrages:
 - Ausgangslage
 - Ergebnis
 - Meilensteine

6.1.3 18.05.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	18.05.2020 - 11:30
Dauer	30min

Besprochene Themen:

- Systemarchitektur
- Nächsten Schritte

Vereinbarungen und Entscheidungen:

- Recherche bzgl. Entwicklung eines Plugins für Theia
- evtl. Theia Plugin zur Abgabe eines Beispieles verwenden
- evlt. Angabe, Abgabe sowie Testfunktion direkt in einer IDE durch Plugin
- Systemarchitektur:

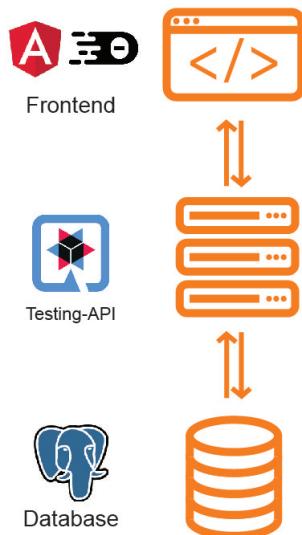


Figure 6.1: Systemarchitektur LeoCode V1

6.1.4 07.06.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	07.06.2020 - 10:45
Dauer	60min

Besprochene Themen:

- weitere Vorgehensweise

Vereinbarungen und Entscheidungen:

- Server vorerst simulieren (lokal)
- Möglichkeit zum Ausführen von Unittests:
 - Maven
 - Jenkins
 - Java (Robot API)
- Was muss geprüft werden?
 - Ausführen von Unittests
 - Überprüfung auf Keywords zB.: Verwendung von while
 - möglicherweise Verwendung von ANTLR für weiterführende Überprüfungen

6.1.5 21.06.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	21.06.2020 - 10:30
Dauer	45min

Besprochene Themen:

- Änderungen/Verbesserungen der Testing-API
- nächste Schritte

Vereinbarungen und Entscheidungen:

- Feedback Testing-API:
 - Logging inkonsistent
 - eigenes Projekt für Tests, separat zum Quarkus Backend
 - Um Test vom Code aus zu Starten Junit Platform launcher
 - System Properties in application.properties

- Files um Testfälle zu simulieren
- README Updaten
- Runtime.exec() um Shell Befehle mit Java auszuführen
- Eigener Discord Server

6.1.6 29.06.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Schule
Datum - Uhrzeit	29.06.2020 - 10:00
Dauer	30min

Besprochene Themen:

- Systemarchitektur
- nächste Schritte

Vereinbarungen und Entscheidungen:

- Abgabefunktion durch VSCode Plugin aber auch durch Angular Client bzw. anderen IDE Plugins
- Angabe ähnlich zu einer Markdown Preview
- Github Pages überarbeiten
- Präsentation mithilfe von RevealJs
- Verwendung eines Swaggers

6.1.7 20.07.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	20.07.2020 - 8:50
Dauer	70min

Besprochene Themen:

- Testing API Feedback
- nächste Schritte

Vereinbarungen und Entscheidungen:

- Diplomarbeit in Markup Sprache schreiben und anschließend umwandeln?
- Feedback Testing API:
 - Testing API sollte OS unabhängig sein

- Windows kompatibel?
- Path in Constructor setzen (@ConfigProperty)
- Nächsten Schritte:
 - ERD
 - ”Ein User System”
 - Testing-API mit Postgres (Docker)
- Auslagerung des User-Management auf Schulserver
- arbeiten mittels Feature-Branches

6.1.8 29.07.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	29.07.2020 - 10:30
Dauer	45min

Besprochene Themen:

- Status Quo
- Diplomarbeit in Markdown schreiben und in latex umwandeln? (pandoc)
- Feedback
- nächsten Schritte

Vereinbarungen und Entscheidungen:

- Backkend Files an Testing-API als Json oder Input (@FormsParam)
- Testing-API trennen von backend, da in Zukunft weitere Sprachen so leichter erweitert werden können
- Angular Client als Frontend
- mögliche Implementierung von Jenkins

6.1.9 03.08.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	03.08.2020 - 09:50
Dauer	15min

Besprochene Themen:

- Status Quo
- nächsten Schritte

Vereinbarungen und Entscheidungen:

- Rest Client Request an Testing-API mit gesammelten Zugriff
- Performance jetzt noch nicht relevant

6.1.10 06.08.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	06.08.2020 - 10:15
Dauer	60min

Besprochene Themen:

- Native Build Error bei Backend (Hibernate)
- Dockern einer Quarkus Applikation, Docker-Compose

Vereinbarungen und Entscheidungen:

- Kann nicht funktionieren, Docker Container greift auf seinen eigenen Localhost zugreift
- Lösung mittels Docker-Compose

6.1.11 07.08.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	07.08.2020 - 9:30
Dauer	30min

Besprochene Themen:

- Jenkins
- Deployment in Oracle Cloud

Vereinbarungen und Entscheidungen:

- Jenkins
- Deployment in Oracle Cloud

6.1.12 29.08.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	29.08.2020 - 10:00
Dauer	1h 30min

Besprochene Themen:

- Systemarchitektur
- nächste Schritte

Vereinbarungen und Entscheidungen:

- Systemarchitektur mit Plant UML
- Umstieg auf Jenkins
- Skizze des Angular Clients
- Erstellen einer Jenkins Pipeline (Webhook)

6.1.13 01.09.2020

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	01.09.2020 - 9:00
Dauer	2h

Besprochene Themen:

- Quarkus Native Build Zeit (Jib Plugin)
- Jenkins Setup (Plugin Error)

Vereinbarungen und Entscheidungen:

- Erstellen einer Jenkins Pipeline

6.1.14 01.12.2020

Teilnehmer	T. Stütz, C. Aberger, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	01.12.2020 - 9:00
Dauer	1h 45min

Besprochene Themen:

- Status Quo
- Nächsten Schritte

Vereinbarungen und Entscheidungen:

- Systemarchitektur: Docker bis jetzt irrelevant
- Exception Handling
- User benötigt Feedback
- Kommunikation zwischen Testing API & Backend mittels Kafka
- Zip File über Filesystem "teilen"
- Batch Processing

6.1.15 03.01.2021

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	03.01.2021 - 10:00
Dauer	1h 30min

Besprochene Themen:

- Status Quo
- Inhaltsverzeichnis Diplomarbeit
- Nächsten Schritte

Vereinbarungen und Entscheidungen:

- evtl. Submission als Submodule (nicht wichtig)
- beim SSE Timestamp hinzufügen
- Systemarchitektur updaten
- Wie wird Diplomarbeit geschrieben, workflow?

6.1.16 13.01.2021

Teilnehmer	T.Stütz, H. Bahar, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	19.01.2021 - 20:00
Dauer	1h

Besprochene Themen:

- Status Quo
- nächsten Schritte

Vereinbarungen und Entscheidungen:

- detaillierter Ablauf einer gedockerten Version
- rudimentäres Frontend mittels Angular

6.1.17 08.02.2021

Teilnehmer	T.Stütz, C. Donnabauer
Ort	Discord
Datum - Uhrzeit	08.02.2021 - 10:54
Dauer	2h

Besprochene Themen:

- schriftlicher Teil
- Marktanalyse Kriterien festgelegt
- Nächsten Schritte

Vereinbarungen und Entscheidungen:

- beim Schriftlichen mit dem schwersten beginnen
- Marktanalyse Varianten: scrimba, stackblitz, codingame, katacoda, freecodecamp, Edu tools und Stepik

Chapter 7

Quellenverzeichnis

- [1] Gerald Aistleitner. "Simplified Hibernate ORM with Panache". en. In: (), p. 20.
- [2] *Angular Material Tutorial - Tutorialspoint*. URL: https://www.tutorialspoint.com/angular_material/index.htm (visited on 04/17/2021).
- [3] *Apache Kafka*. en. URL: <https://kafka.apache.org/documentation/> (visited on 03/30/2021).
- [4] baeldung. *The DAO Pattern in Java — Baeldung*. en-US. June 2018. URL: <https://www.baeldung.com/java-dao-pattern> (visited on 04/17/2021).
- [5] Anshul Bansal. *DAO vs Repository Patterns — Baeldung*. en-US. Sept. 2020. URL: <https://www.baeldung.com/java-dao-vs-repository> (visited on 04/17/2021).
- [6] *EduTools - Plugins — JetBrains*. URL: <https://plugins.jetbrains.com/plugin/10081-edutools> (visited on 04/21/2021).
- [7] Sebastian Eschweiler. *Docker — Beginner's Guide — Part 1: Images & Containers*. en. Feb. 2019. URL: <https://medium.com/codingthesmartway-com-blog/docker-beginners-guide-part-1-images-containers-6f3507fffc98> (visited on 04/19/2021).
- [8] heise online heise. *Nichtrassistische Sprache: Abschied von Blacklist und Whitelist*. de. URL: <https://www.heise.de/news/Nichtrassistische-Sprache-Abschied-von-Blacklist-und-Whitelist-4784291.html> (visited on 04/20/2021).
- [9] *jenkinsci/jenkinsfile-runner*. original-date: 2018-02-24T23:56:01Z. Apr. 2021. URL: <https://github.com/jenkinsci/jenkinsfile-runner> (visited on 04/18/2021).
- [10] *Katacoda - Interactive Learning Platform for Software Engineers*. URL: / (visited on 04/21/2021).
- [11] *Part 1: Apache Kafka for beginners - What is Apache Kafka? - CloudKarafka, Apache Kafka Message streaming as a Service*. en. URL: <https://www.cloudkarafka.com/blog/part1-kafka-for-beginners-what-is-apache-kafka.html> (visited on 04/08/2021).

- [12] *Play with Programming.* en. URL: <https://www.codingame.com/home> (visited on 04/20/2021).
- [13] *Quarkus - Using Apache Kafka with Reactive Messaging.* URL: <https://quarkus.io/guides/kafka> (visited on 04/14/2021).
- [14] *Repository Pattern Graph.* URL: https://norberteder.com/wp-content/uploads/images/image_66.png (visited on 04/17/2021).

Abbildungsverzeichnis

1.1	Struktur LeoLearn	7
2.1	Kommunikation der Komponenten LeoCode V1	9
2.2	Systemarchitektur LeoCode V1	10
2.3	Systemarchitektur LeoCode V2	11
2.4	Online Editor Theia	12
2.5	Systemarchitektur LeoCode V3	13
2.6	Systemarchitektur LeoCode V4	14
2.7	Erstellen einer Aufgabe	15
2.8	Auflistung der Aufgabenstellungen	16
2.9	Detailansicht der Aufgabenstellung	16
2.10	Erstellen einer Abgabe	16
2.11	Statusansicht einer Abgabe	17
2.12	Abgabe Verlauf eines Nutzers	17
2.13	Funktionsweise Repository Pattern, https://norberteder.com/wp-content/uploads/images/image_66.png	18
2.14	von Backend erstellte Zip-File Projektstruktur	21
2.15	Beispiel für Maven Projektstruktur	22
2.16	Docker vs. Virtuelle Maschinen, https://medium.com/codingthesmartway-com-blog/docker-beginners-guide-part-1-images-containers-6f3507fffc98	24
2.17	Docker in Docker Systemarchitektur	25
2.18	Docker Systemarchitektur	25
3.1	Teacher LeoCode: Erstellen eines Beispieles	26
3.2	Student LeoCode: Abgabe eines Beispieles	27
3.3	von Backend erstellte Zip-File Projektstruktur	27
3.4	Create Example	30
3.5	Get Example By ID	31
3.6	Auflistung der Aufgabenstellungen	32
3.7	Create Submission	33
3.8	Portfolio	33
3.9	Submission Status	34
3.10	Server Sent Event Ouput	35

