

Inhaltsverzeichnis

1	Motivation	2
2	Einleitung	2
2.1	Aufgabenbeschreibung	2
2.2	Herangehensweise	2
3	Grundlagen	2
3.1	Code Testing	2
3.2	Das Unit Testsystem	3
3.3	Jenkins	3
3.4	Docker	3
3.5	Git-Hosting mit Gogs	5
4	Test-C Plattform	5
4.1	Einführung	5
4.2	Architektur	5
4.3	Alternativen	21
4.4	Ablauf des Testverfahrens	21
4.5	Installationsanleitung	21
4.6	Verwendung	21
5	Zusammenfassung	21
6	Quellcodeverzeichnis	21
7	Quellenverzeichnis	21

1 Motivation

2 Einleitung

2.1 Aufgabenbeschreibung

Am Anfang stand die Idee, dass ich mich immer tiefer mit Automatisierung beschäftige. Deshalb fiel die Wahl auf ein Projekt in diese Richtung. Im Rahmen dieser wissenschaftliche Vertiefung Modul wird ein auf Docker basierendes Testsystem entwickelt. Dieses Test-Framework soll die Studenten helfen, ihre Übungsaufgaben automatisiert zu testen.

Der Student lädt seinen Quellcode über eine Web-Benutzeroberfläche hoch, dieser Service steht den Studenten nach der Registrierung auf der Testplattform zur Verfügung. Für mich ist die Studenten Registrierung in der Tat eine Erstellung eines eigenen Remote-Testraums, der dadurch der Testprozess verwaltet wird, und das, was ich in dieser Ausarbeitung im Detail erklären werde. Nach Abschluss des Testprozesses erhält der Student einen Bericht mit dem Ergebnis seines Tests. Darüber hinaus sollte dieses Test-Framework der Lehrkraft die Möglichkeit geben, die individuellen Testaufgaben der Studenten zu kontrollieren.

2.2 Herangehensweise

In diesem Abschnitt werde ich die Techniken und Technologien vorstellen, die in diesem Projekt verwendet werden. Daher wurde das Konzept des Code-Testens erläutert. Ebenso wird das in diesem Projekt verwendete Unit-Test-System besprochen. Anschließend werden die Rolle des Jenkins-Tools im Projekt und die Bedeutung dieses Tools für die Automatisierung des Testprozesses erläutert, dann folgt eine ausführliche Erläuterung der Verwendung von Docker im Projekt. Abschließend eine Erläuterung von Git-Hosting und Jgit, die einen der wichtigsten Teile des Projekts darstellen.

Im Hauptabschnitt wird dann die Erstellung der Testplattform erläutert. Es wird die Architektur beschrieben, und die Entscheidungen, die getroffen wurde, indem die Techniken ausgewählt wurde, mit denen das Projekt die Anforderungen erfüllt. Außerdem werden der Ablauf des Testverfahrens und die internen Prozesse des Bestandteils der Testplattform vorgestellt. Dann werden die Installationsanweisungen und Verzerrungen der Testplattform klarstellen.

3 Grundlagen

3.1 Code Testing

Beim Softwaretesten wird ein Programm oder eine Anwendung ausgeführt, um die Fehler zu ermitteln und um zu überprüfen, ob die erforderlichen Bedingungen erfüllt sind.

Softwaretestverfahren können in 2 Gruppen unterteilt werden: Statisches / Statisches Testen.

3.1.1 Manual testing(Statisches)

Eine Softwaretesttechnik, bei der die Software getestet wird, ohne den Code auszuführen. Es besteht aus zwei Teilen:

- Review - Wird normalerweise verwendet, um Fehler oder Unklarheiten in Dokumenten wie Anforderungen, Design, Testfällen usw. zu finden und zu beseitigen.
- Statische Analyse - Der von Entwicklern geschriebene Code wird (normalerweise mithilfe von Werkzeugen) auf strukturelle Fehler analysiert.

3.1.2 Automated Testing(Dynamisches)

Beim dynamischen Testen wird die Software auf die Eingabewerte geprüft und die Ausgabewerte analysiert. Es gibt verschiedene Stufen dynamischer Testtechniken:

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

In diesem Fall verwenden wir eines Dynamisches Testen, um die Programmieraufgaben der Studenten zu bewerten.

3.2 Das Unit Testsystem

3.3 Jenkins

3.3.1 Einführung

3.3.2 Jenkins buildflow

3.3.3 Jenkins pipeline

3.3.4 Jenkins multijob

3.3.5 Jenkinsfile

3.4 Docker

3.4.1 Einführung

3.4.2 Docker Architektur

Nach meiner Erfahrung besteht eine der einfachsten Möglichkeiten, Docker zu verstehen und anzuwenden, darin, sich einen Überblick darüber zu verschaffen, wie die Docker-Plattform im

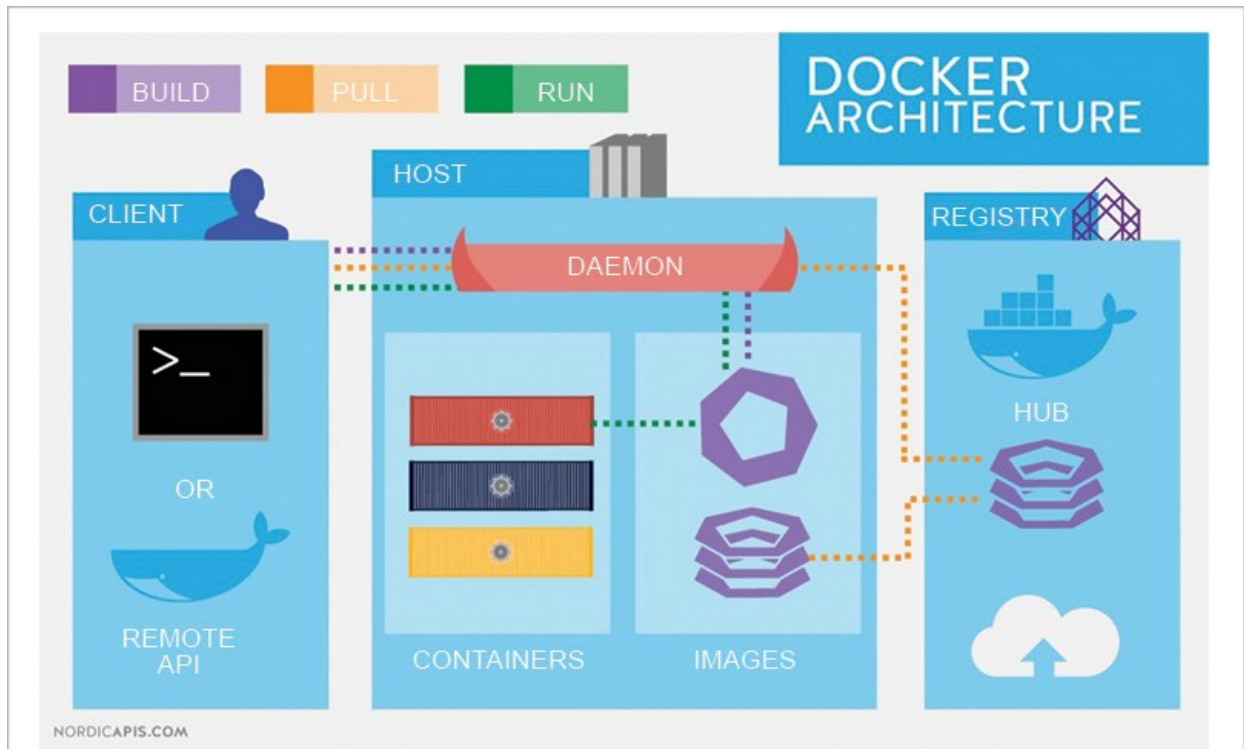


Abbildung 1: Docker Architektur

Hintergrund aufgebaut ist.

In Abbildung 1 sehen Sie die Hauptkomponenten einer Docker-Installation:

- Im Zentrum steht der Docker-Daemon, der für das Erstellen, Ausführen und Überwachen von Containern sowie für das Erstellen und Speichern von Images zuständig ist. Der Docker-Dämon wird vom Host-Betriebssystem verwaltet.
- Der Docker-Client befindet sich auf der linken Seite und wird für die Kommunikation mit dem Docker-Daemon über HTTP verwendet. Ein Docker-Client kann mit mehr als einem Daemon kommunizieren, da sich der Docker-Client auf demselben Host wie der Daemon oder über eine Verbindung auf einem Remote-Host befinden kann. Der Docker-Client bietet eine Befehlszeilenschnittstelle (Command Line Interface, CLI), mit der Sie Anwendungsbefehle für einen Docker-Dämon erstellen, ausführen und stoppen können.
- Docker-Registries speichern und verteilen Images, die Standardregistrierung heißt Docker Hub und hostet Tausende von öffentlichen Images. Viele Unternehmen führen eigene Register, in denen kommerzielle oder vertrauliche Images gespeichert werden können. Der Docker-Daemon lädt als Reaktion auf Docker-Pull-Requests Images von Registries herunter. Es werden auch automatisch Images heruntergeladen, die in Docker-Run-Requests und in der FROM-Anweisung von Dockerfiles angegeben sind, wenn sie nicht lokal verfügbar sind.

Wenn in diesem Abschnitt etwas nicht verstanden wird, wie z. B. Bilder, Dockerfile ..., machen Sie sich keine Sorgen, da in den folgenden Abschnitten alles im Detail erläutert wird.

3.4.3 Docker Images

3.4.4 Docker Container

3.4.5 Docker Volumes

3.4.6 Data Container

3.4.7 Docker Compose

3.4.8 DockerFile

3.5 Git-Hosting mit Gogs

3.5.1 Einführung

3.5.2 Datenbankeinstellung

3.5.3 JGit

3.5.4 Webhook

4 Test-C Plattform

4.1 Einführung

In diesem Abschnitt wird die Test C-Plattform detailliert beschrieben, wobei zunächst die Zusammensetzung der gesamten Programmarchitektur erörtert wird, anhand derer die Programmkomponenten separat diskutiert werden können. Nach der Erklärung der Komponenten des Programms auf separate Weise, wird im Abschnitt (Interne Prozesse) genau erläutert, wie sie zusammenarbeiten.

Wenn der Leser das oben Genannte versteht, kann er verstehen, was ich im Abschnitt (Ablauf des Testverfahrens) erläutert wird.

4.2 Architektur

Wie in der Abbildung 2 gezeigt, ist die Test-C-Plattformarchitektur in drei Hauptteile unterteilt: Client-Seite, Serverseite und Remote-TestUmgebung.

Die Wahl der Technologien beruhte auf ihrer Modernität und ihrer Verbreitung auf dem Arbeitsmarkt. In Teil eins des Projekts (Client-Seite/Frontend) wurde die folgenden Techniken verwendet : Angular 7, NGX-Bootstrap, Html, Css, im zweiten Teil des Projekts (Serverseite / BackEnd) wurden Maven und Spring Boot, Spring Security und Ldap ... verwendet. Der dritte Teil des Projekts, in dem der Testprozess durch Jenkins abgeschlossen ist.

Im folgenden Abschnitt werde ich jeden Teil des Architektur separat erklären.

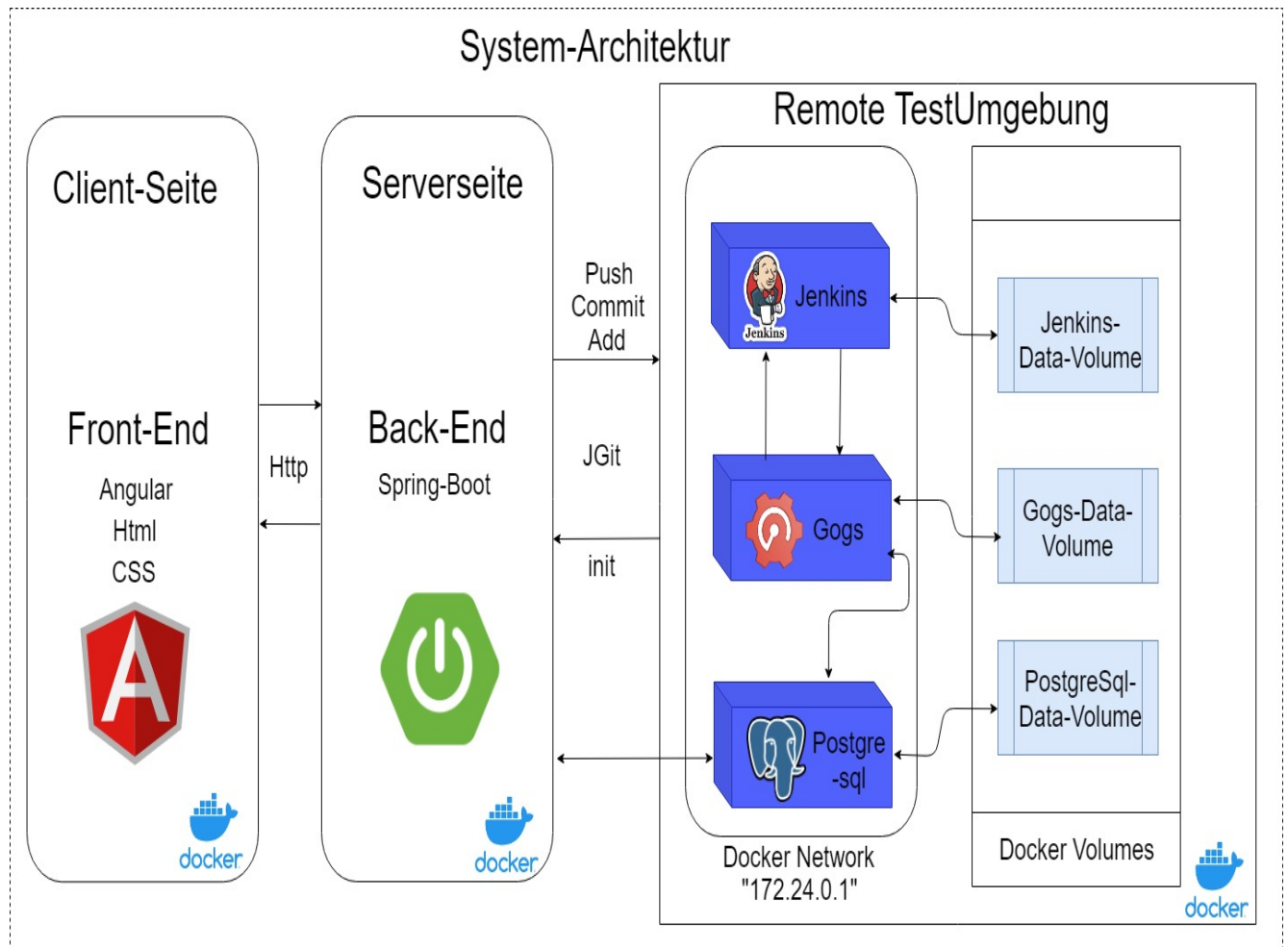


Abbildung 2: Architektur Test-C-Plattform

4.2.1 Client-Seite (Front-End)

In diesem Projekt ist die Benutzeroberfläche in zwei Teile unterteilt: das Administrationsportal und das Studentenportal. Beide Portale wurden mit Angular 7, Html, Css und NGX-Bootstrap erstellt. Über das Administrations Portal kann der Admin Benutzer ein neues Aufgabenblatt erstellen, dies bedeutet, dass ein virtueller Bereich für den Aufgabenblatttest erstellt wird, in dem die Studenten ihre Lösungen für Aufgaben testen können. Der Administrator kann alle Testdateien für jedes Aufgabenblatt in den entsprechenden Testbereich hochladen. Der Administrator kann auch die Testbereiche anzeigen. Jeder Testbereich ist mit dem Namen des zu testenden Aufgabenblatts gekennzeichnet. Für jeden Testbereich werden die spezifischen Testdateien angezeigt und der Administrator kann davon Dateien löschen oder andere hinzufügen. Über das Studentenportal kann der Student die zu testende Lösungsdatei hochladen und dann an den Testbereich senden, der den Test automatisch durchführt und dem Studenten das Testergebnis anzeigt.

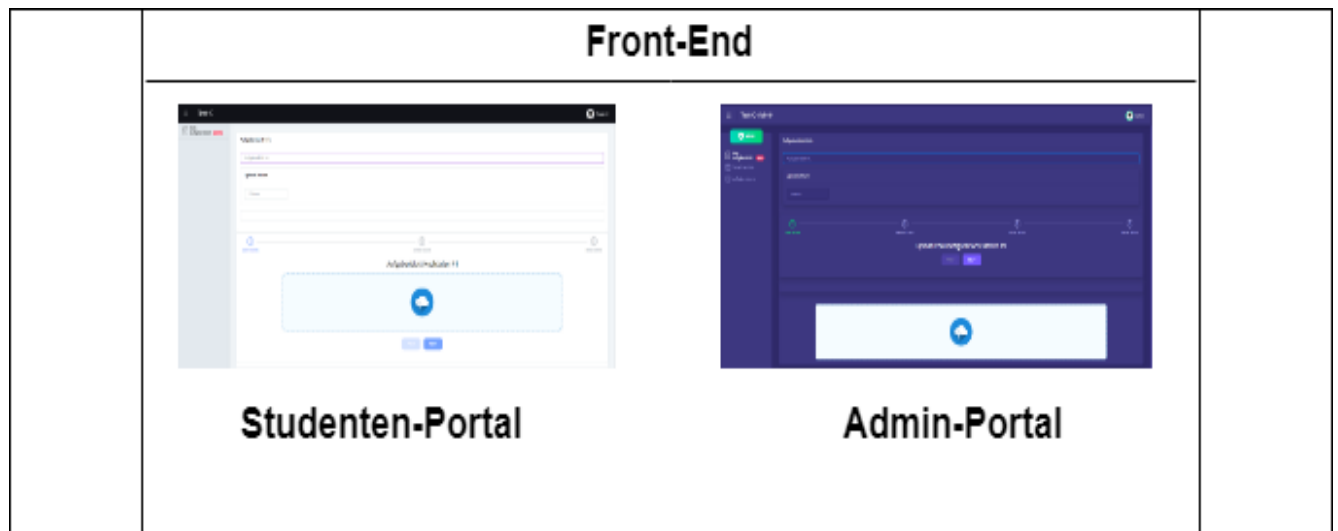


Abbildung 3: Front-End

4.2.2 Remote TestUmgebung

Die Erklärung dieses Teils des Projekts vor der Erklärung des Mittelteils 'Serverseite' ist eine absichtliche Sache, damit der Leser einfach verstehen kann, was im Back-End passieren wird.

Die Remote-Testumgebung besteht aus drei miteinander verbundenen Docker-Containern (Jenkins-Container / Gogs-Container / PostgreSQL-Container). Jeder Container verfügt über ein eigenes Datenvolumen zum Speichern der Daten. Dadurch wird vermieden, dass Informationen gelöscht werden, wenn der Container ausgeschaltet oder gelöscht wird.

Um einen Docker-Container in diesem Teil des Projekts zu erstellen, werden zwei verschiedene Methoden verwendet, entweder über eine Dockerfile Datei oder über eine Docker-Compose Datei. Für den Jenkins-Container wurde eine Dockerfile Datei verwendet. Der Jenkins-Container besteht hauptsächlich aus einem Jenkins, den Compilern: gcc und g++, die für den Build verantwortlich sind, und doxygen, das für den Testbericht verantwortlich ist.

Die Frage, die sich jetzt stellt, ist: Was ist eine Dockerfile Datei?

Ein Dockerfile ist einfach eine Textdatei mit einer Reihe von Anweisungen (instructions), die genutzt werden können, um ein Docker-Image zu erzeugen.

Listing 1: Dockerfile - jenkins

```
FROM jenkins/jenkins

USER root

RUN apt-get -y update && apt-get -y upgrade

# install gcc g++ gfortran
RUN apt-get -y install build-essential
```

```
# install static analysis
RUN apt-get -y install doxygen graphviz
```

Die FROM-Anweisung legt fest, welches Basisimage zu verwenden ist, wie in diesem Fall Jenkins. RUN-Anweisungen geben einen Shell-Befehl an, der im Image ausgeführt werden soll. In diesem Fall werden g++, gcc und doxygen ausgeführt.

Für den Gogs-Container und den PostgreSQL-Container wurde eine Docker-Compose Datei verwendet. Docker Compose wird zum Definieren und Ausführen von Docker-Anwendungen mit mehreren Containern verwendet. Mit Compose wird eine YAML-Datei (docker-compose.yml) verwendet, um die Dienste (services) von der Anwendung zu konfigurieren, wobei jeder Dienst (service) einen Container mit der zum Erstellen erforderlichen Konfiguration darstellt.

Listing 2: docker-compose.yml - Gogs/PostgreSQL

```
version: '2'
services:
  app:
    image: gogs/gogs:latest
    volumes:
      - /tmp/gogs-data:/data
    ports:
      - "3000:3000"
    links:
      - postgres:postgres
  postgres:
    image: postgres:alpine
    volumes:
      - /tmp/gogs-postgres:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=password123
      - POSTGRES_DB=gogs
```

Gogs ist ein einfacher selbst-gehosteter Git Service, der einfach einzurichten und zu betreiben ist und auf fast allem ausgeführt werden kann. Es ist zu Hundertprozent Open Source unter der MIT OSS-Lizenz. Gogs bietet die Anzeige und Bearbeitung von Repository-Dateien, die Verfolgung von Projektproblemen und ein eingebautes Wiki für die Projektdokumentation. Gogs benötigt eine Datenbank, um den Quellcode zu speichern, daher wurde hier postgresql ausgewählt.

Nun stellt sich die Frage, wie die Container miteinander kommunizieren ?

Damit Docker-Container über den Host-Computer miteinander und mit der Außenwelt kommunizieren können, wird in diesem Fall eine Netzwerkschicht (Docker Networking) eingesetzt. Docker unterstützt verschiedene Arten von Netzwerken, die jeweils für bestimmte Anwendungsfälle geeignet sind: Bridge mode, Host mode, Container mode, und No networking.

Listing 3: Docker Network

[


```
{
  "Name": "gogs-compose_default",
  "Id": "e19d76ee7d9f4132329d44e618bea3c5d93466527dd991d35b01fe4262ba6d73",
  "Created": "2019-09-09T17:14:13.0772793Z",
  "Scope": "local",
  "Driver": "bridge",
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "default",
    "Options": null,
    "Config": [
      {
        "Subnet": "172.23.0.0/16",
        "Gateway": "172.23.0.1"
      }
    ]
  },
  "Internal": false,
  "Attachable": false,
  "Ingress": false,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {
    "0e69b0fffaf865665b4c2788f0d43a450301dff06abd50fedc4d24c0c97f0b4": {
      "Name": "jenkins/jenkins",
      "EndpointID": "0fc2f145af6d60b8da71e592299d9a0e2d7f578a392832baa0eb4f9e33b1c6e0",
      "MacAddress": "02:42:ac:17:00:04",
      "IPv4Address": "172.23.0.4/16",
      "IPv6Address": ""
    },
    "264529e624168c9a64a308f2bbadeb26eb3676e54935de87f42675c114721754": {
      "Name": "gogs-compose_app_1",
      "EndpointID": "3309a74372ac94794dcf1e6f7df21cda2a54c1b98a608e3a5b9922098831d894",
      "MacAddress": "02:42:ac:17:00:03",
      "IPv4Address": "172.23.0.3/16",
      "IPv6Address": ""
    },
    "6d04174448af71fd4a5de8735f5ad46e52145f933a3c5fbba3ea53052eccf168": {
      "Name": "gogs-compose_postgres_1",
      "EndpointID": "e48d315587aa881a2d77984cb856a446d1c5fb74f6fa2f95bb3be6671b90dba7",
      "MacAddress": "02:42:ac:17:00:02",
      "IPv4Address": "172.23.0.2/16",
      "IPv6Address": ""
    }
  },
  "Options": {},
  "Labels": {}
}
```

```
}  
]
```

4.2.3 Serverseite (Back-End)

Nun wird über die Serverseite gesprochen, die für die Kommunikation des Benutzers mit der Testumgebung verantwortlich ist.

Die Serverseite stellt viele Dienste wie folgt bereit:

Benutzererkennung In der westfälischen Hochschule Gelsenkirchen basiert die Speicherung der Hauptbenutzeridentitäten auf LDAP-Protokolls (Lightweight Directory Access Protocol) und gespeicherter LDAP-Datenbank. Dies hilft dabei, die Single Sign-On-Technologie (SSO) zu nutzen, die dadurch sich der Benutzer mit derselben ID und denselben Kennwörtern bei allen Anwendungen im selben Netzwerk anmelden kann. Um dies zu erreichen, sind 'spring-boot-starter-security' und 'spring-security-ldap' und 'unboundid-ldapsdk' in diesem Projekt verwendet.

Listing 4: pom.xml - security dependencies

```
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
  
<dependency>  
<groupId>org.springframework.security</groupId>  
<artifactId>spring-security-ldap</artifactId>  
</dependency>  
<dependency>  
<groupId>com.unboundid</groupId>  
<artifactId>unboundid-ldapsdk</artifactId>  
</dependency>  
<dependency>  
<groupId>org.springframework.ldap</groupId>  
<artifactId>spring-ldap-core</artifactId>  
</dependency>
```

Unter Verwendung der obigen Abhängigkeiten werden Sicherheitsrichtlinienkonfigurationen wie folgt geschrieben: Zunächst wird festgelegt, welche Dienste der Benutzer ohne Anmeldung bereitstellen kann und bei welchen Diensten er sich anmelden muss. Wenn der Benutzer seinen Benutzernamen und sein Kennwort eingibt, wird überprüft, ob die Authentifikation-Daten gültig sind, indem sie mit den in der LDAP-Datenbank verfügbaren Daten verglichen werden.

Es wird nun davon ausgegangen, dass der Benutzer gültige Daten eingegeben hat. dh diese Daten sind identisch mit den in der LDAP-Datenbank gespeicherten Daten. Dann wird ein Token

generiert und als Antwort auf die Authentifikations-Anfrage zurückgesendet. Dieses Token wird dann vom Frontend verwendet und mit jeder Anfrage gesendet.

Wird eine Anfrage geschickt, wird vom Backend dann überprüft, ob das Token gültig ist. Ist das der Fall, wird dann die Anfrage an den entsprechenden Controller weitergeleitet und damit die passende Funktion ausgeführt.

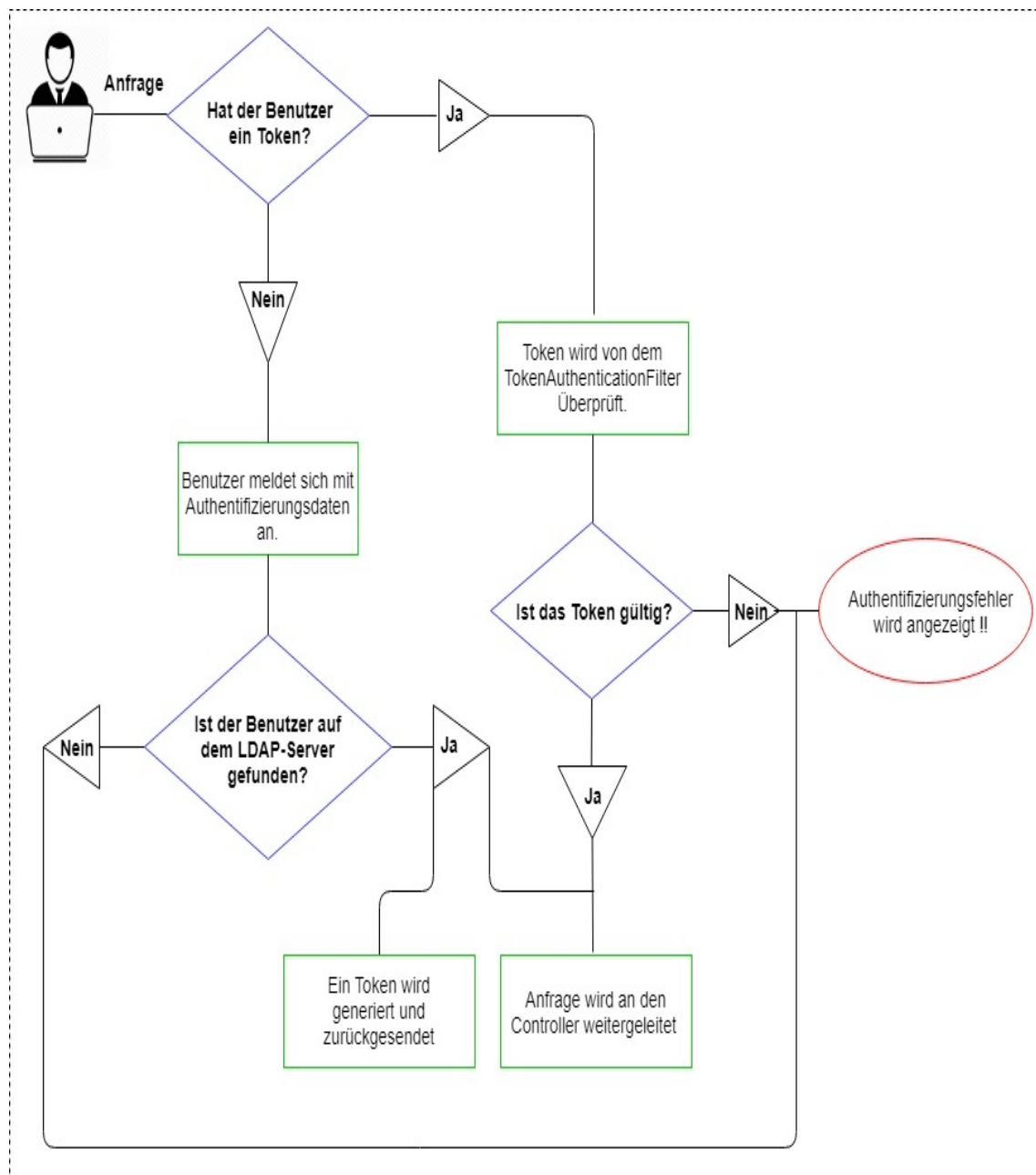


Abbildung 4: Ldap-Auth

Erstellung eines neuen Aufgabenblatts (Admin) Wie oben erläutert, ist die Erstellung eines neuen Aufgabenblatts in der Tat eine Einrichtung eines neuen Testbereichs, an dem die Studenten ihre Aufgabenlösungsdateien testen können. Mit anderen Worten, das Erstellen eines neuen Auf-

gabenblatts hängt vom Erstellen eines neuen Git-Repository ab, das denselben Namen wie das Aufgabenblatt hat. Zum Beispiel (Aufgabenblatt-1) über die Gogs-Oberfläche, die dem Administrator zur Verfügung steht. In diesem Git-Gogs-Repository werden alle zum Testen erforderlichen Dateien gespeichert. Anschließend wird ein Jenkins Multibranch-Pipeline-Job erstellt, der auch denselben Namen wie das Aufgabenblatt hat.

Schaffung von Jenkins Job (Admin) Für die Erstellung von Multibranch-Pipeline-Job bietet Jenkins eine REST-API an, die viele Dienste bereitstellt, z. B. das Erstellen und Löschen von Job und viele andere Dinge. Um diese Jenkins-API zu verwenden, muss eine Maven-Abhängigkeit in pom.xml hinzugefügt werden.

Listing 5: pom.xml - Jenkins-API

```
<dependency>
<groupId>com.offbytwo.jenkins</groupId>
<artifactId>jenkins-client</artifactId>
<version>0.3.7</version>
</dependency>
```

Zur Verdeutlichung wird erläutert, wie mit Jenkins-API einen neuen Job erstellt werden kann. Um einen Job in Jenkins zu erstellen, müssen dem Job ein Name und eine Reihe von Einstellungen nach Bedarf zugewiesen werden.

Der Jobname wird als String für die Jenkins-API-Methode angegeben und die Einstellungen (Konfigurationen) werden jedoch als XML-Datei dargestellt.

Listing 6: JenkinsServiceImp - createJob

```
@Override
public void createJob(String jobName, String jobXml) throws JenkinsException {
try{
JenkinsServer jenkinsServer = new JenkinsServer(new URI(jenkinsUrl), jenkinsUser,
jenkinsPassword);
jenkinsServer.createJob(jobName, jobXml, true);
}catch (Exception e){
throw new JenkinsException(e);
}
}
```

Die Idee dabei ist, dass in diesem Fall jeder Jenkins-Job die gleichen Einstellungen wie die anderen hat. Unabhängig davon, für welches Aufgabenblatt der Job erstellt wurde. Daher ist es ausreichend, bei jeder Erstellung eines Jenkins-Jobs dieselbe Konfigurationsdatei (XML-Datei) zu verwenden. Natürlich mit einigen Änderungen, die behoben werden, wie die Git Repository URL. Zu diesem Zweck wird ein Java-XML-Parser eingesetzt. Es sind viele Java XML-Parser verfügbar. In diesem Fall wird der DOM-Parser benutzt. Der DOM-Parser lädt die XML-Datei in den Speicher und gibt er dem Programmierer dann die Möglichkeit die XML-Datei Knoten für

Knoten durchlaufen, um das sie zu analysieren. DOM-Parser eignet sich für kleine Dateien. Mit zunehmender Dateigröße verlangsamt sich die Leistung und der Arbeitsspeicher wird erhöht.

Listing 7: modifyXmlFil

```
public String modifyXmlFile(String jobname) throws ParserConfigurationException,
    SAXException, IOException {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        File file = new File("config.xml");
        Document doc = builder.parse(file);
        Element element = doc.getDocumentElement();
        NodeList nodes = element.getChildNodes();
        NodeList sources = nodes.item(19).getChildNodes();
        Node data = sources.item(1);
        Node branchSource = data.getChildNodes().item(1);
        Node source = branchSource.getChildNodes().item(1);
        NodeList sourceNodes = source.getChildNodes();
        Node remote = sourceNodes.item(3);
        remote.setTextContent("http://172.23.0.3:3000/swissi/"+jobname+".git");
        DOMSource domSource = new DOMSource(doc);
        StringWriter writer = new StringWriter();
        StreamResult result = new StreamResult(writer);
        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer();
        transformer.transform(domSource, result);
        return writer.toString();
    } catch (Exception ex) {
        ex.printStackTrace();
        return null;
    }
}
```

Was jetzt fehlt, um den neuen Testbericht zu vervollständigen ist, die zum Testen notwendigen Dateien hochzuladen. Diese Dateien und ihre Rolle beim Abschließen des Testprozesses werden später ausführlich besprochen, aber jetzt geht es darum, wie sie hochgeladen und an die Remote-Testumgebung gesendet werden.

Zum Testen notwendigen Dateien hochzuladen Nach dem Hochladen werden die Dateien in ihrem jeweiligen Git-Repository gespeichert. Um dies zu veranschaulichen, ist es wünschenswert, paar Git-Konzepte zu erwähnen.

Git hat zwei Repository-Typen: lokal und remote. Das lokale Repository wird aus einem Remote-Repository erstellt (Clone). Das Senden (Pushing) von Dateien und Code-Änderungen vom lokalen Repository bis zum Remote-Repository erfolgt in mehreren Schritten:

- Add: Dadurch werden neue oder aktualisierte Dateien in die „Stage“ oder den „Index“ kopiert.
- Commit: Dadurch werden die bereitgestellten Dateien in das lokale Repository kopiert.
- Push: Dadurch werden Ihre Dateien vom lokalen Repo auf das Remote-Repo kopiert (nur die Änderungen, die die Remote-Repos nicht haben).

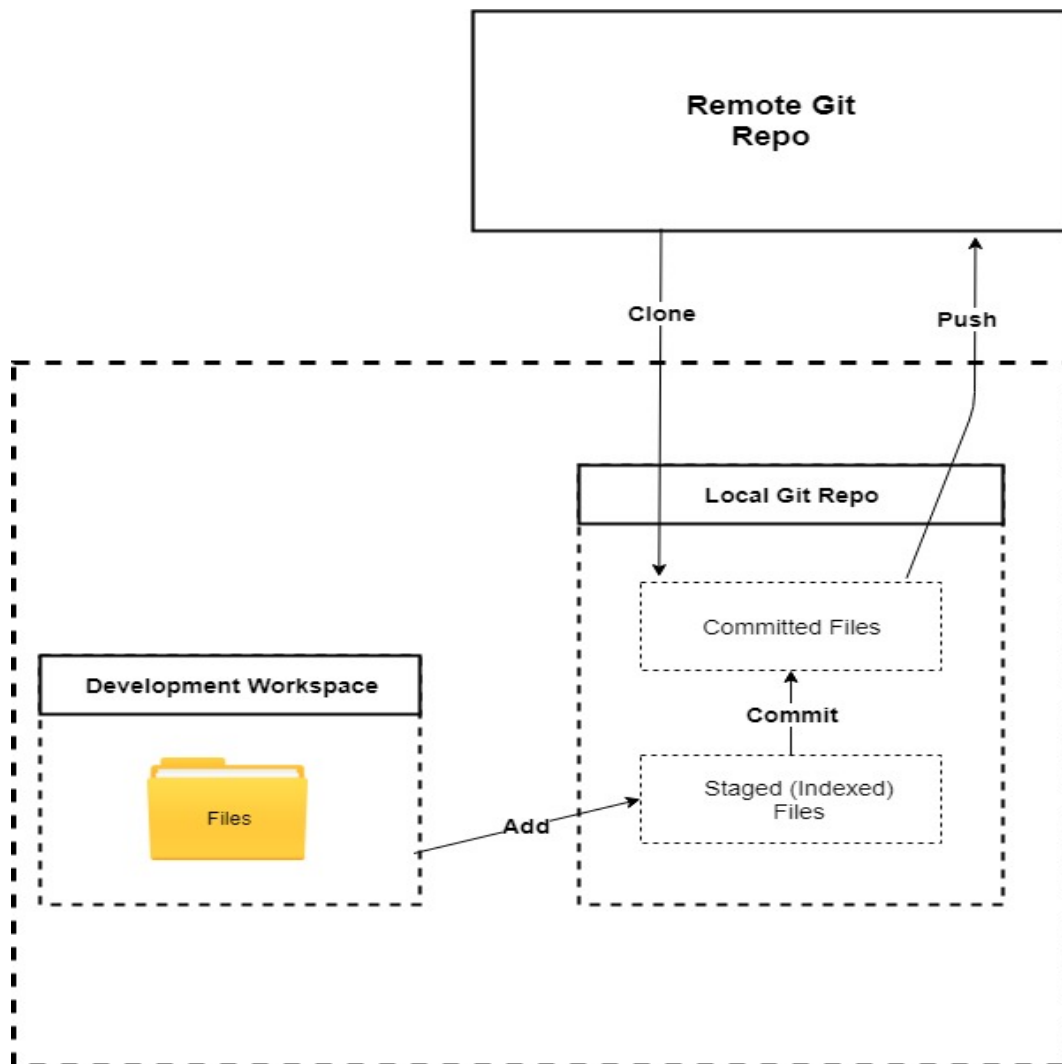


Abbildung 5: Git-Konzepte

Nun wird erklärt, wie das oben Genannte Git-Konzepte in diesem Fall angewendet wird.

Um Git aus einem Java-Programm zu verwenden, gibt es eine voll funktionsfähige Git-Bibliothek namens JGit. JGit ist eine relativ vollständige Implementierung von Git, die ursprünglich in Java geschrieben wurde und in der Java-Community weit verbreitet ist. Es gibt verschiedene Möglichkeiten, ein Projekt mit JGit zu verbinden und Code dagegen zu schreiben. Am einfachsten ist wahrscheinlich die Verwendung von Maven - die Integration erfolgt durch Hinzufügen des folgenden Snippets zum `<dependencies>` Tag in der pom.xml-Datei:

Listing 8: JGit-dependency

```
<dependency>
<groupId>org.eclipse.jgit</groupId>
<artifactId>org.eclipse.jgit</artifactId>
<version>3.5.0.201409260305-r</version>
</dependency>
```

Angenommen, der Administrator hat ein Remote-Git-Repository für ein bestimmtes Aufgabenblatt erstellt. Aus diesem Remote-Repository wird durch `Git.cloneRepository()` ein lokales Repository gemacht.

Listing 9: clone Repository

```
public static Git cloneRepo(String repositoryUrl, String gitLocalRepositoryPath,
    String tfsUser, String password) throws GitAPIException, IOException {
    Path path = Paths.get(gitLocalRepositoryPath);
    if (!Files.exists(path)) {
        CloneCommand cloneCommand =
            Git.cloneRepository().setURI(repositoryUrl).setDirectory(new
                File(gitLocalRepositoryPath));
        UsernamePasswordCredentialsProvider user = new
            UsernamePasswordCredentialsProvider(tfsUser, password);
        cloneCommand.setCredentialsProvider(user);
        Git repository = cloneCommand.call();
        return repository;
    } else {
        Git repository = Git.open(new File(gitLocalRepositoryPath));
        System.out.println("exist" + repository.toString());

        return repository;
    }
}
```

Dieses lokale Repository wird sich in einem BackEnd-Ordner namens 'aufgabenblaetter' befinden. Die zum Ausführen des Tests erforderlichen Dateien werden nach dem Upload in diesem BackEnd-Ordner in seinem eigenen lokalen Repository gespeichert. Wenn beispielsweise Dateien zum Test von Aufgabenblatt-1 hochgeladen werden, werden sie unter 'aufgabenblaetter/Aufgabenblatt-1' gespeichert.

Jetzt erfolgt der Versand (Push) von Dateien vom lokalen Repository zum Remote-Repository. Wie bereits erwähnt, besteht dieser Prozess aus drei Schritten. Zunächst werden die Änderungen mit der von der Git-Bibliothek bereitgestellten `add`-Methode zum Staging-Bereich hinzugefügt.

Listing 10: Add File To Index

```
public static boolean addFileToIndex(Git repository) throws IOException,
    GitAPIException {
    try {
        repository.add().addFilepattern(".").call();
        return true;
    } catch (Exception e) {
```

```
System.out.println(e.getMessage());
}
return false;
}
```

Zweitens wird ein neues Commit mit dem aktuellen Inhalt des Index erstellt.

Listing 11: commit Changes

```
public static boolean commitChanges(Git repository, String commitMessage) throws
    GitAPIException, IOException {
    repository.commit()
        .setAll(true)
        .setMessage("Commit changes to all files")
        .call();

    System.out.println("Committed all changes to repository at ");
    return true;
}
```

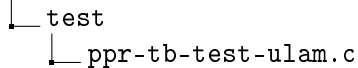
Drittens werden Commits vom lokalen Repository an das Remote-Repository übertragen.

Listing 12: Push Changes

```
public static boolean pushChanges(Git repository) throws GitAPIException,
    URISyntaxException, IOException {
    try
    {
        PushCommand pushCommand = repository.push();
        pushCommand.setCredentialsProvider(new UsernamePasswordCredentialsProvider("swissi",
            "Mh123456"));
        pushCommand.call();
        return true;
    }
    catch (GitAPIException e)
    {
        throw new RuntimeException(e);
    }
}
```

Die zum Ausführen des Tests erforderlichen Dateien Jedes Git-Repository, das die Quellcodeverwaltung für ein bestimmtes Aufgabenblatt in der Remote-Testumgebung darstellt, hat die folgende Form und als Beispiel den folgenden Inhalt.

```
Aufgabenblatt-N
├─ makefile
├─ Jenkinsfile
├─ src
│   └─ ulam.c
```

Jedes Repository, unabhängig davon, welchem Aufgabenblatt es zugewiesen ist, enthält immer ein Makefile, ein Jenkinsfile, einen src-Ordner und einen test-Ordner.

- **Src-Ordner:** Dies ist der unter-Repository-Ordner, in den der Student seine Aufgabenblattlösung hochladen wird. Dieser Vorgang wird nachstehend ausführlich erörtert.
- **Test-Ordner:** Hier wird die Datei mit den Testfällen vom Administrator hochgeladen.
- **Makefile:** Wie bereits erwähnt, gibt es eine vom Administrator hochgeladene Testdatei und eine andere vom Studenten hochgeladene Lösungsdatei. Dies bedeutet, dass mehrere Quelldateien kompiliert werden müssen, um das Projekt zu bauen und ein Testergebnis zu erhalten. Um diesen Prozess zu erleichtern und solches Hindernis zu überwinden wird in diesem Fall ein Makefile benutzt.

Ein Makefile wird mit dem UNIX-Dienstprogramm 'make' verwendet, um zu bestimmen, welche Teile eines Programms kompiliert werden sollen. Ein Makefile ist im Grunde ein Skript, mit dem das Dienstprogramm make die geeigneten Programmdateien auswählt, die kompiliert und miteinander verknüpft werden sollen. Das Dienstprogramm make protokolliert die letzten Aktualisierungen von Dateien, sodass nur die Dateien aktualisiert werden, die Änderungen enthalten. Es müssen jedoch auch alle Dateien kompiliert werden, die von den aktualisierten Dateien abhängig sind, was sehr zeitaufwändig sein kann. Mit Hilfe von makefile automatisiert das Dienstprogramm make diese Kompilierung, um sicherzustellen, dass alle aktualisierten Dateien - und nur diese - kompiliert werden und dass die neuesten Versionen der Dateien mit dem Hauptprogramm verknüpft sind.

Ein Makefile enthält drei Arten von Informationen für das make-Programm: ein Ziel oder auch target genannt wird, das normalerweise der Name einer Datei, die von einem Programm generiert wird. Beispiele ausführbare Dateien oder Objektdaten. Es enthält auch Abhängigkeiten und Regeln (Befehle), die angeben, wie das Ziel aus den Quellen erstellt werden soll.

- **Jenkinsfile:** Ist eine Textdatei, die die Definition einer Jenkins-Pipeline enthält und in die Quellcodeverwaltung eingecheckt wird. Die Rolle dieser Datei wird nachfolgend erläutert.

Bevor zu den folgenden Abschnitten übergangen wird, wird erläutert, wie der Student seine Lösungsdatei hochladen kann.

Student-Lösungsdatei hochladen Der Student lädt die zu testende Datei über die Frontend-Seite. Er wählt die Datei aus und drückt dann die Upload-Taste. Das Backend erhält also eine Anfrage zum Hochladen von Dateien. Um diese Aufgabe auszuführen, muss das Backend die folgenden Schritte ausführen: Angenommen, der Benutzer möchte seine Aufgabenblatt-1-Lösung zum ersten Mal testen. Anschließend wird aus dem lokalen Git-Repository von Aufgabenblatt-1 ein Branch mit dem Benutzernamen als Namen erstellt. Dann wird Die Datei durch Ausführen der Methoden Jgit-add und -commit an den angegebenen Branch gesendet. Der Benutzer hat nun die Möglichkeit, entweder den Test abubrechen, was bedeutet, dass die hochgeladene Datei

gelöscht wird, oder fortzufahren und den Testknopf zu drücken. Angenommen, der Benutzer hat den Test abgeschlossen und die Testtaste gedrückt. Dann wird der Student-Branch durch eine Jgit-Push-Methode an das Remote-Repository übertragen. Zur weiteren Verdeutlichung wird eine grafische Darstellung angezeigt, die diesen Vorgang erläutert.

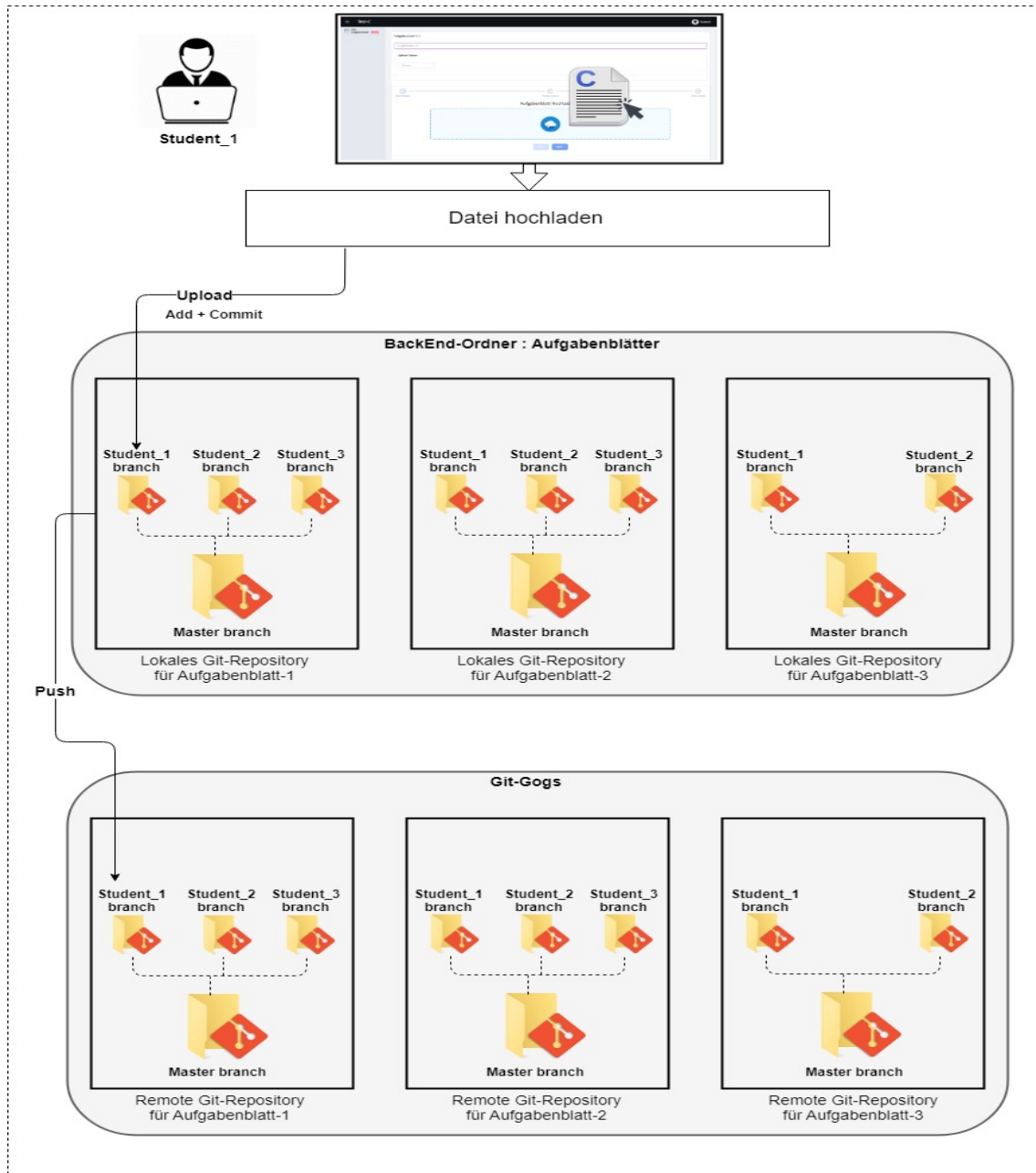


Abbildung 6: Hochladen von Lösungsdatei

Für jedes Aufgabenblatt wird ein lokales und ein remote Git-Repository erstellt. Für jeden Student wird aus jedem Aufgabenblatt-Repository ein Branch erzeugt. Es wird beispielsweise davon ausgegangen, dass der Administrator drei Aufgabenblätter hinzufügt. Das heißt, es werden drei Git-Repositorys erstellt, von denen jedes ein Arbeitsblatt darstellt. Drei Studenten möchten ihre Lösungen testen. Zwei von ihnen haben die drei Übungsblätter ge-

löst und einer hat nur das erste und das zweite Aufgabenblatt gelöst. Anschließend werden aus den für die Aufgabenblätter 1 und 2 erzeugten Git-Repositorys drei Branchs erstellt. Weil die drei Studenten die ersten beiden Übungsblätter gelöst und getestet haben. Aus dem dritten Repository werden jedoch nur zwei Branchs erzeugt, da nur zwei Studenten ihre Aufgabenblatt-3-Lösung getestet haben.

Nun wird besprochen, wie nach Erhalt der Lösungsdatei der Test Prozess automatisch gestartet und das Ergebnis dem Student angezeigt wird.

Test Starten (jenkins Job Build) Wie bereits erwähnt, wird für jedes Repository automatisch ein Jenkins-Multibranch-Pipeline-Projekt generiert. Ein Multibranch-Pipeline-Projekt durchsucht einfach das Quellcode-Repository und erstellt automatisch einen Pipeline-Job für jedes Branch, das eine Jenkinsfile-Datei enthält.

Eine Jenkinsfile-Datei ist eine Textdatei, die die Pipeline als Code enthält, der den gesamten Job-Workflow festschreibt. Daher lädt der Administrator diese Datei bei jeder Erstellung eines neuen Aufgabenblatts hoch. Dieser Vorgang ist in der folgenden Grafik dargestellt.

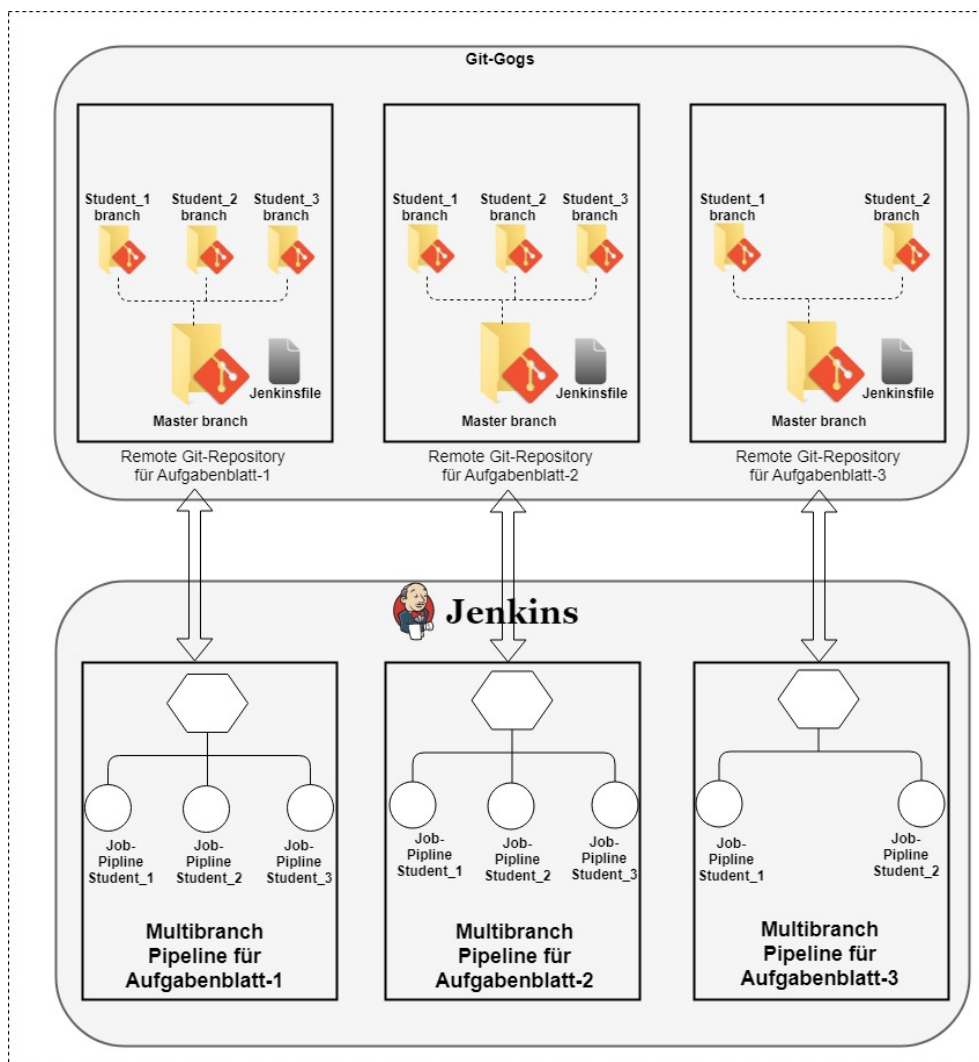


Abbildung 7: Multibranch Pipeline Projekt

In diesem Fall besteht die Job-Pipeline nur aus einer Phase (stage), in der das Makefile ausgeführt werden sollte, um den Testprozess zu starten. Genau das, was in der Jenkinsfile-Datei steht.

Listing 13: Jenkinsfile (Job-Pipeline-Skript)

```
pipeline{
agent any
stages {
    stage ('Test with make file') {
        steps {
            sh 'make'
        }
    }
}
}
```

'make' ist der Befehl zum Ausführen von makefile.

Jetzt werden sich die Ideen neu gruppieren und klar aufbauen. Der 'Student-1' hat die zu testende Datei hochgeladen. Zum Beispiel die Lösung von Übungsblatt-1. Wenn er seine Aufgabenblattlösung zum ersten Mal testet. Ein Branch namens 'Student-1' wird vom Aufgabenblatt-1-Repository erzeugt. Dann wird die Datei an diesem Branch übergeben. Wenn dies nicht der Fall ist und eine Branch vorhanden ist, wird es einfach verwendet. Dann wird Student-1-Branch auf Remote verschoben. Danach sendet das BackEnd eine Job-Build Anfrage an den Jenkins-Server.

Listing 14: Job-Build Anfrage

```
public int buildJob(String jobName) throws JenkinsException {
int buildNumber = 0;
try{
JenkinsServer jenkinsServer = new JenkinsServer(new URI(jenkinsUrl), jenkinsUser,
jenkinsPassword);
Map<String, Job> jobs = jenkinsServer.getJobs();
JobWithDetails job = jobs.get(jobName).details();
buildNumber = job.getNextBuildNumber();
job.build(true);
}catch (Exception e){
throw new JenkinsException(e);
}
return buildNumber;
}
```

4.2.4 Interne Prozesse

4.3 Alternativen

4.4 Ablauf des Testverfahrens

4.5 Installationsanleitung

4.6 Verwendung

5 Zusammenfassung

Abbildungsverzeichnis

1	Docker Architektur	4
2	Architektur Test-C-Plattform	6
3	Front-End	7
4	Ldap-Auth	11
5	Git-Konzepte	14
6	Hochladen von Lösungsdatei	18
7	Multibranch Pipeline Projekt	19

6 Quellcodeverzeichnis

7 Quellenverzeichnis