Technische Hochschule Brandenburg
Continuous Delivery & DevOps SoSe 2021

Ein Banksystem mit CI Pipeline

Projekt vorgelegt von Gruppe 2:
Kanstantsin Liakh
Jan Szataniak
Consuelo Alberca Susano
Lau Kwai Fan

Betreuer: Prof. Dr. Kai Jander

Brandenburg/H., den 15. Juli 2021

Table of Contents

Übersicht über das Projekt

Ziel

Das Ziel dieses Projekts ist, ein Banksystem mit Kommandozeilensteuerung mit der entsprechenden Continuous Integration Pipeline zu entwickeln.

Entwicklung des Projekts

Das Projekt wurde in drei Phasen entwickelt. Die folgende Tabelle stellt die Aufgaben in jeweiliger Phase dar.

Phase	Aufgaben
Entwicklung der Basisversion	• Erstellung der Klasse für die Basiskonten, für das Banksystem und für die Funktionen der Standard-Version • Entwicklung der Unit Tests für die entsprechenden Klasse • Erstellung der Jenkins-Pipeline mit 4 Schritte
Entwicklung der ProfessionalVersion	• Erstellung der Klasse für das professionelle Konto, für die Funktionen der Professional-Version • Entwicklung der Unit Tests für die entsprechenden Klasse • Erweiterung der Jenkins-Pipeline, um zwei .jar-Dateien für die zwei Versionen des Banksystems zu erstellen
Download Server und Server-Deployment	• Bearbeitung mit dem Download-Server bei http://file.io, um die zip Datei des Projekts automatisch hochzuladen • Erweiterung der Jenkins-Pipeline, um das Projekt bei http://file.io automatisch hochzuladen • Verfügung des Banksystems über einen TCP-Port

Aufbau des Banksystems

Das Banksystem wurde mit Java entwickelt und besteht aus 11 Klassen. Die folgenden Klasse sind für die Standard-Version und die Professional-Version notwendig:

- BaseAccount Klasse – eine abstrakte Klasse, die beliebig viele Klasse verschiedener Kontotypen erzeugen kann. In diesem Banksystem erben die BankAccount Klasse und die ProfessionalBankAccount Klasse diese Klasse.
- BankAccount Klasse – repräsentiert die Bankkonten, mit der eine Kunde der Standard-Version die Basisfunktionen benutzen kann.
- BankAccountService Klasse – repräsentiert die Basisfunktionen welche verfügbar für beide Versionen sind.
- CreateAccount Klasse – repräsentiert die Funktion, mit der ein Kunde ein Bankkonto der Standard-Version erstellen kann. Ein Kunde kann beliebig viele Konten erstellen.
- BankSystem Klasse – repräsentiert das Banksystem, das die Geschäftslogik einer Bank ausführt. Kunden können dadurch verschiedenen Bank Services benutzen.
- BankSystemServer Klasse – repräsentiert den Server, mit der das Banksystem über einen TCP-Port bedienbar ist.

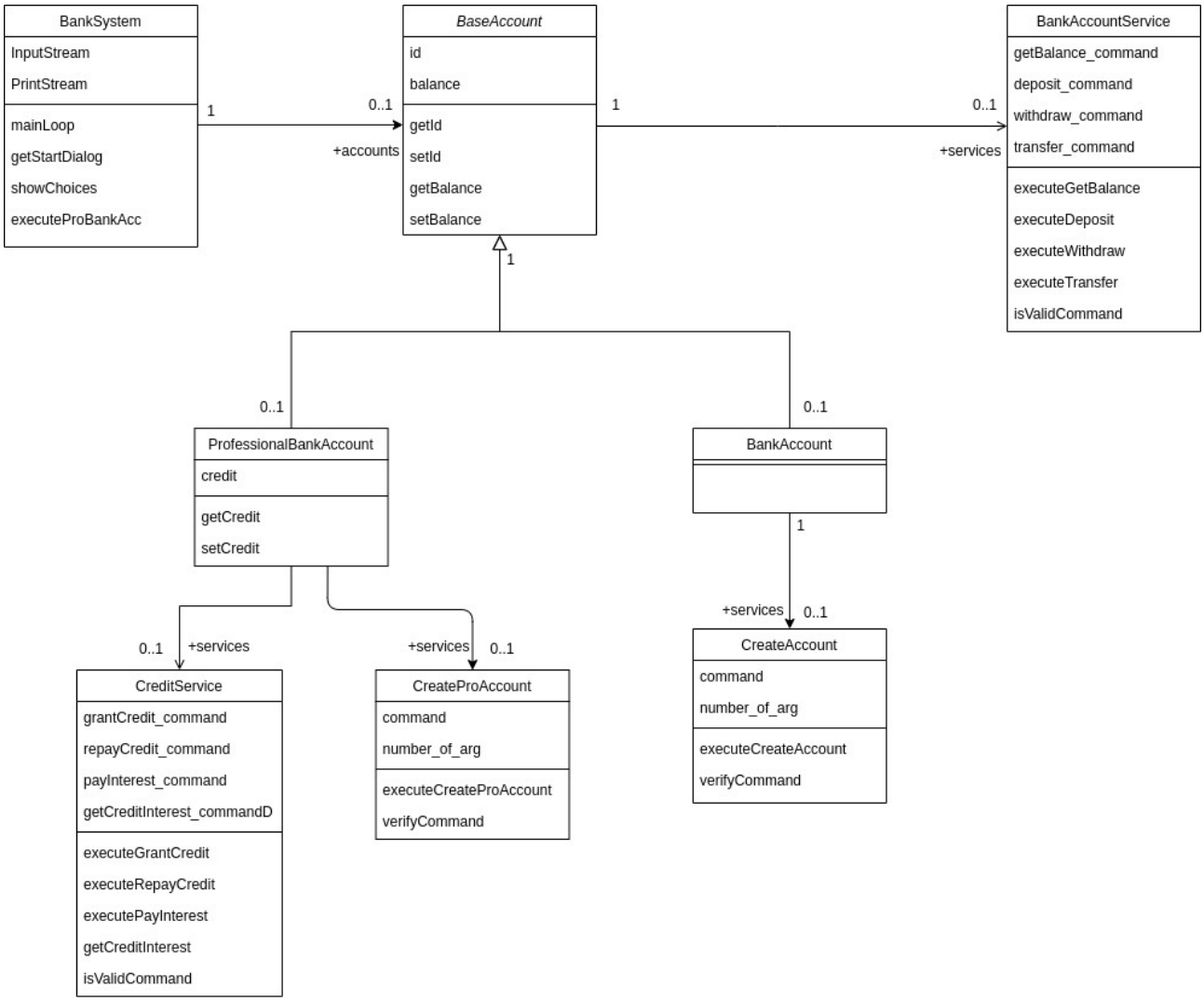
- Main Klasse – beinhaltet die main Methode, um die Instanze eines Banksystem der Basisversion zu erzeugen und das Geschäftslogik zu starten.

Die folgenden Klasse sind ausschließlich für die Professional-Version entwickelt:

- ProMain Klasse - beinhaltet die main Methode, um die Instanze eines Banksystem der Professional-Version zu erzeugen und das Geschäftslogik zu starten.
- ProfessionalBankAccount Klasse - repräsentiert die Bankkonten, mit der eine Kunde der Professional-Version die zusätzliche Funktionen z.B. Kreditservice benutzen kann.
- CreditService Klasse - repräsentiert die besonderen Funktionen z.B. Kredite vergeben, Kredit zurückzahlen. Diese Funktionen sind nur für die Professional-Version verfügbar.
- CreateProAccount Klasse - repräsentiert die Funktion, mit der ein Kunde ein Konto der Professional-Version erstellen kann. Ein Kunde kann beliebig viele Konten erstellen.

Klassendiagramms

Die Hauptkomponenten des Banksystems kann durch das folgenden Klassendiagramms dargestellt.



Funktionen bei der Basisversion

Die Funktionen bei der Basisversion sind wie folgt:

- das System einloggen
- ein Konto oder mehrere Konten erstellen
- einen Betrag in einem existierenden Konto einzahlen
- Bilanz eines Kontos der Kunde anschauen
- einen Betrag von einem existierenden Konto abheben
- einen Betrag von einem existierenden Konto auf einem anderen existierenden Konto der Kunde bzw. auf einem anderen existierenden Konto einer anderen Kunden überweisen
- vom System ausloggen

Funktionen bei der Professionalversion

Neben die Funktionen bei der Basisversion wurden zusätzliche Funktionen für die Professionalversion zur Verfügung gestellt:

- Kredit vom Bank gewähren und dem Konto wird der Betrag des Kredits gutgeschrieben
- existierende Kredit zurückzahlen
- Zinsen, die von den Konten abgezogen ist, berechnen und anschauen

Nutzung des Bankssystems

Um das Projekt lokal zu benutzen kann man durch die Linke https://github.com/LauKwaiFanHK/bank_system.git das Projekt in eine IDE herunterladen. Danach muss man die „finished-server“-Branch laden und das Projekt auf diese Branch ausführen. Nach der Ausführung des Projekts muss man im Terminal gehen und den Befehl „telnet localhost 4444“ oder „nc localhost 4444“ ausführen. Das folgende Bild wird dann angezeigt:

```
(base) fan@fan-UX32VD:~$ telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hey,it is your bank assistant,we are happy to see you!
To use the bank services, please login.


0. Please type 'login' to login
1. To create a new account, type 'create ' and press enter.
2. To transfer money from account A to account B, type 'transfer ', 'loginName ', <accountAId>, 'receiverName', <accountBId
>, <amount> and press enter.
3. To get balance from your account, type 'getBalance ', <accountId> and press enter.
4. To deposit money into your account, type 'deposit ', <accountId> and <amount> and press enter.
5. To withdraw money from your account, type 'withdraw  ','loginName ', <accountId> and <amount> and press enter.
6. If you dont want to proceed, type 'bye' and press enter.
7. To apply for credit, type in 'grantcredit ', <accountId> and <amount> and press enter.
8. To repay credit, type in 'repaycredit ', <accountId> and <amount> and press enter.
9. To check the amount of credit interest being deducted from your account, type in 'getCreditInterest ', <accountId> and p
ress enter.
```

Man kann dann die Anweisung auf diesem Interface folgen, um einzuloggen und anschließend die Bankservices zu benutzen.

Wenn man sein Session beendet würde, gibt man dann den Befehl „logout“ ein. Danach kann ein andere Kunde einloggen.

```
logout
Bye, Fan!
```

Wenn man das Programm beendet würde, gibt man den Befehl „bye“ ein. Die Verbindung auf dem TCP-Port wird dann abstürzen und das Programm wird beenden.

```
bye
Connection closed by foreign host.
```

Andere Eigenschaften des Banksystems

- Das Banksystem wurde möglichst interaktiv für die Kunden aufgebaut. Dafür wurden Dialoge z.B. Begrüßungstext, eine Nachricht nach Ausführung jedes Befehls den Kunde angezeigt. Bespiele sind wie folgt:

```
Hey,it is your bank assistant,we are happy to see you!
To use the bank services, please login.
```

```
Hi, Fan!
```

```
create
New account is created. The account number is 0 .
```

- Das Banksystem erlaubt beliebig viele Kunden einzuloggen und die Funktionen zu benutzen. Für eine Personalisierungsservice kann jede Kunde seine Namen eingeben. Zur Überweisung eine Betrage auf einen anderen Kunden muss man den Namen des Absenders und den Namen des Empfänger eingeben, beispielsweise:

```
transfer Jane 1 Fan 0 15
15 euro is transferred from account number: 1 to account number: 0
```

  Dazu darf nur der eingeloggten Kunde die Bilanz seines Kontos oder seiner Konten anschauen. Ein Kunde darf nicht die Bilanz Kontos von anderen Kunden abrufen. Wenn so passiert wird eine Fehlermeldung angezeigt.

```
getBalance 1
Your account balance is 25.0 euro.
```

```
getBalance 0
Invalid account id: 0
```

- Ein Kunde der Professional-Version kann Kredit vom Bank anfragen. Beim Gewähren eines Kredits wird die Zinsen anschließend automatisch vom Bankkonto des Kunden abgezogen (Die globale Zinssatz ist als 3,5% angenommen.). Danach wird sein Bankkonto gutgeschrieben, beispielsweise:

```
deposit 0 50
Your account balance is 50.0 euro.
```

```
grantcredit 0 20
Your credit balance is 20.0 euro.
```

```
getBalance 0
Your account balance is 69.3 euro.
```

Unit Tests

Für die Unit Tests wurde die Framework Junit benutzt. Unit Tests wurden für die BankAccountService Klasse, BankAccountTests Klasse, CreateAccount Klasse, CreateProAccount Klasse, CreditProAccount Klasse , CreditService Klasse und die ProfessionsalBankAccount Klasse entwickelt. Denn die BankSystem Klasse bezieht sich auf der Präsentationsschicht der Applikation, eine Integrationstest anstatt eine Unit Test ist für diese Klasse geeigneter. Deswegen wurde keine Unit Test dafür entwickelt. Denn eine abstrakte Klasse kann nicht instantiiert werden und die Methode davon wurden bereits in Unit Tests für die Kinder-Klasse abgedeckt, deswegen wurde keine Test dafür geschrieben. Eine Beispiel für die Test der BankAccount Klasse ist wie folgt:

```java
package myprojecttests;

import myproject.BankAccount;

public class BankAccountTests {
    @Test
    public void tests() {
        BankAccount bankAccount = new BankAccount();
        bankAccount.setId(8);
        bankAccount.setBalance(50);
        assertEquals(50, bankAccount.getBalance(), 0.01);
        assertEquals(8, bankAccount.getId(), 0.01);
    }
}
```

Aufbau des Projekts mit Gradle

Das Build-System Gradle wird benutzt, um die benötigten Abhängigkeiten herunterzuladen und anschließend das Projekt zu kompilieren. Die „build.gradle“ Datei dient als ein Skript zur Konfigurierung des Build-Prozesses. Darin sind 2 Plugins spezifiziert:

```
plugins {
    id 'java'
    id 'application'
}
```

Die Java Plugin stellt verschiedene Fähigkeiten zur Testen und Bündelung bereits z.B. jar zur Erstellung der JAR Datei, javadoc zur Erzeugung der API Dokumentation, compileTestJava zur Kompilierung des Quellcode mittels des Kompilierers. Die Application Plugin erleichtert den Start der Applikation in der lokalen Entwicklungsumgebung und packt das Banksystem als zip Datei ein. Die main Klasse dient als einen Eintrittpunkt für die Applikation und muss unbedingt spezifiziert werden.

```
application {
    // package and class to be main method
    mainClassName = 'myproject.Main'
}
```

Außerdem wurden Repository und zusätzliche Abhängigkeiten spezifiziert:

```
repositories {
    mavenCentral()
}

dependencies {
    implementation group: 'org.apache.commons', name: 'commons-lang3', version: '3.9'
    testImplementation group: 'junit', name: 'junit', version: '4.12'
}
```

Da die Open-Source Abhängigkeiten für das Java Projekt für die Entwicklung des Banksystems notwendig sind, wurde die weit verbreitete Maven Central Repository benutzt. Gradle ladet beim Build des Projekts die benötigten Abhängigkeiten von der Repository. Als eine Konfiguration Software Bibliotek bietet org.apache.commons die Applikation eine Konfigurierungsinterface an. Dadurch kann die Java Applikation Konfigurierungsdatei, die aus unterschiedlichen Quelle sind, auslesen. Zur Unit Tests is JUnit auch in disem Build-Skript spezifiziert.

Um 2 Versionen des Banksystems auszuliefern wurden 2 unterschiedlichen Gradle Tasks zur Erzeugung zwei jar-Dateien spezifiziert:

```
jar {
    manifest {
                // set location of main class
                attributes 'Main-Class': application.mainClassName
                // places with classes which may be needed by JVM
            attributes 'Class-Path': 'commons-lang3-3.9.jar'
    }
        exclude 'myproject/Main.class'
        exclude 'myproject/CreateAccount.class'
    baseName = 'ProfessionalVersion'
}

task jarv2(type: Jar) {
    manifest {
        attributes 'Main-Class': 'myproject.Main'
        attributes 'Class-Path': 'commons-lang3-3.9.jar'
    }
    from sourceSets.main.output
    exclude 'myproject/ProfessionalBankAccount.class'
    exclude 'myproject/CreateProAccount.class'
    exclude 'myproject/ProMain.class'
    baseName = 'StandardVersion'
}

jar.dependsOn(jarv2)
```

Bei der ersten Gradle Task geht es darum, eine jar-Datei mit dem Name „ProfessionalVersion“ zu erzeugen. Die wesentliche Information ist der Pfad zur ProMain Klasse, die die main Methode für diese Version beinhaltet. Diese Information wurde innerhalb manifest spezifiziert. Die zwei Klasse (Main und CreateAccount) sind besonders für die Basisversion bereitgestellt und sind von dieser Version mittels des Gradle Property „exclude“ ausgeschlossen. Dadurch werden alle für-die-Professionalversion-notwendige Klasse in eine jar-Datei eingepackt.

Bei der zweiten Gradle Task geht es darum, eine andere jar-Datei mit dem Name „StandardVersion“ zu erzeugen, welche alle Klasse außer der 3 Klasse (ProfessionalBankAccount, CreateProAccount und ProMain) beinhaltet. Dadurch sind für diese Version die Kreditservices für die Kunden ausgeschlossen.

Jenkins CI Pipeline

Die Jenkins CI Pipeline ermöglicht die Automatisierung der Auslieferung der Applikation. Das folgende Bild stellt die Definition einzelner Schritt in einem Konfigurationsskript „Jenkinsfile“ dar:

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'echo "Building.."'
                sh './gradlew build'
            }
        }
        stage('Test') {
            steps {
                sh 'echo "Testing.."'
                sh './gradlew test'
            }
        }
        stage('Document') {
            steps {
                sh 'echo "Documenting"'
                sh './javadocs.sh'
            }
        }
        stage('Pack') {
            steps {
                sh 'echo "Packing"'
                sh 'zip -r project_package.zip docs src/main/java/myproject src/test/java/myprojecttests build/libs/StandardVersion.jar build/libs/ProfessionalVersion.jar readme.md'
            }
        }
        stage('Upload to download server') {
            steps {
                sh 'echo "Uploading"'
                script {
                    response = sh(script: 'curl -X POST "https://file.io/" -H  "accept: application/json" -H  "Authorization: Bearer 3EAL7UG.CZJ8SDR-S514XJP-M5NPK94-55HBRR2" -H  "Content-Type: multipart/form-data" -F "expires=2021
                    echo "curl response: ${response}"
                    currentBuild.displayName = "Bank system build"
                    currentBuild.description = "The bank system project from group2 can be downloaded using the link: " + response
                }
            }
        }
        stage('Run') {
             steps {
                sh './gradlew run'
            }
         }
```

Festlegung der Pipeline

Die Jenkins CI Pipeline legt insgesamt 6 Schritte fest:

- Schritt 1: Bauen des Projekts („Build“)
  - Mit dem Befehl „./gradlew build“ wird das Projekt mit Gradle Wrapper aufgebaut. Gradle Wrapper wird bevorzugt, weil Gradle nicht bedingt auf dem Rechner installiert werden muss, um ein Build auszuführen. Als das Gradle Wrapper Skript in Git eingecheckt wurde, können alle Teamkollegen das Projekt ohne die installierte Gradle, gleichmäßig aufbauen.
- Schritt 2: Ausführung der Tests („Test“)
  - Mit dem Befehl „./gradlew test“ werden alle Unit Tests durch Gradle Wrapper automatisch getestet. Da die Quellcode in Laufe der Zeit weiter entwickelt und verändert wurde, waren die ursprüngliche Unit Tests nicht mehr geeignet. Diese Schritt ist besonders hilfreich für uns, Fehler mit Details zu erkennen. Damit konnten wir die Testcode entsprechend verändern und die Tests wurden dadurch die aktualisierte Quellcode angepasst.
- Schritt 3: Generierung der Javadocs der Java-Klassen („Document“)
  - Mit dem Befehl „./javadocs.sh“ werden alle Javadocs automatisch generiert. Die involvierten Klasse sind durch das Skript javadocs.sh spezifiziert. Also die Javadoc aller Java Klassen innerhalb die Paket „myproject“ werden dadurch generiert.

- Schritt 4: Verpackung der notwendigen Dateien des Projekts („Pack“)

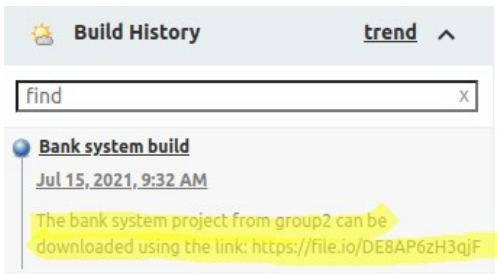
```
stage('Pack') {
    steps {
        sh 'echo "Packing"'
        sh 'zip -r project_package.zip docs src/main/java/myproject src/test/java/myprojecttests build/libs/bank_system.jar build/libs/ProfessionalVersion.jar readme.md'
    }
}
```

  - Mit dem obigen Befehl werden alle notwendigen Dateien einschließlich der Javadocs, der Quellcode, der Testcode, der 2 jar-Dateien und der readme.md Datei in einer .zip-Datei „project_package.zip“ eingepackt. Diese .zip-Datei wird beim nächste Schritt benutzt.

- Schritt 5: Hochladen der .zip-Datei auf einen Filehosting-Dienst („Upload to download sever“)

```
stage('Upload to download server') {
    steps {
        sh 'echo "Uploading"'
        script {
            response = sh(script: 'curl -X POST "https://file.io/" -H  "accept: application/json" -H  "Authorization: Bearer 3EAL7UG.CZJ8SDR-S514XJP-M5NPK94-55HBRR2" -H
            "Content-Type: multipart/form-data" -F "expires=2021-08-31" -F "maxDownloads=1" -F "autoDelete=true" -F "file=@project_package.zip" | jq -r ".link"',
            returnStdout: true).trim()
            echo "curl response: ${response}"
            currentBuild.displayName = "Bank system build"
            currentBuild.description = "The bank system project from group2 can be downloaded using the link: " + response
        }
    }
}
```

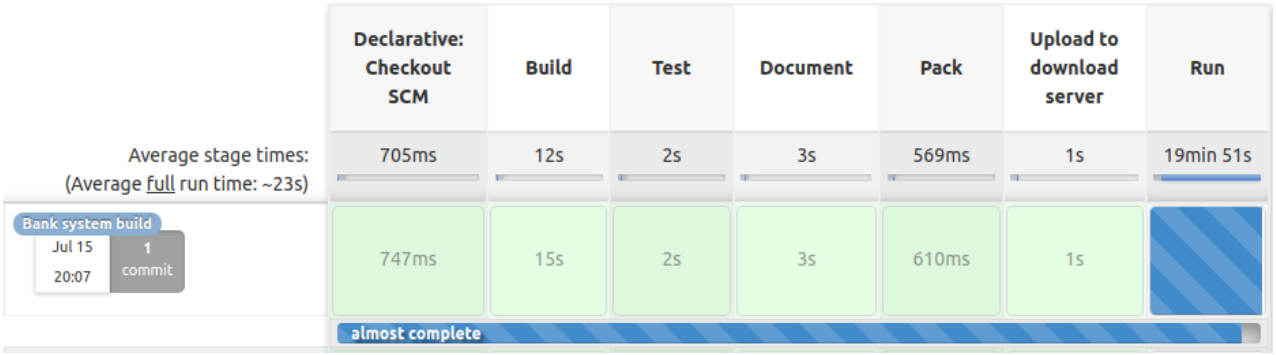
  - Der obigen Befehl dient dazu, eine POST Anfrage mittels CURL den Filehosting-Dienst http://file.io zu verschicken. Dadurch wird die generierte zip-Datei auf diesem Download-Server hochgeladen. Bei einem erfolgreichen Hochladen der zip-Datei wird eine JSON Antwort mit HTTP Status Code 200 zurückgegeben. Diese JSON Antwort beinhaltet eine Linke zum Herunterladen der zip-Datei, welche mittels jQuery extrahiert wird. Diese extrahierte Linke wird in einem Variable gespeichert. Diese Variable wird danach bei der Festlegung einer Build-Description für Jenkins benutzt. Nach einer Ausführung der Pipeline kann man dann die Linke beim Beschreibungsfeld auf dem Jenkins Interface finden und die zip-Datei entsprechend finden.



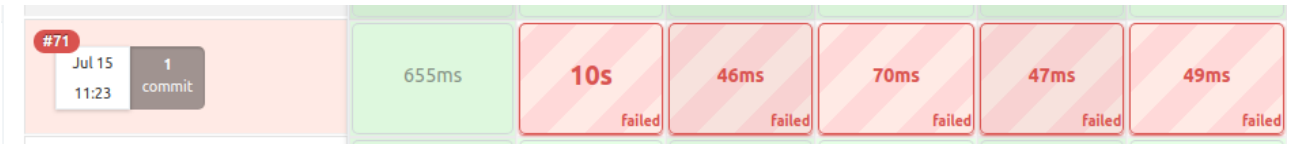
- Schritt 6: Ausführung des Programms
  - Nach der vorherigen Schritte wird das Banksystem durch den folgenden Befehl automatisch ausgeführt:

```
stage('Run') {
     steps {
        sh './gradlew run'
     }
 }
```

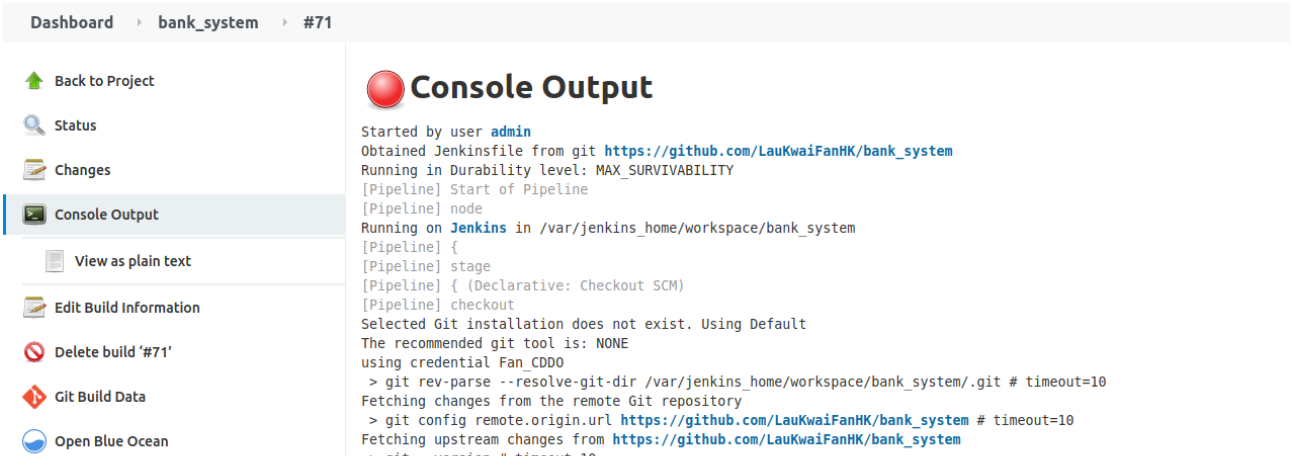
Ausführung der CI Pipeline in Jenkins
Da zip und jQuery zur Ausführung der Pipeline benutzt werden, ist es notwendig, diese zwei Pakete im Voraus herunterzuladen. Alle Schritte, die in Jenkinsfile definiert wurden, werden mit dem vordefinierten Befehl ausgeführt. Jenkins bietet eine schöne Interface an, um den Fortschritt der Ausführung anzuschauen.

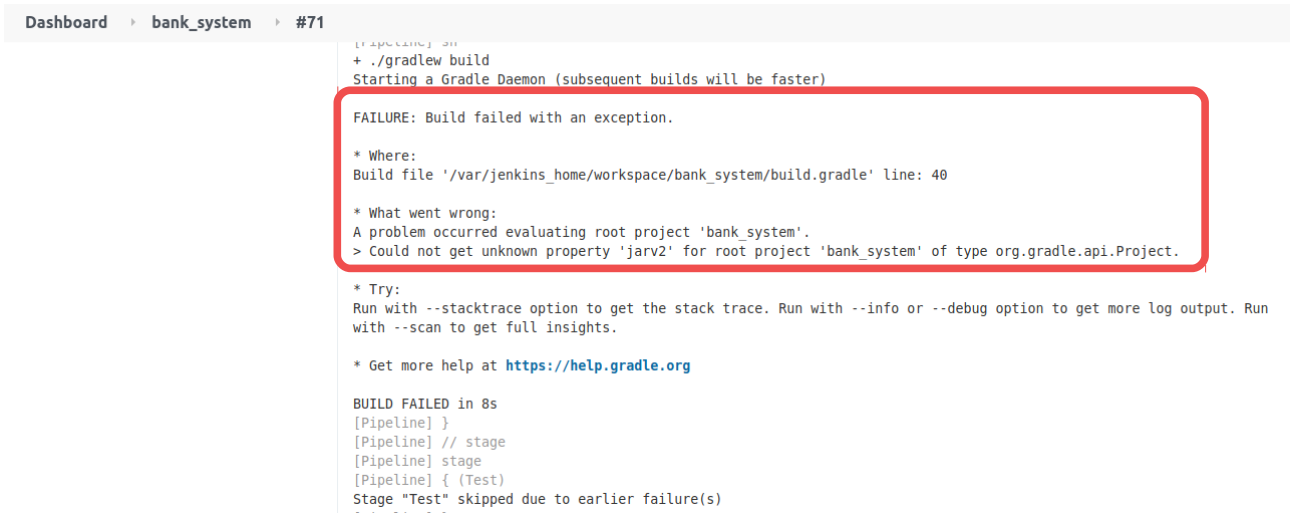


Was besonders hilfreich bei Jenkins ist, dass das Auslieferungsprozess detailliert geloggt wird. Bei einer fehlerhaften Ausführung der Pipeline kann man sofort den Fehler verstehen und es entsprechend korrigieren. Beispielweise ist die folgende Ausführung der Pipeline gescheitert:



Wir konnten schnell die Ursache aus dem Console Output erkennen:





Besonderheiten beim Vorgehen

Aspekte die schwierig umzusetzen waren

1. Erzeugung der zwei jar-Dateien

Wir hatten Schwierigkeit, zwei jar-Dateien für die Standard-Version und für die Professional-Version zu erzeugen. Wir haben für die Professional-Version eine Jar Task definitert, um eine jar-Datei mit dem Name „ProfessionalVersion.jar“ zu erzeugen. Danach haben wir für die Standard-Version eine zusätzliche Gradle Task erstellt, indem wir alle Klassen außer die nur-für-die-ProfessionVersion-Klassen in einer andere jar-Datei eingepackt haben. Das folgende Bild zeigt die Definition im build.gradle Datei:

```
application {
    // package and class to be main method
    mainClassName = 'myproject.Main'
}


jar {
    manifest {
            // set location of main class
            attributes 'Main-Class': application.mainClassName
            // places with classes which may be needed by JVM
        attributes 'Class-Path': 'commons-lang3-3.9.jar'
    }
    baseName = 'ProfessionalVersion'
}

task jarv2(type: Jar) {
    manifest {
        attributes 'Main-Class': application.mainClassName
        attributes 'Class-Path': 'commons-lang3-3.9.jar'
    }
    exclude 'src/main/java/myproject/ProfessionalBankAccount.java'
    exclude 'src/main/java/myproject/CreditService.java'
    exclude 'src/main/java/myproject/CreateProAccount.java'
    baseName = 'StandardVersion'
}
```

Allerdings mit der obigen Definition konnte die Standard-Version.jar nicht richtig erzeugt werden. Nach mehrerer Veränderungen der Task Definition konnten wir dieses Problem nicht löschen. Eventuell haben wir die Code umstrukturiert, indem wir eine neue Klasse „ProMain“ hinzugefügt haben. ProMain Klasse beinhaltet eine main Methode für die Professional-Version. Dadurch wird eine Klasse mit einer main Methode für die jeweilige Version bereitgestellt. Also wird die Standard-Version mit der „Main“ Klasse ausgeführt und die Professional-Version wird mit der „ProMain“ Klasse ausgeführt. Nur wenn die 2 jar Tasks sich auf verschiedener Klassen beziehen, können zwei jar-Dateien richtig erzeugt werden. Also das folgende Bild zeigt die Task Definition, die zur erfolgreicher Erzeugung der zwei jar-Dateien führt:

```
application {
    // package and class to be main method
    mainClassName = 'myproject.ProMain'
}

jar {
    manifest {
                // set location of main class
                attributes 'Main-Class': application.mainClassName
                // places with classes which may be needed by JVM
            attributes 'Class-Path': 'commons-lang3-3.9.jar'
    }
        exclude 'myproject/Main.class'
        exclude 'myproject/CreateAccount.class'
    baseName = 'ProfessionalVersion'
}

task jarv2(type: Jar) {
    manifest {
        attributes 'Main-Class': 'myproject.Main'
        attributes 'Class-Path': 'commons-lang3-3.9.jar'
    }
    from sourceSets.main.output
    exclude 'myproject/ProfessionalBankAccount.class'
    exclude 'myproject/CreditService.class'
    exclude 'myproject/CreateProAccount.class'
    exclude 'myproject/ProMain.class'
    baseName = 'StandardVersion'
}

jar.dependsOn(jarv2)
```

2. Programmierung der Geschäftslogik

Im Banksystem kann der Kunde verschiedene Befehle eingeben, um die Funktionen des Systems abzurufen. Dafür haben wir eine mainLoop Methode geschrieben, die eingegebene Befehl (z.B. „deposit 2 50“) überprüft und die entsprechende Methode („executeDeposit“ Methode) abruft. Weil einige Bankservices wurden zur Verfügung gestellt und im Laufe des Projekts wurden noch weitere Bankservices entwickelt, ist die mainLoop Methode eventuell sehr lang geschrieben. Außerdem bei der mainLoop Methode sind Input und Output zusammen verpackt. Zudem gibt es viele lokale Variable in einem Case innerhalb mainLoop. Dies macht es schwierig, die Code Refactor zu machen.

Beispielsweise wird die Befehl von Kunden „create“ durch die folgende Code-Snippet geprüft:

```
67              default: {
68                  if (accountName != null) {
69                      String[] inputArgs = choice.split(" ");
70                      int numberOfArgs = inputArgs.length;
71                      String command = inputArgs[0];
72
73                      if (numberOfArgs == 1 && command.equals("create")) {
74                          if (isProVersion) {
75                              CreateProAccount createProAccount = new CreateProAccount();
76                              int accountId = createProAccount.executeCreateProAccount(inputArgs[0], 0, accounts);
77                              if (accountId != -1) {
78                                  accountMap.putIfAbsent(accountName, new ArrayList<>());
79                                  accountMap.get(accountName).add(accountId);
80
81                                  standardOut
82                                          .println("New account is created. The account number is " + accountId + " .");
83                              }
84                          } else {
85                              CreateAccount createAccount = new CreateAccount();
86                              int accountId = createAccount.executeCreateAccount(inputArgs[0], 0, accounts);
87                              if (accountId != -1) {
88                                  accountMap.putIfAbsent(accountName, new ArrayList<>());
89                                  accountMap.get(accountName).add(accountId);
90
91                                  standardOut
92                                          .println("New account is created. The account number is " + accountId + " .");
93                              }
94                          }
```

Wenn der Befehl „create“ richtig eingegeben wird, wird die Banksystem Version überprüft. Je nach der Version (z.B. isProVersion = true) wird das entsprechende Kontotyp (z.B. CreateProAccount) erzeugt und die entsprechende Methode (z.B. executeCreateProAccount) wird abgerufen, um eine Bezeichner für das Konto zu erstellen. Danach wird das erzeugte Bankkonto in einem Hashmap abgebildet. Am Ende wird eine Nachricht den Kunde angezeigt (z.B. „New account is created...“). Eine solche If-Schleife besteht aus zu viele Zeile Code. Wenn die Methode viele solche If-Schleife beinhaltet, ist es zu lang und die Code ist zu kompliziert geworden. Es ist auch nicht verständnisvoll für die andere Entwickler.

3. Continuous Entwicklung des Systems

Wir haben während der Entwicklung des Banksystems erlebt, dass wir die Code immer neu schreiben oder umstrukturieren müssen, wenn wir neue Funktionalitäten für die Applikation entwickeln würden. Zur Entwicklung komplexerer Funktionalitäten benötigt es immer weitere und tiefere Programmiererfahrung und -erkenntnisse.
Beispielsweise haben wir eine Socket programmiert, um das Banksystem über einen TCP-Port bedienbar zu machen. Diese Aufgabe war für uns nicht einfach, denn wir neben der Bereitstellung einer Socket mussten auch überlegen, wie die Code über den TCP-Port eingebracht werden können. Eventuell haben wir dafür eine OutputStream erstellen, damit die String Datei über den TCP-Port anstatt in Konsole angezeigt werden:

```
18 public class BankSystem {
19
20     private InputStream standardIn;
21     private PrintStream standardOut;
22
23     public BankSystem(InputStream standardIn, PrintStream standardOut) {
24         this.standardIn = standardIn;
25         this.standardOut = standardOut;
26     }
27
```

Also ein Objekt der InputStream und ein Objekt der PrintStream wurden erste deklariert. Ein Konstruktor wurde neu hinzugefügt, die die obigen Objekte einnimmt. Danach mussten alle vorherige „system.out“ vom Objekt der PrintStream ersetzt werden.

```
47         while (choice != null) {
48             switch (choice) {
49             case "login": {
50                 if (accountName == null) {
51                     standardOut.println("Please enter your name");
52                     accountName = scanner.nextLine();
53                     standardOut.println("Hi, " + accountName + "!");
54                 } else {
55                     standardOut.println("you are logged in already");
56                 }
57                 break;
```

Eine neue Banksystem Objekt wird dann in BanksystemServer Klasse, wo die Verbindung mit der TCP-Port stattfindet, erzeugt.

```
 9 public class BankSystemServer {
10     public Socket clientSocket;
11     public ServerSocket serverSocket;
12     public PrintStream out;
13     public InputStream in;
14
15     public void start(int port) throws IOException {
16         serverSocket = new ServerSocket(port);
17         clientSocket = serverSocket.accept();
18         out = new PrintStream(clientSocket.getOutputStream(), true);
19         in = clientSocket.getInputStream();
20
21         BankSystem bankSystem = new BankSystem(in, out);
22         bankSystem.mainLoop();
23     }
```

4. Deployment der Applikation in Cloud

Es gelingt uns nicht, die Applikation in Google Cloud auszuliefern. Allerdings ist die Applikation in lokale Netzwerk bedienbar.

Aspekte die einfach waren

1. Erzeugung der Javadoc

Viele Tutorials für Javadoc sind im Internet vorhanden und die Javadoc Konvention ist einfach zu verstehen und folgen. Beispielsweise muss man einfach eine Beschreibung, alle Parameter und die Rückgabewert für die executeTransfer Methode geben:

```
153    /**
154     * Executes the command to transfer money from a bank account to another bank
155     * account.
156     *
157     * @return a double representing the balance of the sender's account.
158     * @param input                    A string representing the input command
159     * @param numberOfArg              An integer representing the number of
160     *                                 arguments taken for command validation
161     * @param senderAccountId          An integer representing the identifier of
162     *                                 the sender's bank account
163     * @param receiverAccountId        An integer representing the identifier of
164     *                                 the receiver's bank account
165     * @param amount                   A double representing the amount of money to
166     *                                 withdraw from a bank account
167     * @param existedSenderAccountIds  An array list that store all existing bank
168     *                                 IDs of the sender
169     * @param existedReceiverAccountIds An array list that store all existing bank
170     *                                 IDs of the receiver
171     * @param list                     An array list that store the existing bank
172     *                                 accounts in the bank system
173     * @param standardOut              A PrintStream object that print representations of string data.
174     */
175    public double executeTransfer(String input, int numberOfArg, Integer senderAccountId, Integer receiverAccountId,
176            double amount, List<Integer> existedSenderAccountIds, List<Integer> existedReceiverAccountIds,
177            List<BaseAccount> list, PrintStream standardOut) {
178        if (!isValidCommand(input, numberOfArg, TRANSFER_COMMAND, 5)) {
179            standardOut.println("Invalid command: " + input);
180            return -1;
181        }
```

Fazit

Zusammenfassend haben wir ein funktionierende Banksystem mit Continuous Integration Pipeline entwickelt. Zwei Produkte nämlich die Standard-Version und die Professional-Version wurden bereitgestellt. Das zusätzliches Modul für das Banksystem ermöglicht, das Kredite an einzelne Konten zu vergeben und auf Befehl Zinsen einzuziehen. Das Projekt ist auf einem Download-Server zur Verfügung gestellt und es ist bedienbar über einen TCP-Port. Wir haben entsprechende Tests für die Klassen des Banksystems entwickelt. Außerdem haben wir verschiedene populäre Technologie und Werkzeuge z.B. Gradle das Versionierungssystem Git und Github, Jenkins, Google Cloud geübt. Wir haben gelernt, wie verschiedene Systeme (Gradle, Git, jUnit usw) miteinander verbinden und wie ein Build-Prozess einer Software auf automatisierte Weise ausgeliefert werden kann.