Aufgabe - CI CD-Pipeline für Java Projekt mit GitHub Actions

Lernziele

- Aufsetzen einer Maven-basierten Java-Projektstruktur
- Implementierung von automatisierten Tests
- Einrichtung verschiedener Code-Qualitätswerkzeuge
- Konfiguration einer GitHub Actions / Jenkins Pipeline

Step by step-Beschreibung 👟

1. Projekt-Setup

- Erstelle ein neues Java Maven-Projekt in IntelliJ
- Implementiere eine IBAN-Prüfer Klasse als Hauptfunktionalität
- Konfiguriere die `pom.xml` für die JAR-Generierung

```
import java.util.HashMap;
import java.util.Map;
import java.util.ArrayList;
import java.util.List;

public class IBANChecker {
    private static final Map<String, Integer> chars = new HashMap<>();

    static {
        chars.put("AT", 20);
        chars.put("BE", 16);
        chars.put("CZ", 24);
        chars.put("DE", 22);
        chars.put("DK", 18);
        chars.put("FR", 27);
    }

    public static void main(String[] args) {
        String iban = "DE227902007600279131";
        System.out.println("Welcome to the IBAN Checker!");
        System.out.println("IBAN " + iban + " is " + validate(iban));
    }
```

```
    public static boolean validate(String iban) {
        if (!checkLength(iban)) {
            return false;
        }

        String rearrangedIban = rearrangeIban(iban);
        String convertedIban = convertToInteger(rearrangedIban);
        List<String> segments = createSegments(convertedIban);

        return calculate(segments) == 1;
    }

    private static int calculate(List<String> segments) {
        long n = 0;

        for (String segment : segments) {
            if (segment.length() == 9) {
                n = Long.parseLong(segment) % 97;
            } else {
                segment = n + segment;
                n = Long.parseLong(segment) % 97;
            }
        }

        return (int) n;
    }

    private static boolean checkLength(String iban) {
        String countryCode = iban.substring(0, 2);
        return chars.containsKey(countryCode) && chars.get(countryCode) ==
iban.length();
    }

    private static String convertToInteger(String iban) {
        StringBuilder convertedIban = new StringBuilder();
        String upperIban = iban.toUpperCase();

        for (char c : upperIban.toCharArray()) {
            if (Character.isDigit(c)) {
                convertedIban.append(c);
            }
            if (Character.isLetter(c)) {
                convertedIban.append(c - 55);
            }
        }

        return convertedIban.toString();
    }

    private static List<String> createSegments(String iban) {
```

```
        List<String> segments = new ArrayList<>();
        String remainingIban = iban;

        segments.add(remainingIban.substring(0, 9));
        remainingIban = remainingIban.substring(9);

        while (remainingIban.length() >= 9) {
            segments.add(remainingIban.substring(0, 7));
            remainingIban = remainingIban.substring(7);
        }

        segments.add(remainingIban);
        return segments;
    }

    private static String rearrangeIban(String iban) {
        return iban.substring(4) + iban.substring(0, 4);
    }
}
```

2. Test-Implementation 🔪

- Füge JUnit 5 als Testabhängigkeit hinzu
- Implementiere Unit-Tests für den IBAN-Prüfer:
  - valide IBAN ("DE22790200760027913168")
  - nicht-valide IBAN ("DE21790200760027913173")
  - falsche Länge ("DE227902007600279131")
  - unbekannter Ländercode ("XX22790200760027913168")
- Konfiguriere Maven für die Ausführung der Tests

3. Code-Coverage mit JaCoCo ✅

- Integriere das JaCoCo-Maven-Plugin
- Konfiguriere die Coverage-Berichterstellung
- Führe die Tests aus und überprüfe die Coverage-Berichte

4. Javadoc-Generierung 📃

- Füge das Maven-Javadoc-Plugin hinzu
- Dokumentiere den Code mit Javadoc-Kommentaren
- Generiere die API-Dokumentation

5. Code-Qualitätsanalyse ("Linter") 👀

- Integriere SpotBugs als statisches Analyse-Tool
- Konfiguriere SpotBugs-Regeln

- Führe die Analyse aus und behebe gefundene Probleme

---

6. CI/CD-Pipeline aufbauen👷

Variante 1: Github Actions

- Erstelle einen Github-Actions-Workflow für kontinuierliche Integration
- Konfiguriere die Build-Umgebung mit JDK 20
- Automatisiere den Build-Prozess und die Test-Ausführung
- Erweitere den Workflow um Artifact-Uploads
- Konfiguriere das Speichern von:
  - JAR-Datei
  - Javadoc
  - JaCoCo-Report
  - SpotBugs-Report

Bonus-Task ✨

- Richte GitHub Pages für die Javadoc-Dokumentation ein
- Implementiere Branch-Protection-Rules
- Füge Badge für den Build-Status hinzu

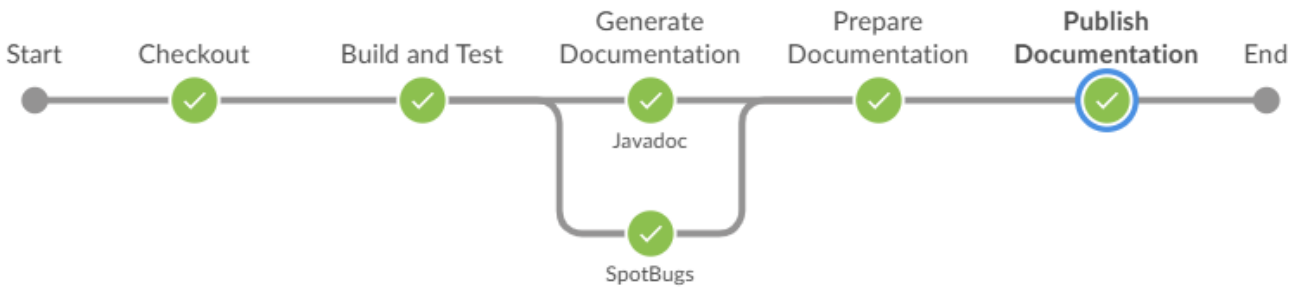
Variante 2: Jenkins

- Erstellung einer Pipeline mit einer `Jenkinsfile`
- Nutze dazu den Pipeline-Syntax Generator: http://localhost:8080/pipeline-syntax/
- Hilfreiche Steps dabei:
  - sh: Ausführen von Shell-Befehlen (bspw. mvn)
  - archiveArtifacts: Archivieren von Artefakten mittels "Glob-Pattern" (bspw. den Ordner mit den JavaDocs / JaCoCo Coverage)
  - junit: Archivieren von JUnit XML-Testergebnissen
  - publishHTML: Publizieren von HTML-Reports (Alternativ zu archiveArtifacts, bspw. für JavaDoc / JaCoCo)
  - script: Schreiben von Groovy-Code (java-DSL)
- Implementiere folgende Stages:
  - Code Checkout (aus dem Github-Repo)
  - Build und Test des Java-Projekts (`mvn clean verify`)
  - Generierung der Dokumentation
    - JavaDoc
    - SpotBugs-Analyse

- Erstellung der Dokumentationsverzeichnisstruktur
- Generierung einer HTML-Indexseite mit Links zu den einzelnen Dokumenten

Bonus-Task ✨

- Veröffentlichung der Dokumentation mittels Jenkins HTML Publisher 👀



Erwartete Ergebnisse 🤓

- Funktionierender IBAN-Validator
- Vollständige Testsuite
- Automatisierte CI/CD-Pipeline (mit Github Actions oder Jenkins)
- Dokumentation und Qualitätsberichte
- Reproduzierbarer Build-Prozess