Dev Dokumentation

Die Dev Dokumentation umfasst die Informationen bezüglich der Entwicklungsumgebung, Entwicklungskonfiguration und Arbeitsweise.

Coding Conventions

Coding Conventions beziehen sich auf eine Reihe von Regeln, Richtlinien und Vereinbarungen, die von Entwicklern*innen verwendet werden, um den Stil und die Struktur von Quellcode in Softwareprojekten zu standardisieren. Diese Konventionen dienen dazu, den Code lesbarer, wartbarer und konsistenter zu machen, indem sie einheitliche Namensgebung, Einrückung, Kommentierung und andere Aspekte der Programmierung festlegen.

Coding Conventions sind nicht nur für die individuelle Lesbarkeit des Codes wichtig, sondern auch für die Zusammenarbeit in Entwicklerteams. Durch die Einhaltung gemeinsamer Konventionen wird die Codebasis für alle Entwickler*innen verständlicher und erleichtert die Wartung, Erweiterung und Fehlerbehebung im gesamten Projekt.

Codestyle mit CheckStyle

Checkstyle ist ein statisches Code-Analysetool, das verwendet wird, um sicherzustellen, dass der Quellcode bestimmten Coding-Standards und -Konventionen entspricht. Es überprüft den Code auf stilistische Fehler, Formatierungsregeln, Namensgebung und andere Aspekte, um eine konsistente Code-Qualität sicherzustellen. Wir haben die Google CheckStyle aals unsere Basis genommen. Denn dieser CodeStyle bietet eine gute Leserlichkeit und Abweichungen werden nicht direkt als Error sondern als Warning markiert.

Naming Convention für Branches und co

Die Naming Convention für Branches und Commits zeigen auf wie diese benannt werden sollen. Dadurch wird eine Norm für die Namensgebung gewährleistet und man findet sich schnell zurecht.

Table 1. Naming Conventions

Element	Pattern	Beschreibung
Branch	SXXXXX-[Branchname]	Nummer=Story, kurzer Name
Commit	SXXXXX-[Commitmessage]	Commitmessage = Nomen + verb

Clean Code

Der Code soll, wenn möglich, mit dem Clean-Code-Prinzip geschrieben werden. So bezieht sich Clean Code auf die Praxis, Quellcode auf eine klare, verständliche und wartbare Weise zu schreiben. Es legt Wert auf Lesbarkeit, Einfachheit und gut strukturierten Code. Clean Code verwendet sinnvolle Namen für Variablen, Funktionen und Klassen, reduziert komplexe Logik und

minimiert wiederholten Code. Es fördert auch die Verwendung von Kommentaren zur Dokumentation und die regelmäßige Durchführung von Refactorings, um den Code kontinuierlich zu verbessern und wartungsfreundlicher zu gestalten.

Sicherstellung der Einhaltung der Konventionen

Wie	Begründung
Code Review	Code-Reviews ermöglichen eine effiziente Verteilung von Wissen und Know-how innerhalb eines Entwicklungsteams. Fehler verursachen weniger Aufwand, wenn sie bereits bei Code-Reviews korrigiert werden. Je später ein Fehler im Code gefunden wird, desto höher ist der Aufwand, ihn zu beheben. Es lohnt sich, Code-Reviews als Massnahme zur Fehlerfrüherkennung zu etablieren. Wir haben uns dazu entschieden, am Schluss ein grosses Code-Review zu machen, da uns die Zeit während des Semesters gefehlt hat um sie regelmässig durchzuführen.
CodeStyle als Intellij Formatierungsstandard	Die Hinterlegung der CheckStyle Definiton als Code-Formatierungsstandard für die Erstellung des Codes machen es für andere Teammitglieder*in einfach, ihn zu verstehen. In einem Jahr oder fünf Jahren ist es einfacher, den Code zu verstehen, ohne jeden einzelnen Satz lesen zu müssen. Auch wird dadurch direkt den CodeStyle eingehalten, sobald der/die Entwickler*in eine Codeformatierung ausführt.
CodeStyle mit dem Checkstyle Plugin	Das Checkstyle Plugin gewährleistet, dass beim Kompilieren überprüft wird ob der CodeStyle eingehalten wurde. Des Weiteren kann der Checkstyle auch manuell ausgeführt werden, um Probleme frühzeitig zu erkennen. Checkstyle ist ein statisches Code-Analysetool, das verwendet wird, um die Einhaltung von Coding-Standards und -Konventionen im Quellcode zu überprüfen.

Wie	Begründung
Sonarcloud	Sonarcloud prüft. ob der Code den gängigen Clean-Code Prinzipien entspricht. SonarCloud ist ein cloudbasiertes statisches Code-Analysetool, das Entwicklern dabei hilft, die Qualität ihres Codes zu verbessern. Es überprüft den Quellcode auf verschiedene Aspekte wie Code-Stil, Code-Duplizierung, Bugs, Sicherheitslücken und Designmängel. Mit SonarCloud können Entwickler potenzielle Probleme frühzeitig erkennen und beheben, was zu robusterem und zuverlässigerem Code führt. Es bietet auch detaillierte Berichte und Metriken, um einen umfassenden Überblick über den Zustand des Codes und den Fortschritt bei der Code-Qualität zu erhalten. Warnungen der Stufe rot werden sofort behoben und Warnungen der Stufe grün und gelb behandeln wir am Ende.

Setup Dev Environment

Für das Aufsetzen einer Entwicklungsumgebung wird hier die entsprechende Anleitung geliefert.

Java installieren

Damit man bequem sich Java installieren kann, wird empfohlen sich SDK man herunterzuladen.

- SDKMan herunterladen und installieren Siehe skdman Abschnitt install
- Mit "sdk list java" die neuste Java 17.x.x-tem Version raussuchen
- Mit "sdk install java 17.x.x-tem" diese Version installieren
- Mit "sdk list maven* die neuste maven Version raussuchen
- Mit "sdk install maven x.x.x" diese Version installieren

Projekt konfigurieren

Dieser Abschnitt erläutert das Aufsetzen des Projektes.

- Git Repository klonen (Tipp: Am besten mit ssh, dann muss man jeweils nur den SSH key eingeben und nicht Email plus Passwort wie bei https)
- Repository mit Intellij öffnen
- Prüfen, ob die richtige Javaversion "17.x.x-tem" gesetzt ist über "File → Project Structure". Wenn nicht dann muss diese angepasst werden!
- Das Projekt kompilieren Profile "Run Local" oder die Main Methode als Startpunkt setzen.

Plugins installieren und konfigurieren

Hier wird auf die benötigten Plugins und deren Konfiguration verwiesen.

- Folgende Plugins installieren: SonarLint, AsciiDoc und CheckStyle-IDEA über "IntelliJ → Settings → Plugins"
- SonarLint mit SonarQube verbinden, damit die richtige Konfiguration als Standard definiert ist über "Settings → Tools → SonarLint"
  - Neue Connection hinzufügen und Connection Type "Sonarcloud" auswählen
  - Token eingeben (Ist auf Gitlab unter "Settings → CI/CL Settings → Variables → SONAR_TOKEN" zu finden)
  - Blackout auswählen
  - Checkbox aktiviert behalten
  - Im SonarLint Plugin das Settings Symbol anklicken
  - "Bind to SonarQube" auswählen die zuvor erstellte Connection auswählen
  - Projektkey ist "blackout_blackoutapplication" und bestätigen
- CheckStyle aufklappen und bei den Rules "Blackout" auswählen (Mehr Informationen zu CheckStyle])

Arbeitsweise

Die Arbeitsweise erläutert, die anvisierte Vorgehensweise beim Umsetzen von Features. Dadurch kann der Einstieg für Anfänger einfacher gewährleistet werden, Streitigkeiten anhand eines Standards gelöst werden und für eine gute Struktur des Projektes hingearbeitet werden

Starten

- Branch erstellen für das Feature (Naming Convention berücksichtigen und den Branch "Main" als Basis für den Branch verwendet)
  - Für Anfänger am besten im Gitlab
  - Für Fortgeschrittene muss der locale Main mit pull aktualisiert werden, sonst ist er schon veraltet bevor man beginnt zu arbeiten
- Branch pullen oder für wenn er lokal erstellt worden ist pushen.

Umsetzen

- Immer regelmässig granulare Commits erstellen (Einzelne Aspekte commiten und nicht tausend Änderungen)

Abschliessen

- Auf Main wechseln und diesen mit einem Pull aktualisieren "git switch main, git pull main"
- Auf seinen Feature Branch wechseln und den Main reinmergen "git switch [Branchname], git

merge main"

- Evtl. Mergekonflikt lösen, bei Unklarheiten lieber die Person fragen, welche die Änderung gemacht hat, die den Mergekonflikt verursacht. Damit können Fehler schon vorgebeugt werden, bei unvorsichtigen lösen von Mergekonflikten.
- Branch pushen
- Merge-Request erstellen
- Code-Review durchführen und Branch mergen

Sonarcloud

Sonarcloud ist ein Tool für das Prüfen der Codequalität, mit sogenanten Qualitätsrichtlinien. Die Qualitätsrichtlinien sind von Sonar standardmässig definiert und entsprechen den allgemeinen Best Practise bezüglich Clean-Code, vermeiden von Bugs und Codestruktur. Natürlich können die Regeln angepasst werden, jedoch belassen wir sei bei den Best Practise. Sonarcloud kann gratis benutzt werden, solange das Projekt öffentlich ist. Unser Projekt unterstützt die Analyse des Main Branches (longterm branch) und den Featurebranches (shortterm branches). Longterm branches repräsentieren branches, welche lange bestehen aka main branches. Hier werden alle Probleme, welche in der Analyse gefunden werden, hervorgehoben, der longterm branch wird nich mit anderen verglichen. Shorterm branches sind nur kurze Branches die zu einem kurzen Zeitpunkt bestehen, diese werden mit dem definierten longterm branch verglichen und so eingeführte Probleme hervorgehoben. Pull request werden nicht untersützt. Anbei ist der Link zu unserem Sonarcloud project Link zu Sonarcloud.

Pipeline

Wir verwenden eine Pipeline, um unseren Build-Process zu automatisieren. Dies bietet uns die Möglichkeit, dass wir nach jedem push eine Qualitätskontrolle haben und uns manuelle Arbeit sparen. Des Weiteren können wir so garantieren, dass die Qualitätskontrolle garantiert durchgeführt werden und nicht einfach manuell von einem Entwickler übersteuert werden kann. Pipelines sind eine Anleitung, was für Kommandos und in welcher Reihenfolge ausgeführt werden. Sobald ein Schritt fehlschlägt, bricht die ganze Pipeline ab und wird auf failed gesetzt. Unsere Schritte sind voneinander abhängig, da jeder vorherige Schritt zuerst ausgeführt werden muss, bevor der nächste in Anspruch genommen werden kann. Daher können wir keine Schritte parallelisieren. Wir haben unsere Pipeline folgendermassen aufgebaut:

(codestyle-check) → (unit-test) → (sonarcloud)

codestyle-check

Zuerst werden die codestyle-checks mit dem checkstyle plugin durcheführt. Checkstyle ist ein Stylingdokument, welches verwendet wird um zu prüfen, ob der Code einem definierten Codestyle entspricht. Wir haben den Google-Standart als basis genommen (Siehe Coding Convention]).

unit-test

Danach muss die Applikation im Schritt unit-test gebaut werden, damit dann die unit-test durchgeführt werden können. Die unit-test testen einzelne Einheiten der Applikation und bieten so eine automatisierte Testingvariante, um Regressionen zu erkennen.

sonarcloud

Sobald die Applikation erfolgreich getestet wurde, wird der Schritt sonarcloud ausgeführt. In diesem Schritt wird mit dem Sonarcloud Plugin die Analyse auf Sonarcloud angestossen. Wir haben für unser Projekt dort eine Unterscheidung in Longterm und Shortterm branches eingebaut. Link zu Sonarcloud

Pipeline-Datei

Die Pipeline-Datei ist auf dem GitLab vorhanden.

.gitlab-ci.yml

```
# Neustes Mavenversion auf Java 17
image: maven:3.9.0-amazoncorretto-17

# Stages welche ausgeführt werden
stages:
  - code_style_check
  - unit_test
  - code_quality_analysis

# SONAR_TOKEN soll vertraulich behandelt werden (Es ist wie ein Passwort)
# Daher ist es eine protected variable, welche maskiert ist (Settings -> CI/CD
Settings -> Variables "Ausklappen").
# Das heisst man muss das Token nie direkt reinschreiben, sonder kann die variable
verwenden, diese Variable wird beim build ersetzt.
#  Weil sie auch maskiert ist, wir das Token nie im Log anzeigt, das trägt auch zur
sicherheit bei.
variables:
  MAVEN_CLI_OPTS: >-
    -Dsonar.login=$SONAR_TOKEN
    -Dsonar.host.url=https://sonarcloud.io
    -Dsonar.organization=blackout
    -Dsonar.projectKey=blackout_blackoutapplication
  MAVEN_CLI_OPTS_FEATURE_BRANCH: >-
    -Dsonar.branch.name=$CI_COMMIT_BRANCH
    -Dsonar.branch.target=main

#Stage "code_style_check"
code_style_check:
    stage: code_style_check
    script:
      - mvn checkstyle:check
```

```
#Stage "unit_test"
unit_test:
    stage: unit_test
    script:
      - mvn test

#Stages "code_analysis"
code_quality_analysis_on_feature_branch:
    stage: code_quality_analysis
    except:
      - main
    script:
      - 'mvn verify sonar:sonar $MAVEN_CLI_OPTS $MAVEN_CLI_OPTS_FEATURE_BRANCH'

code_quality_analysis_on_main:
    stage: code_quality_analysis
    only:
      - main
    script:
      - 'mvn verify sonar:sonar $MAVEN_CLI_OPTS'
```

Unittest Templates

Die Templates dient für das einfache Erstellen von Unittests.

Unittest.java Template

```
#if (${PACKAGE_NAME} && ${PACKAGE_NAME} != "")package ${PACKAGE_NAME};#end

import org.junit.jupiter.api.Test;

#parse("File Header.java")
public class ${NAME} {
}
```

Manuelle Tests Template

Die Templates dient für das einfache Erstellen von manuellen Tests.

Testcase.adoc Template

```
=== ${TEST_NAME}

[%autowidth]
|===
|*Beschreibung*
|
|*Vorbedingung*
```

```
|
|*Testdaten*
|
|*Schritte*
|
|*Ergebnis*
|
|===
```

Testrun.adoc Template

```
== Testdurchlauf vom Test: ${TEST_NAME}

[%autowidth]
|===
|*Datum* | ${DAY}.${MONTH}.${YEAR}
|*Verantwortlich* | Andri Pieren
|===

/include::../04_testing/00_test_case/${FILE_NAME}.adoc[Testbeschreibung]

// TODO Status anpassen
=== Status -> Test has /

Die Bestimmung des Bestanden / Nicht Bestanden-Status hängt davon ab, wie das
erwartete Ergebnis und +
das tatsächliche Ergebnis miteinander verglichen werden.

*Gleiches Ergebnis* = Bestanden/Pass ->
*Unterschiedliche Ergebnisse* = Fehlschlagen/Fail ->

=== Eventuelle Bemerkungen
```