Waldbrandpräventions-Applikation - KIWA APP

Projektdokumentation – Gruppe 42

Philipp Brodersen philipp.brodersen@stud.tu-darmstadt.de
Noel Goldschmidt noel.goldschmidt@stud.tu-darmstadt.de
Johanna Herbst johanna.herbst@stud.tu-darmstadt.de
Yannic Hochheimer yannic.hochheimer@stud.tu-darmstadt.de
Adrian Schubek adrian.schubek@stud.tu-darmstadt.de

Teamleitung:
Mika Pomper mika.pomper@stud.tu-darmstadt.de

Auftrag:
Tobias Heuser tobias.heuser@umi.city
[ui!] Urban Software Institute
31. März 2023

TECHNISCHE UNIVERSITÄT DARMSTADT

Bachelorpraktikum
Wintersemester 2022/23
Fachbereich Informatik

Inhaltsverzeichnis

1 Projektbeschreibung

Motivation und Zielsetzung: Das Projekt KIWA (kurz für: **KI**-basierte **Wa**ldbrandprävention) hat das Ziel, mithilfe von maschinellem Lernen aus Bildinformationen, die durch das Überfliegen von Waldgebieten gesammelt werden, mögliche Brandherde frühzeitig zu erkennen. Dies kann dazu beitragen, die Ausbreitung von Waldbränden zu verhindern und damit auch Schäden an der Umwelt und an der Bevölkerung zu minimieren. Bisher fliegen in Gebieten mit hoher Waldbrandwarnstufe meist Luftbeobachter in Kleinflugzeugen Patrouille. Wird Rauch oder Feuer gesichtet, folgt die Benachrichtigung der Koordinationsstelle und ggf. die Einleitung weiterer Maßnahmen wie z.B. der Brandbekämpfung. Allerdings fehlt es bislang an digitalen Werkzeugen, die die relevanten Datenquellen zur Waldbrandprävention vereinen, aufbereiten und für die Einsatzkräfte leicht verständlich zur Verfügung stellen.

Im Rahmen des Bachelorpraktikums konzentrieren wir uns also auf die Entwicklung einer solchen webbasierten Fachanwendung. Die Zielgruppe stellen in erster Linie Feuerwehren dar. So soll die Anwendung es den Feuerwehrleuten erleichtern, die Waldbrandgefahr in einem bestimmten Gebiet einzusehen und zu bewerten, indem die aktuellsten und relevantesten Daten direkt in KIWA einsehbar sind.
Die Anwendung ermöglicht es den Nutzer*innen, alle relevanten Datenquellen zur Waldbrandpräventation an einem Ort gesammelt anzuzeigen (vgl. Abb. 1.1). Dazu zählen beispielsweise digitale Karten des Überwachungsgebiets, die gerade vorherrschende Brandgefahrenstufe innerhalb eines Gebietes, die Position und erfasste Daten der Drohnen, die die Waldgebiete überfliegen, sowie die Anzeige von Wettervorhersagen für die folgenden Tage. KIWA bietet dann ebenfalls die Möglichkeit diese Daten zu verwalten, die erkannten Brandherde einfach und intuitiv zu visualisieren (vgl. Abb. 1.2), sowie ggf. die zuständigen Einsatzkräfte direkt aus der Anwendung heraus zu alarmieren.

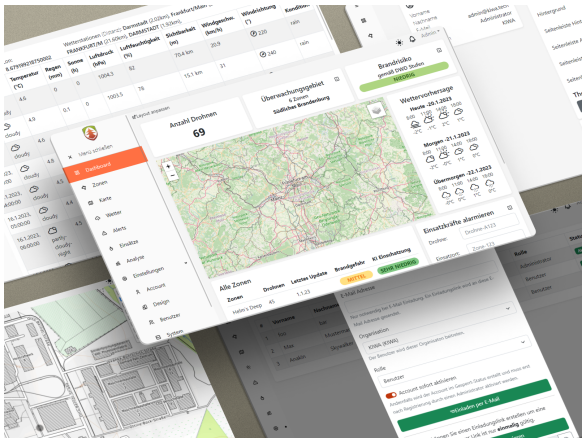

Abbildung 1.1: Dashboard der Anwendung

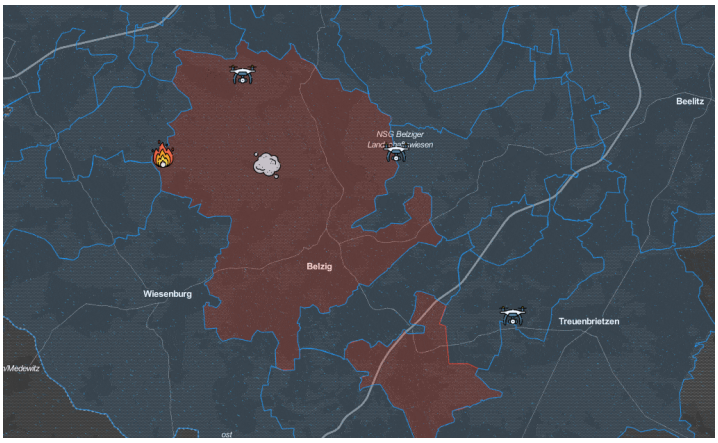

Abbildung 1.2: intuitive Visualisierung der erkannten Brandherde

Des Weiteren soll KIWA auch als eine zentrale Infrastruktur für den Datenaustausch, die Anzeige von Drohnenbildern, KI-Vorhersagen und Daten von Geoinformationssystemen (GIS-Daten), wie sie bspw. bei den von uns verwendeten Kartenansichten zum Einsatz kommen, dienen. Dabei werden die Eingaben (bestehend aus Bilddaten, die durch Drohnen erfasst werden) durch ein KI-System (mit Bezug zur Identifizierung von Feuer und Rauch) ausgewertet. Gängige Ausgaben sind visuelle Darstellungen von möglichen Brandherden, die Wahrscheinlichkeit für Feuer und/oder Rauch in diesen Gebieten und eine dementsprechende Einstufung der Brandgefahr, sowie die Anzeige der Konfidenz der KI-Vorhersage. Das bereits trainierte KI-Modell inklusive Beispieldaten (wie Drohnendaten, Bilddaten etc.) wurden uns dabei von unserem Auftraggeber ([ui!]) zur Verfügung gestellt. Das selbstständige Trainieren von Machine-Learning Modellen war somit nicht notwendig.

In unserem Projekt konzentrieren wir uns also einzig und allein auf die Entwicklung der Fachanwendung. Dies bedeutet, dass sowohl das komplette Frontend, als auch das Backend von unserer Gruppe während des Bachelorpraktikums von Grund auf neu entwickelt und implementiert wird. Konkrete Details hierzu sind im Abschnitt zu den "Verwendeten Technologien"nachzulesen. Die Webanwendung soll zudem auf mehreren Plattformen und verschiedensten Endgeräten (wie Smartphone, PC, Tablet) zum Einsatz kommen, weswegen wir stark auf die Portabilität der Anwendung, und damit einhergehend, auf die Funktionalität und korrekte Darstellung von Inhalten geachtet haben. Wie bereits erwähnt, ist uns ebenso die Erhaltung der Benutzerfreundlichkeit ein wichtiges Anliegen. So soll die Anwendung auch nicht-technikaffinen Nutzer*innen eine leichte, intuitive Benutzererfahrung bieten. Die Drohnen und angezeigten Daten, sowie die Einschätzung der KI werden dazu durch aussagekräftige Bilder unterstützt. Auf diese Weise lässt sich beispielsweise der momentane Standort der Drohnen in Echtzeit nachverfolgen und überprüfen. Die angezeigte Vorhersage der KI ist somit für die/den Endbenutzer*in leichter nachvollziehbar, da genau zu sehen ist, wie die KI zu ihrem „Ergebnis“, also zu ihrer Vorhersage, gekommen ist. Durch diese starke Verbildlichung soll im weiteren Verlauf des Projektes das Vertrauen in das KI-Modell gesteigert werden.

Umsetzung und verwendete Technologien: Während der Softwareentwicklung haben wir einen agilen Designzyklus verfolgt, bei dem wir mit der Konzeption der Anwendung begonnen haben, diese anschließend implementiert und Rückmeldungen zur Nutzungsweise der Software eingeholt haben.
Ein weiterer Bestandteil des Praktikums ist zudem die Anforderungserhebung mit Stakeholder*innen. Hierbei geht es darum, die Bedürfnisse und Anforderungen der Nutzer*innen und Interessensgruppen, die von der Anwendung betroffen sind, zu erfassen und in die Entwicklung der Anwendung mit einfließen zu lassen, weswegen sich die Verwendung eines agilen Softwarentwicklungsprozesses, in dem sich zu Beginn gestellte

Anforderungen im Verlauf des Projekts auch verändern können, noch einmal mehr anbietet.
Insbesondere haben wir hier Feedback von der Feuerwehrschule Würzburg erhalten. Aufgrund dieses Feedbacks haben wir Anpassungen vorgenommen und die Anwendung weiter verbessert, um sicherzustellen, dass diese den Anforderungen und Bedürfnissen aller Nutzer*innen gerecht wird und die Nutzung nicht durch das Vorhandensein von unnötigen Funktionalitäten erschwert wird.

Für die technische Umsetzung der Anwendung nutzen wir im Frontend die Programmiersprache React[1] mit Typescript[2] (ermöglicht bspw. die Verwendung von Methodensignaturen und generischer Programmierung) und dem Framework Bootstrap (CSS)[3], um z. B. auf Bootstrap-Funktionalitäten wie Formulare und ähnliche vordefinierte Komponenten zurückgreifen zu können, und diese nicht von Grund auf implementieren zu müssen. Im Backend kommt die Programmiersprache Python[4] mit Fast-API[5] als Framework zum Einsatz, sodass z. B. externe Dienste wie Kartenansichten von OpenStreetMap oder Wettervorhersagen des Deutschen Wetterdienstes(DWD) leicht eingebunden werden können. Als Datenbankplattform verwenden wir SQLite[6], um Benutzerinformationen und Login-Daten in verschlüsselter Form zu speichern. Zudem nutzen wir nginx[7] („engine-x“) als Reverse Proxy, um Frontend und Backend miteinander zu verbinden. Alle Komponenten der Anwendung sind containerisiert, dazu verwenden wir die Virtualisierungssoftware Docker[8]. Dabei werden die Images für Front- und Backend automatisch mit der jeweils aktuellen Version des Front- bzw. Backend Repositorys gebaut und auf Docker Hub[9] hochgeladen. Zur Versionsverwaltung setzen wir GitHub[10] ein.

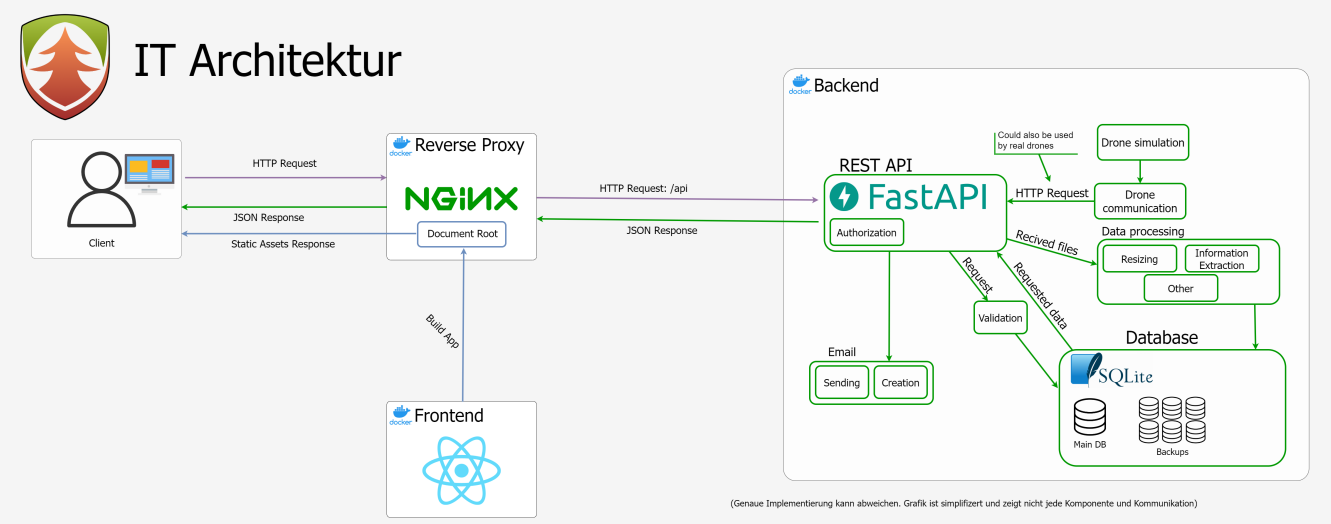

Abbildung 1.3: Darstellung der IT-Architektur

In Abbildung 1.3 werden die einzelnen Komponenten und ihre Kommunikation gezeigt. Im Backend können

[1] https://reactjs.org/
[2] https://www.typescriptlang.org/
[3] https://react-bootstrap.github.io/
[4] https://www.python.org/
[5] https://fastapi.tiangolo.com/
[6] https://www.sqlite.org/index.html
[7] https://nginx.org/en/
[8] https://www.docker.com/
[9] https://hub.docker.com/u/waldbrandpraevention
[10] https://github.com/

Drohnen über eingerichtete API Endpunkte ihre Updates (Standort und Timestamp) und Events (Detektion von Feuer und/oder Rauch) per POST-Request senden. Die empfangenen Daten werden verarbeitet und in der Datenbank gespeichert. Anschließend können sie vom Frontend, welches standardmäßig alle 10 Sekunden eine GET-Anfrage für die aktuellen Zonendaten an die API schickt, angezeigt werden. Der Intervall von 10 Sekunden ist angepasst auf die voraussichtliche Updatefrequenz, der in zukünftig eingesetzten Drohnen.

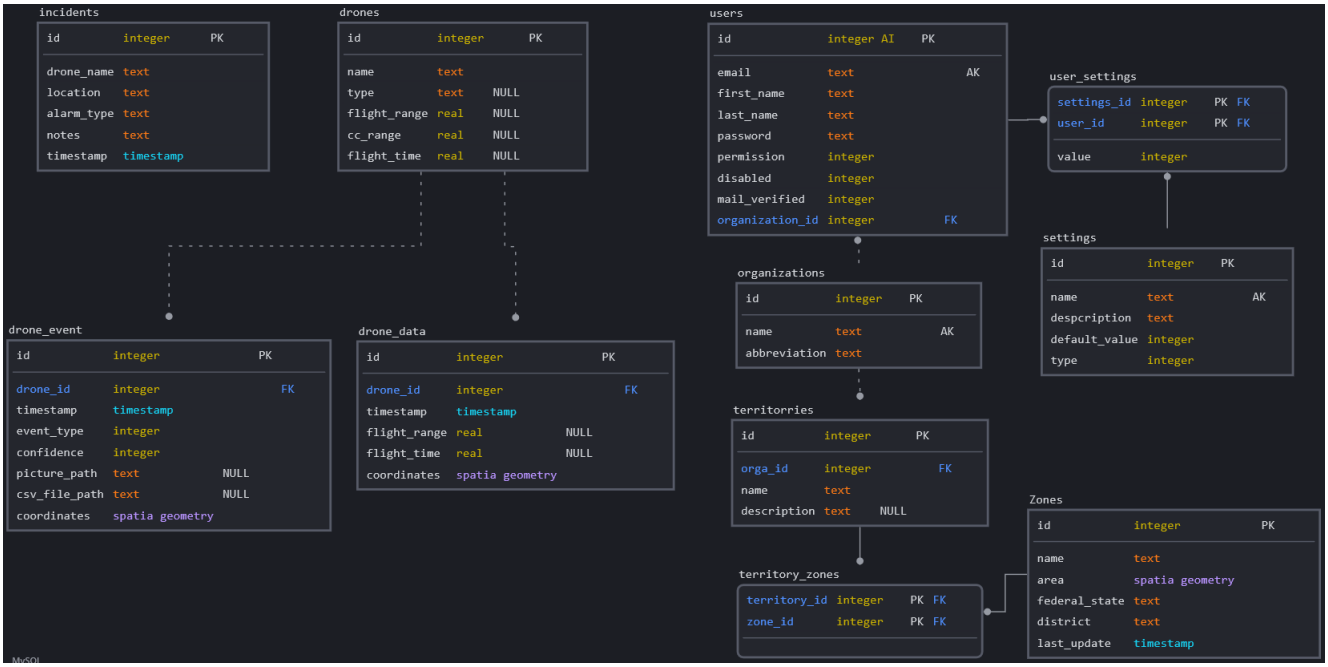

Abbildung 1.4: Darstellung der Datenbank

In Abbildung 1.4 wird das Datenbankschema gezeigt. Für SQLite ungewöhnlich sind die Attribute vom Typ spatia geometry[11], die das speichern von Geodaten, wie beispielsweise eine lat/lon Koordinate oder auch Polygone (z.B. die Grenze einer Gemeinde), ermöglichen. So können Zonenpolygone generiert, kombiniert und auf einer Karte dargestellt werden.

Nutzer und Orgadaten: In der Tabelle 'users' können Nutzer*innen angelegt werden. Dabei muss die id einer Organisation angegeben werden, denn so wird später sichergestellt, dass Nutzer*innen nur auf die Daten im Territorium ihrer Organisation zugreifen können.
Über die Tabelle 'user_settings' und 'settings' können neue Einstellungen mit default_value definiert und für Nutzer*innen gesetzt werden. Für jede Einstellung wird der Typ ihres Wertes festgelegt, mögliche Typen sind Integer, String oder JSON.
In 'zones' werden die einzelnen Zonen definiert, welche den Territorien von Organisationen zugeordnet werden können. Wir haben uns dafür entschieden, die bestehenden Gemeindegrenzen als Zonen zu definieren und können sie aus einer geojson-Datei in die Datenbank einlesen. Diese Definition ist intuitiv und hilft der Feuerwehr bei der internen Kommunikation über potentielle Brandstellen und deren Standort.
Über 'territories' und 'territory_zones' können diese Zonen, Territorien zugeordnet werden, welche widerum fest zu einer Organisation gehören. So können größere Gebiete aus vielen Zonen angelegt und einer Organisation zugeordnet werden.

[11] http://www.gaia-gis.it/gaia-sins/spatialite-cookbook/html/wkt-wkb.html

Drohnendaten: In der Tabelle ’drones’ können Drohnen angelegt werden. Dies passiert beispielsweise im Anmeldeprozess einer Drohne über den /drones/signup/ API Endpunkt. Hinzu kommen ’drone_data’ für Updates und ’drone_event’ für Events. Ein Update ist eine Kombination aus Standort, Timestamp und Daten zur verbleibenden Reichweite, während ein Event Daten zu einer potenziellen Brandstelle enthält.

Eine Demo und weitere ausführlichere technische Informationen zum Projekt sind im GitHub README[a] [bc] dokumentiert.

[a] https://github.com/waldbrandpraevention/frontend
[b] https://github.com/waldbrandpraevention/backend)
[c] https://kiwa.tech

2 Qualitätssicherung

In diesem Projekt kam Qualitätssicherung zum Einsatz. Diese befasst sich mit der Festlegung von Qualitätszielen (darunter fällt z. B. die Funktionalität der Software, sowie Anforderungen an das System wie einfache Wartbarkeit), dem Festlegen von Standards (z.B. zur Testprotokollierung) und der Sicherstellung und Durchführung von Maßnahmen, sodass die vereinbarten QS-Ziele erfüllt werden. Sie dient dazu, die Qualität der Software, die dem Auftraggeber am Ende des Projekts übergeben wird, zu garantieren und Fehler oder das Nichterfüllen bestimmter Anforderungen frühzeitig zu vermeiden.

2.1 Qualitätsziel 1: Wartbarkeit

Durch das QS-Ziel Wartbarkeit wird sichergestellt, dass die Software von anderen Programmierer*innen effizient weiterentwickelt werden kann und keine lange Einarbeitungszeit erfordert. Der bestehende Code kann einfach erweitert und weiterverwendet werden, ohne das größere Überarbeitungen notwendig sind oder gar das komplette Code-Design verändert werden muss. So können auch Anforderungen, die sich über Zeit verändern, berücksichtigt und schnell umgesetzt werden.
Mithilfe des Subziels Modularität erreichen wir, dass die Software in unabhängige Komponenten aufgeteilt werden kann, die keinerlei Einfluss mehr auf den Rest des Code haben.

Bezug zum Projekt: Es sind bereits jetzt Änderungen an der Software abzusehen, da sich die Anforderungen der Feuerwehr bzw. die Wichtigkeit von einzelnen Anforderungen nach intensiverer Nutzung der Anwendung verschieben können. So kann es bspw. notwendig werden, einzelne Kacheln (mit neuen Informationen) hinzuzufügen, andere werden vielleicht nicht gebraucht oder schlicht weniger häufiger genutzt, als zu Beginn angenommen. Hierbei ist es wichtig, es den Softwareentwickler*innen möglichst einfach zu machen, neue Kacheln hinzuzufügen, ohne den Code von Grund auf verändern zu müssen. Die Einarbeitung in den von uns geschriebenen Code sollte den Entwickler*innen also sehr einfach gemacht werden, sodass keine lange Einarbeitungszeit von Nöten ist. Zudem ist es hilfreich, die Kacheln als unabhängige Komponenten vom Rest des Codes abzugrenzen. Auf diese Weise können diese leicht verändert oder gänzlich entfernt werden, ohne einen Einfluss auf die restliche Funktionalität der Software zu haben.

Maßnahme: In unserem Projekt nutzen wir statische Analysetools, sogenannte Linting Tools. Für TypeScript kommt hier ESLint zum Einsatz. Für die Analyse des Python Codes nutzen wir Pylint. Diese Tools können Vorschläge machen, wie der Code umgestaltet (refactored) werden kann. Um die Modularität sicherzustellen, wird das Ergebnis der statischen Codeanalyse dann dazu verwendet, den bereits geschriebenen Quellcode zu überarbeiten, um Abhängigkeiten (beispielweise zwischen zwei Klassen oder Komponenten) aufzulösen. Um das oben genannten bessere Einarbeiten zu ermöglichen und sicherzustellen, dass eventuell vorhandene Abhängigkeiten innerhalb Quellcodes bereits aufgelöst wurden, werden vor jedem Merge in den Hauptstrang, manuelle Checks der Dokumentation durchgeführt. Hierzu verwenden wir als QS-Maßnahme Code Reviews. So wird außerdem sichergestellt, dass die vorhandene Dokumentation ausreichend ist, der Code die gewünschten

Anforderungen vollumfänglich erfüllt, die Implementation korrekt funktioniert und dem/der Benutzer*in die Nutzung der Software nicht durch das Vorhandensein von unnötigen Funktionen erschwert wird.

Prozessbeschreibung: Am Ende jeder Iteration, vor dem Merge in den Hauptstrang (main branch), führen wir für einige Dateien, die in der noch laufenden Iteration bearbeitet oder neu erstellt wurden, jeweils ein Code Review durch. Der/Die Entwickler*in, der/die den Code geschrieben hat, darf dabei nicht den eigenen Code reviewen. Das übernimmt meist ein*e Entwickler*in, der/die gerade an ähnlichen Aufgaben (Frontend/Backend) arbeitet und daher bereits mit der Syntax des Codes bzw. auch der Programmiersprache (Python/React) vertraut ist. Unsere Abgabe enthält exemplarisch ein Code Review pro Iteration als PDF-Dokument. Hierbei verwenden wir eine selbst erstellte Checkliste, die sich an dem im Moodle-Kurs des Bachelorpraktikums zur Verfügung gestellten generischen Code-Review Template orientiert. Wenn mittels Code Review keine Mängel festgestellt wurden, wird die entsprechende Klasse oder Datei in den main-branch eingebunden. Sollte geschriebener Code die Code-Review Checkliste nicht bestehen, so wird dieser von dem/der Entwickler*in, der/die den ursprünglichen Code implementiert hat, mithilfe der Ergebnisse aus dem Review, überarbeitet. Im Anschluss wird noch einmal ein Code Review durchgeführt.

2.2 Qualitätsziel 2: Funktionalität

Bezug zum Projekt: Durch das QS-Ziel Funktionalität wird sichergestellt, dass die Software die gewünschten Anforderungen vollumfänglich erfüllt und deren Funktionsweise validiert. Beispielsweise soll auf die KI-Vorhersagen, die dem/der Nutzer*in auf dem Dashboard angezeigt werden und bei Rauch- oder Feuerdetektion warnen, natürlich Verlass sein. Der/die Nutzer*in soll dabei jedoch nicht vom Vorhandensein unnötiger Funktionen aufgehalten wird.

Maßnahme: Als QS- Maßnahme nutzen wir manuelle (API-Kommunikation) und automatische Tests, um die Funktionweise der Software sicherzustellen. Hierbei soll bei jedem durchgeführten Test eine Code Coverage von mehr als 80% gewährleistet sein. Die Tests werden immer am Ende jeder Iteration (vor dem Zusammenführen in den Hauptstrang) durchgeführt. Zudem verwenden wir die Linting Tools in einer Continuos Integration Pipeline sowohl für Front- als auch Backend, d. h. das Projekt wird gebaut, anschließend werden automatische Tests ausgeführt. So wird garantiert, dass immer eine korrekte lauffähige Version vom Code im main-Branch vorliegt und die Arbeit mehrere Entwickler zu einem Hauptstrang kombiniert. Bei eventuell auftretenden Fehlern wird der Code nicht in den main-Branch eingebunden. Zudem setzen wir End-to-End Testing (E2E) ein, um die korrekte Darstellung und Funktion der Anwendung durch Nutzer*innen zu testen.

Prozessbeschreibung: Der Prozess zur Umsetzung dieser QS-Maßnahme beginnt damit, dass die Entwickler*innen Code-Änderungen in ihrem lokalen Entwicklungsbranch implementieren. Bevor der Code in den Hauptstrang (main-Branch) zusammengeführt wird, führen die Entwickler*innen manuelle und automatische Tests durch, um sicherzustellen, dass die Funktionalität der Software den Anforderungen entspricht. Dabei sollte die Code Coverage bei jedem durchgeführten Test mehr als 80% betragen. Wenn die Code Coverage bei einem Test unter 80% liegt, wird der/die Entwickler*in, der/die den Code implementiert hat, aufgefordert, zusätzliche Tests zu schreiben, um sicherzustellen, dass die betreffenden Bereiche der Software eine ordnungsgemäße Testabdeckung erfahren. Erst, wenn die Code Coverage über 80% liegt, wird der Code in den Hauptstrang integriert. Um die Code Coverage zu messen, verwenden wir ein Linting-Tool, das automatisch während der Continuous Integration Pipeline ausgeführt wird. Dieses Tool gibt uns eine genaue Auskunft darüber, welche Teile des Codes getestet bzw. mindestens einmal ausgeführt wurden, und welche nicht. Wir orientieren uns an dieser Ausgabe und arbeiten dann gezielt daran, die Code Coverage (bspw. durch

zusätzliche Tests) zu erhöhen.

Je nach Art und Komplexität der Änderungen werden zudem manuelle oder automatische Tests durchgeführt. Manuelle Tests kommen beispielweise vor allem im Frontend zum Einsatz, wo es meist einzig und allein um die korrekte Anzeige und Darstellung von Inhalten geht, was sich durch automatische Tests nur schwer prüfen lässt. Im Backend, vor allem bei aufwendigeren Berechnungen oder wiederkehrenden Funktionen werden hingegen automatische Tests verwendet. Diese werden dann in einer Continuos Integration Pipeline ausgeführt, die sicherstellt, dass der Code gebaut wird und alle automatischen Tests bestanden werden, bevor dieser in den main-Branch integriert wird. Wenn die Tests erfolgreich sind, werden die Code-Änderungen in den Hauptstrang übernommen. Andernfalls werden die Fehler von dem/der Entwickler*in, der/die den ursprünglichen Code geschrieben hat, mithilfe der Testergebnisse behoben. Im Anschluss wird derselbe Test noch einmal durchgeführt, um sicherzustellen, dass der Code nun korrekt implementiert ist und alle Anforderungen erfüllt. Sollte es trotz sorgfältiger Tests weiterhin zu Fehlern oder Problemen kommen, wird die Gruppe informiert und es wird gemeinsam versucht, eine Lösung zu finden. In unserer Abgabe ist pro Iteration ein PDF-Dokument enthalten, dass - exemplarisch für eine geänderte Datei - die Durchführung und das Ergebnis der manuellen bzw. automatischen Tests für diese Datei belegt.

2.3 Qualitätsziel 3: Portabilität

Bezug zum Projekt: Durch das QS-Ziel Portabilität wird sichergestellt, dass die Software leicht (d. h. ohne hohen Aufwand) an andere, bestehende Software- oder Hardwareplattformen angepasst werden kann. In unserem Projekt betrifft dies vor allem Kompatibilität mit verschiedenen Betriebssystemen und Webbrowsern. So soll die Software sowohl am Computer als auch auf mobilen Endgeräten (Tablet, Smartphone) korrekt funktionieren und Ansichten in angepasster Größe dargestellt werden.

Maßnahme: Es werden automatische Tests, wie bspw. das browserslist Tool [1] mit React verwendet, um Kompatibilität mit verschiedenen Browsern zu gewährleisten.
Wir verwenden Docker Container, sodass unsere Software auf unterschiedlichen Systemen ausführbar ist. Die Images für Front- und Backend werden automatisch mit der jeweils aktuellen Version des Front- bzw. Backend-Repositories gebaut und auf Docker Hub hochgeladen. Die Anwendung wird mit docker compose[2] und nginx[3] als Reverse Proxy ausgeführt. Alle Komponenten sind im Container-Image erhalten, sodass durch die Nutzung von Docker auch die Plattformunabhängigkeit unserer Anwendung (und damit die Kompatibilität mit verschiedensten Systemen) sichergestellt wird.
So wird sichergestellt, dass die Implementation auch auf fremder Hardware oder in einem anderen Browser korrekt funktioniert.

Prozessbeschreibung: Vor jedem Merge in den Hauptstrang (main-branch) wird der zu mergende Branch, mithilfe von automatisierten Tools (browserlist), auf die Kompatibilität mit verschiedenen Browsern geprüft. Hierfür werden alle Browser (außer Opera Mini), die mind. 0.2% Marktanteil haben und in den letzten 24 Monaten gewartet wurden, getestet. Sollte es dabei Konflikte mit einem dieser Browser geben, wird der Code von dem/der Entwickler*in, der/die den entsprechenden Code geschrieben hat, überarbeitet. Durch die integrierte CI/CD-Pipeline wird bei jedem Merge in den Hauptstrang ein Docker-Image erstellt und auf Docker Hub hochgeladen. So ist die aktuellste Version jederzeit plattformunabhängig nutzbar. In unserer Abgabe ist

[1] https://browsersl.ist/#q=defaults
[2] https://docs.docker.com/compose/
[3] https://nginx.org/

exemplarisch je ein Docker-Log (und somit implizit auch die bestandenen Tests des browserlist-Tools) pro Iteration als PDF-Dokument enthalten. Ausgenommen davon sind Iteration 1 und 2, da unsere CI/CD-Pipeline erst ab Iteration 3 eingesetzt wurde.

3 Abweichende QS Ziele

Alle genannten QS Ziele konnten wie geplant durchgeführt und durch Maßnahmen gesichert werden.

4 Projektverlauf und projektgefährdende Ereignisse

Projektverlauf: Das erste Planungstreffen mit dem*der Auftraggeber*in fand am 08.11.2022 statt. Die Ergebnisse des Projekts wurden am 17.03.2023 im finalen Treffen übergeben.

- Iteration 1: 28.11.2022 – 09.12.2022 *Velocity:* 0.47058823529411764
- Iteration 2: 09.12.2022 – 23.12.2022 *Velocity:* 0.6756756756756757
- Iteration 3: 23.12.2022 – 06.01.2023 *Velocity:* 0.6071428571428571
- Iteration 4: 06.01.2023 – 20.01.2023 *Velocity:* 0.7887323943661971
- Iteration 5: 20.01.2023 – 03.02.2023 *Velocity:* 1.5769230769230769
- Iteration 6: 03.02.2023 – 17.02.2023 *Velocity:* 0.5694444444444444
- Iteration 7: 17.02.2023 – 03.03.2023 *Velocity:* 0.3838383838383838
- Iteration 8: 03.03.2023 – 17.03.2023 *Velocity:* 0.2500000000000001

Abweichende Zeiten: -

Projektgefährdende Ereignisse: Im Laufe des Projekts gab es keine projektgefährdenden Ereignisse.

5 Softwarelizenz

Der für dieses Projekt erstellte Code steht im Einvernehmen aller Beteiligten unter folgender Lizenz:

MIT Lizenz:[1]

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

The Software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the Software.

[1] https://de.wikipedia.org/wiki/MIT-Lizenz