Projektdokumentation: Pollution Detection - Mobile App Entwicklung

Jona Rams

29. Oktober 2023

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

API Application Programming Interface.

CI Continuous Integration.

CRUD Create, Read, Update, Delete.

ERM Entity-Relationship-Modell.

HTTP Hypertext Transfer Protocol.

ID Identifier.

MoSCoW Must, Should, Could, Would.

ORM Object-Relational Mapping.

Python3 Version 3.10.10 der Python-Programmiersprache.

RESTful Representational State Transfer.

SQL Structured Query Language.

UUID Universally Unique Identifier.

venv virtual environment.

VSCode Visual Studio Code.

1 Einleitung

In dieser Projektdokumentation wird der Ablauf des Projekts zur Entwicklung einer mobilen App für die Kantonspolizei Zürich erläutert. Dieses Projekt wurde von Jona Rams im Rahmen seines Abschlussprojekts für den Fachinformatiker mit der Fachrichtung Anwendungsentwicklung im Auftrag der Kantonspolizei Zürich durchgeführt. Die LogObject AG erhielt den Auftrag zur Entwicklung einer mobilen Anwendung, die es den Polizeibeamten der Kantonspolizei Zürich ermöglichen soll, Verschmutzungen während ihrer Streifenarbeit einfach zu erfassen und zu zählen. Die erfassten Daten werden digital in eine Datenbank geladen, um eine effiziente Erfassung und Auswertung zu gewährleisten.

1.1 Projektbeschreibung

Die Hauptaufgabe dieses Projekts war die Erstellung eines Prototyps für die „Pollution Detection"-App im Auftrag der Kantonspolizei Zürich. Die App unterstützt Polizeibeamte bei der Erfassung und Quantifizierung von Verschmutzungen während ihrer Streifenarbeit. Die „Pollution Detection"-App bietet eine benutzerfreundliche Lösung, die es den Beamten ermöglicht, Verschmutzungen in von Kunden bereitgestellten geografischen Gebieten, den sogenannten Geoareas, effizient zu identifizieren und zu kategorisieren. Ein Schlüsselelement der App ist die nahtlose Integration der gesammelten Daten in eine zentrale Datenbank, wodurch eine zuverlässige Speicherung und einfache Analyse gewährleistet ist.
Die Anwendung ermöglicht den Beamten nicht nur den Zugriff auf gespeicherte Daten über eine intuitive Benutzeroberfläche, sondern bietet auch die Flexibilität, eigene Verschmutzungsarten hinzuzufügen, die speziell für ihre Geoareas relevant sind.
Das Hauptziel ist es, die Umweltüberwachung der Kantonspolizei Zürich zu optimieren und den Beamten ein leistungsfähiges Tool an die Hand zu geben, das sowohl benutzerfreundlich als auch anpassbar ist. Dies fördert die Erhaltung der öffentlichen Ordnung in den Geoareas und ermöglicht eine effektivere Bekämpfung der Umweltverschmutzung.

1.2 Projektziel

Das Hauptziel dieses Projekts war die Entwicklung der „Pollution Detection"-App zur Digitalisierung der Verschmutzungserfassung für die Kantonspolizei Zürich. Durch diese Digitalisierung soll der Prozess der Datenerfassung und -dokumentation für die Beamten erheblich vereinfacht und beschleunigt werden. Die App zielt darauf ab, den manuellen Aufwand zu reduzieren, die Genauigkeit der erfassten Daten zu erhöhen und den Beamten ein flexibles und benutzerfreundliches Tool zur Umweltüberwachung zur Verfügung zu stellen.

2 Projektplanung

2.1 Projektphasen

Gemäß den Vorgaben der IHK zu Essen war für die Realisierung des Projekts ein Zeitraum von 80 Stunden vorgesehen. Vor Beginn des Projekts wurde eine strukturierte Einteilung in verschiedene Entwicklungsphasen vorgenommen, um alle Aspekte der Softwareentwicklung abzudecken. Eine Übersicht über die geplanten Hauptphasen ist in Tabelle 1 darge-

stellt. Für eine ausführlichere Darstellung der geplanten Schritte innerhalb jeder Phase wird auf den Anhang verwiesen.

Phase	Zeit (Stunden)
Analyse	6
Entwurf	12
Implementierung	43
Tests	11
Deployment	4
Dokumentation	3

Tabelle 1: Projektphasen

2.2 Ressourcenplanung

Die vollständige Auflistung aller für das Projekt verwendeten Ressourcen finden Sie im Anhang A.1: Verwendete Ressourcen. Diese Planung beinhaltet nicht nur die verwendete Hard- und Software, sondern berücksichtigt auch das involvierte Personal. Bei der Auswahl der Software wurde insbesondere darauf geachtet, dass keine zusätzlichen Lizenzkosten entstehen und das notwendige Fachwissen bereits vorhanden ist. Zudem wurde darauf geachtet, dass die eingesetzten Technologien und Tools nicht gegen bestehende Richtlinien verstoßen. Die genauen technischen Spezifikationen und eingesetzten Tools, wie beispielsweise die Entwicklungsumgebung, werden im genannten Anhang detailliert aufgeführt.

2.3 Entwicklungsprozess

Vor dem eigentlichen Projektstart war es entscheidend, einen passenden Entwicklungsprozess auszuwählen, welcher die methodische Herangehensweise für die Projektumsetzung festlegt. Das Wasserfallmodell, das ich für dieses Vorhaben gewählt habe, ist ein geordneter und phasenbasierter Ansatz in der Softwareentwicklung. Dabei folgt jede Phase strikt auf die vorherige, wodurch klare Meilensteine und Abgrenzungen zwischen den einzelnen Entwicklungsschritten entstehen. Bei der Entwicklung der Benutzeroberfläche wählte ich jedoch einen agileren Ansatz, um einen kontinuierlichen Austausch mit dem Fachbereich zu gewährleisten und auf deren Feedback zeitnah reagieren zu können.

3 Analysephase

3.1 Ist-Analyse

Wie bereits in Abschnitt 1.1 (Projektbeschreibung) dargelegt, zielt das Projekt „Pollution Detection“ darauf ab, den Erfassungsprozess von Umweltverschmutzungsdaten für die Kantonspolizei Zürich zu digitalisieren. Aktuell wird die Erfassung von Verschmutzungsdaten während der Streifenarbeit manuell durchgeführt. Die Polizeibeamten notieren die Informationen auf physischen Datenträgern, was nicht nur zeitaufwendig, sondern auch

fehleranfällig ist.
Nach jeder Streife müssen die gesammelten Daten manuell in ein zentrales System eingegeben werden, was zu weiteren Verzögerungen und potenziellen Fehlern führt. Dieser analoge Ansatz hat mehrere Nachteile. Erstens erhöht er den Arbeitsaufwand für die Beamten erheblich, da sie sowohl vor Ort Daten erfassen als auch im Büro eingeben müssen. Zweitens besteht aufgrund des manuellen Eingabeprozesses ein erhöhtes Risiko von Fehlern, sei es durch Missverständnisse, Schreibfehler oder Übertragungsfehler. Dies kann zu inkonsistenten oder unvollständigen Datensätzen führen.
Zusätzlich zur manuellen Erfassung und Eingabe fehlt derzeit eine effiziente Methode, um die gesammelten Daten zentral zu speichern und abzurufen. Die Daten werden zwar physisch gesammelt, aber es gibt keine standardisierte Methode, um sie in einem digitalen Format zu konsolidieren und für zukünftige Referenzen oder Analysen zugänglich zu machen. Dies unterstreicht die Notwendigkeit einer digitalen Lösung, die den gesamten Prozess der Datenerfassung und -speicherung optimiert.

3.2 Wirschaftlichkeitsanalyse

3.2.1 Projektkosten

Im Folgenden werden die Kosten dargestellt, die während des Projekts anfallen. Zu berücksichtigen sind sowohl die Personalkosten des Entwicklers als auch der anderen Projektteilnehmer. Zusätzlich müssen die Kosten für die in Abschnitt 2.2 (Ressourcenplanung) aufgelisteten Ressourcen berücksichtigt werden.
Da genaue Personalkosten vertraulich sind, erfolgt die Kalkulation auf Basis von Stundensätzen, die intern festgelegt wurden. Die angegebenen Stundensätze umfassen hauptsächlich das Bruttogehalt sowie die arbeitgeberseitigen Sozialabgaben. Hinzu kommen die Kosten für die Nutzung der Ressourcen.
Für einen Mitarbeiter wird ein Stundensatz von 25,00 € veranschlagt. Für einen Auszubildenden beträgt der Stundensatz 10,00 €. Die Kosten für die Ressourcennutzung werden pauschal mit 12,00 € pro Stunde angesetzt.
Das Projekt hat eine geplante Laufzeit von 80 Stunden. In der nachfolgenden Tabelle Tabelle 2 sind die Kosten nach den einzelnen Projektvorgängen aufgeschlüsselt und summiert, um die Gesamtkosten des Projekts zu ermitteln. Diese belaufen sich auf insgesamt 1908,00 €.
Die aufgeführten Kosten entsprechen den mit unserer Firma vereinbarten Beträgen.

Vorgang	Zeit (Stunden)	Kosten pro Stunde (€)	Kosten gesamt (€)
Entwicklung	80	22 (10 Azubi + 12 Ressourcen)	1760
Code-Review	2	37 (25 Fachkraft + 12 Ressourcen)	74
Fachgespräch	2	37 (25 Fachkraft + 12 Ressourcen)	74
Gesamt	-	-	1908

Tabelle 2: Kostenaufstellung

3.2.2 Amortisationsdauer

Im Folgenden soll die Amortisationsdauer berechnet werden, um zu bestimmen, ab welchem Zeitpunkt sich die Investition in die Entwicklung der App finanziell auszahlt. Die

Entwicklungskosten für die App wurden bereits in Tabelle Tabelle 2 detailliert dargelegt und belaufen sich auf 1908 €. Eine bedeutende Zeitersparung ergibt sich hauptsächlich durch die Automatisierung des Datenerfassungsprozesses und das Entfallen manueller Übertragungsaufgaben. Weitere Details zur bisherigen Vorgehensweise und den damit verbundenen Zeitaufwänden können unter 3.1 Ist-Analyse entnommen werden.

1. Jährliche Zeiteinsparung

a) Manueller Prozess:

- 3 Tage die Woche das manuelle Notieren (10 mal am Tag): $3 \times 10 \times 5$ min $= 150$ min
- 3 Tage die Woche das Übertragen der Daten: 3×10 min $= 30$ min
- Gesamtwöchentliche Dauer: 150 min $+ 30$ min $= 180$ min
- Jährlich (angenommen 52 Wochen): 180 min $\times 52 = 9360$ min

b) Digitaler Prozess mit der App:

- 3 Tage die Woche das digitale Eintragen (10 mal am Tag): $3 \times 10 \times 1$ min $= 30$ min
- Jährlich (angenommen 52 Wochen): 30 min $\times 52 = 1560$ min
- Zeiteinsparung jährlich: 9360 min $- 1560$ min $= 7800$ min

2. Kostenersparnis

Unter der Annahme, dass die Kosten für einen Mitarbeiter, der diese Aufgaben durchführt, bei 40 €/Stunde liegen:
Kostenersparnis pro Jahr: $\frac{7800 \text{ min}}{60 \text{ min/h}} \times 40$ €/h $= 5200$ €

3. Amortisationszeit

Mit den gegebenen Entwicklungskosten von 1908 €:
Amortisationszeit: $\frac{1908 \text{ €}}{5200 \text{ €/Jahr}} = 0,37$ Jahre oder ungefähr 4,4 Monate.
Bei einer Nutzung von 3 Tagen die Woche und 10 mal am Tag würde sich die App demnach in etwa 4,4 Monaten amortisieren.
Zusammenfassend zeigt sich, dass die Investition in die App bereits nach kurzer Zeit finanziell lohnenswert ist. Eine detaillierte grafische Darstellung der Amortisation finden Sie im unter A.2: Amortisation

3.3 Anwendungsfälle

Um eine grobe Übersicht darüber zu erhalten, wie der Beamte mit der Anwendung arbeiten kann und welche Anwendungsfälle aus Endanwendersicht abgebildet werden müssen, wurde im Laufe der Analyse-Phase ein Use-Case-Diagramm erstellt. Dieses befindet sich unter A.3: Use-Case-Diagramm.
Der Hauptakteur, der später mit der Anwendung arbeiten wird, ist der Beamte. Dieser hat die Aufgabe, Daten in die App einzuspeisen und von den bereitgestellten Funktionen zu profitieren. Das Diagramm zeigt die verschiedenen Interaktionen und Aufgaben, die der Streifenbeamte mit der App durchführen kann. Diese Interaktionen definieren die verschiedenen Rollen und Berechtigungen, die innerhalb der Anwendung benötigt werden.

3.4 Lastenheft

Am Ende der Analysephase wurde gemeinsam mit den Projektbeteiligten ein Lastenheft erstellt. In dem Lastenheft sind alle zu berücksichtigenden Anforderungen an die zu implementierende Softwarelösung, sortiert nach ihrer Wichtigkeit, festgehalten. Die Formulierung der Anforderungen erfolgte unter Anwendung der Must, Should, Could, Would (MoSCoW)-Methode. Das bedeutet, dass in jeder Anforderung die Dringlichkeit in den Kategorien „muss“, „soll“, „kann“ und „würde gerne“ dargestellt wird. Diese Methodik half ebenfalls bei der Priorisierung der Anforderungen. Das komplette Lastenheft kann im Anhang A.5: Lastenheft eingesehen werden.

4 Entwurfsphase

4.1 Zielplattform

Wie bereits unter 1.1 (Projektbeschreibung) erläutert, wird das Projekt als eigenständige Android-App realisiert. Die Daten, mit denen die App interagieren soll, sind in einer bestehenden PostgreSQL-Datenbank gespeichert. Dies ermöglicht einen direkten Zugriff auf bereits vorhandene Datenstrukturen, ohne dass zusätzliche Datenbanksysteme installiert werden müssen.
Die Entscheidung, das Backend in Python3 zu entwickeln, basierte auf mehreren Faktoren. Zum einen verfüge ich bereits über umfangreiche Erfahrungen mit Python3, was den Entwicklungsprozess beschleunigt und die Fehleranfälligkeit reduziert. Zum anderen bietet Python3 eine breite Palette an Bibliotheken und Frameworks, die die Integration mit PostgreSQL erleichtern. Der Kunde gab mir in dieser Hinsicht freie Hand, was die Wahl der Technologie betrifft, und ich entschied mich für Python3 aufgrund seiner Effizienz und Anpassungsfähigkeit.
Zum Abschluss des Projekts wird die Anwendung zuerst auf einer Testplattform ausgerollt. Das Backend wird vorerst auf einem firmeninternen Laptop gehostet, um weitere Tests und Optimierungen in einer kontrollierten Umgebung zu ermöglichen. Parallel dazu wird eine PostgreSQL-Testdatenbank eingerichtet, um Testdaten effizient zu verwalten und den reibungslosen Ablauf der App sicherzustellen.
Die Benutzeroberfläche wurde mit dem Ionic-Framework auf Angular-Basis entwickelt. Dieses Framework wurde gewählt, da es eine nahtlose Integration mit Android bietet und die Entwicklung von responsiven und benutzerfreundlichen Oberflächen erleichtert. Tests können entweder über einen Android-Emulator oder direkt auf einem Android-Gerät durchgeführt werden, um sicherzustellen, dass die App in realen Bedingungen optimal funktioniert.

4.2 Architekturdesign

Für das „Pollution Detection“-Projekt wurde eine Drei-Schichten-Architektur gewählt. Die erste Schicht, die Präsentationsschicht, besteht aus dem Frontend, das mit dem Ionic Framework auf Angular-Basis entwickelt wurde. Dieses Frontend dient als Benutzerschnittstelle für die mobile Anwendung und stellt eine interaktive Oberfläche bereit, über die Benutzer Daten eingeben und Informationen anzeigen können. Es kommuniziert direkt mit dem Backend über HTTP-Anfragen, um Daten abzurufen oder zu senden.
Die zweite Schicht, die Geschäftslogikschicht, ist das in Python3 entwickelte Backend. Es

dient als zentrale Schnittstelle für alle Datenoperationen und stellt eine RESTful API bereit, die CRUD-Operationen (Create, Read, Update, Delete) unterstützt. Diese API ermöglicht es, Daten zu erstellen, abzurufen, zu aktualisieren und zu löschen und bildet die Brücke zwischen der Präsentationsschicht und der Datenzugriffsschicht. Innerhalb dieser Schicht ist auch die Geschäftslogik der Anwendung enthalten.
Die dritte und letzte Schicht, die Datenzugriffsschicht, beherbergt die PostgreSQL-Datenbank. In dieser Datenbank sind die Verschmutzungsdaten und andere relevante Informationen gespeichert. Sie bietet Mechanismen für den sicheren Zugriff auf die Daten und sorgt für deren Integrität und Konsistenz.
Die klare Trennung in diese drei Schichten bietet nicht nur Flexibilität und Skalierbarkeit, sondern erleichtert auch das Debugging und die Fehlerbehebung, da Probleme leichter auf eine der drei Schichten zurückgeführt werden können.

Die Interaktionen und Abhängigkeiten zwischen den beschriebenen Schichten sind im Anhang A.6: Komponentendiagramm dargestellt. Dieses Diagramm wurde bereits in der Entwurfsphase erstellt, um eine klare Struktur und Übersicht des Systems zu bieten.

4.3 Entwurf der Benutzteroberfläche

Wie unter 2.3 (Entwicklungsprozess) beschrieben, wurde für die Benutzeroberfläche ein intensiver Austausch mit den relevanten Stakeholdern vorgenommen, da sie die Hauptnutzer dieser Anwendung sein werden. Als ersten Schritt wurden Entwürfe erstellt. Die grafischen Darstellungen dieser Mockups finden Sie unter A.8: Mockups.

Im Header der App ist das „LOGOBJECT“-Logo platziert, um eine klare Zuordnung zur Marke zu ermöglichen. Der Hauptfokus der Startseite soll auf der Möglichkeit liegen, eine GeoArea auszuwählen. Diese Auswahl stellt ein zentrales Element der Anwendung dar und soll daher intuitiv und ohne Umwege für den Nutzer erreichbar sein.
Nach der Auswahl der GeoArea soll die App automatisch die zugehörigen Pollutions laden und anzeigen. Hierbei ist geplant, dem Nutzer interaktive Bedienelemente zur Verfügung zu stellen, mit denen er Anpassungen an den Zahlenwerten vornehmen kann. Dafür sind Plus- und Minus-Buttons vorgesehen, die eine einfache und direkte Bearbeitung ermöglichen.
Um eine effiziente Nutzung der App zu gewährleisten, wurde in den Entwürfen darauf geachtet, dass alle Funktionen mit wenigen Klicks erreichbar sind und der Nutzer stets einen klaren Überblick über die dargestellten Inhalte behält.
Die Rücksprache mit den Stakeholdern hat bestätigt, dass dieser geplante Aufbau den Anforderungen entspricht, wobei jedoch Wert darauf gelegt wird, dass die Navigation einfach bleibt und der Benutzer nicht durch zu viele Unterseiten navigieren muss.

4.4 Datenmodell

Das Datenmodell, welches für die geplante mobile App konzipiert wurde, setzt sich aus den Entitäten **User**, **GeoArea** und **Pollution** zusammen.
Die Entität **User** repräsentiert einen Anwender der App und besitzt folgende Attribute:

- `id`: Ein eindeutiger Identifikator für jeden Benutzer.
- `email`: Die E-Mail-Adresse des Benutzers.

- `password`: Das Passwort des Benutzers.

Die Entitäten **GeoArea** und **Pollution** wurden vom Kunden vorgegeben.
Die Entität **GeoArea** kapselt Informationen zu geografischen Bereichen, in denen **Pollutions** erfasst werden. Sie verfügt über folgende Attribute:

- `id`: Ein eindeutiger Identifikator für jede GeoArea.
- `datecreated`: Das Erstellungsdatum der GeoArea.
- `language`: Die Sprache, die in dieser GeoArea vorherrschend ist.
- `last_update`: Das Datum des letzten Updates der GeoArea.
- `mandant`: Der Mandant der GeoArea.
- `admincomment`: Ein Kommentarfeld für Administratoren.
- `automaticsearch`: Ein Boolean-Wert, der angibt, ob in dieser GeoArea eine automatische Suche aktiviert ist.
- `name`: Der Name der GeoArea.
- `polygon`: Die geografischen Grenzen der GeoArea.

Die Entität **Pollution** beschreibt spezifische Verschmutzungen oder Umweltbelastungen innerhalb einer GeoArea und hat folgende Attribute:

- `id`: Ein eindeutiger Identifikator für jede Pollution.
- `name`: Der Name der Pollution.
- `count`: Eine Zählung oder Intensität der Pollution.
- `description`: Eine Beschreibung der Pollution.
- `geoarea_fk`: Ein Fremdschlüssel, der die Pollution mit einer GeoArea verknüpft.

Eine essenzielle Beziehung in diesem Modell ist die zwischen **GeoArea** und **Pollution**. Jede **Pollution** muss zwingend einer **GeoArea** zugeordnet sein, wodurch eine 1:n-Beziehung entsteht. Dies bedeutet, dass eine GeoArea mehrere Pollutions besitzen kann, wohingegen eine Pollution immer genau einer GeoArea angehört.

Der **User** könnte in weiteren Ausarbeitungen des Modells in Beziehung zu **GeoAreas** oder **Pollutions** stehen, etwa durch das Erfassen oder Bearbeiten von Daten. Diese potenziellen Beziehungen sind jedoch zum jetzigen Zeitpunkt noch nicht finalisiert.

Im finalen Schritt des Entwicklungsprozesses wird dieses Datenmodell in ein Entity-Relationship-Modell (ERM) überführt. Dieses dient als Blaupause für die Implementierung der Datenstruktur in der eigentlichen App und stellt die Beziehungen zwischen den Entitäten klar dar. Das dazugehörige ERM wird im Anhang A.8: Entity-Relationship-Modell dargestellt und bildet die Grundlage für die spätere Implementierung der Domänenklassen.

4.5 Geschäftslogik

Der grundsätzliche Aufbau der Anwendung wurde bereits im Abschnitt 4.2 (Architekturdesign) erörtert. Um eine detaillierte Darstellung der Geschäftslogik und deren Implementierung zu bieten, habe ich eigenständig ein Klassendiagramm angefertigt. Dieses Diagramm dient als visuelle Darstellung der internen Struktur und verdeutlicht die Trennung sowie die Beziehungen der einzelnen Komponenten. Es bietet zudem eine Übersicht über die Schnittstellen und die Funktionsweisen der verschiedenen Services, Klassen und Module. Für eine ausführliche Ansicht des Klassendiagramms verweise ich auf den Anhang A.9: Klassendiagramm.

4.6 Code-Qualitätssicherung

Im Laufe des Projektes wurden gezielte Maßnahmen ergriffen, um die Qualität des Codes zu gewährleisten. Ein zentrales Element dieses Prozesses war das Code-Review mit dem Ausbilder, wodurch die fachliche und technische Korrektheit des Projekts überprüft wurde.

Des Weiteren wurde durch GitHub Actions von Anfang an eine Continuous Integration (CI) implementiert. Dies gewährleistete eine automatische Überprüfung des Codes nach jedem Push und ermöglichte das rasche Erkennen und Beheben von Fehlern.

Für die detaillierte Codeüberwachung kam Flake8 zum Einsatz, welches den Python-Code auf Fehler, potenzielle Probleme und die Einhaltung des PEP 8-Stils untersuchte. Ein Screenshot der Analyseergebnisse von Flake8 ist im Anhang A.10: Statische Codeanalyse zu finden.

4.7 Deployment

Bei dem vorliegenden Projekt handelt es sich um eine mobile Anwendung. Aufgrund dessen ist das Deployment auf einem Firmenlaptop innerhalb des Firmennetzwerks notwendig, um in einer Testumgebung die Anwendung zu evaluieren. Das zugehörige Verteilungsdiagramm, welches den gesamten Prozess und die Architektur darstellt, ist im Anhang A.12 abgebildet.

Die Entwicklung der mobilen App erfolgte mittels Ionic auf Angular-Basis. In diesem Projekt wurde *Capacitor* als Cross-Platform Native Runtime eingesetzt. Mit den Befehlen `ionic build`, gefolgt von `npx cap sync` und `npx cap open android`, wird die Web-App in eine native Android-App konvertiert. Obwohl der Fokus auf *Capacitor* liegt, wird im Hintergrund *Gradle* als Build-System verwendet, sobald das Projekt in *Android Studio* geöffnet ist, um die Android-App zu bauen.

Für das Backend und die Datenbank der Anwendung kommen *Docker-Container* zum Einsatz. *Docker* erlaubt es, Anwendungen konsistent in Containern auszuführen. Bei jeder Aktualisierung des Quellcodes wird automatisch ein neues *Docker-Image* für das Backend erstellt und ist somit stets aktuell für den Betrieb auf dem Firmenlaptop verfügbar.

Nach erfolgreichem Deployment der Backend-Komponenten und der Datenbank kann die mobile App in der Testumgebung evaluiert werden. Diese kann sowohl über einen Android-Emulator, beispielsweise von Android Studio, direkt auf dem Laptop ausgeführt

oder von einem Smartphone, das im Firmennetzwerk eingeloggt ist, aufgerufen werden.

Insgesamt wurde durch den gezielten Einsatz verschiedener Technologien eine effiziente Testumgebung geschaffen, die eine gründliche Überprüfung der App vor einem endgültigen Rollout sicherstellt.

4.8 Plichtenhfeft

Basierend auf den in der Entwurfsphase entwickelten Konzepten wurde ein Pflichtenheft erstellt. Dieses baut direkt auf dem in Abschnitt 3.4 (Lastenheft) beschriebenen Lastenheft auf und definiert die konkrete Umsetzung der identifizierten Anforderungen. Das Pflichtenheft dient nicht nur als Überprüfungsinstrument am Projektende, sondern begleitet und leitet uns auch während der gesamten Entwicklungsphase. Ein Auszug daraus ist im Anhang A.11: Pflichtenheft zu finden.

5 Implementierungsphase

Zu Beginn der Implementierungsphase stand die Einrichtung der notwendigen Projektstrukturen im Vordergrund.

Für das Backend wurde zuerst in VSCode ein Python3-Projekt ins Leben gerufen. Dies ist ein essenzieller Schritt, um eine isolierte Entwicklungsumgebung zu gewährleisten und sicherzustellen, dass alle Abhängigkeiten und Bibliotheken korrekt verwaltet werden. Dafür wurde eine virtual environment (venv) erstellt. Diese bietet die Möglichkeit, Abhängigkeiten projektbezogen zu isolieren und somit sicherzustellen, dass unterschiedliche Projekte mit potenziell unterschiedlichen Anforderungen an die Paketversionen nicht in Konflikt geraten.

Im Bereich des Frontends begann der Implementierungsprozess mit der Installation von Ionic. Ionic ist ein weit verbreitetes Framework zur Erstellung plattformübergreifender mobiler Anwendungen unter Verwendung von Webtechnologien. Nach der erfolgreichen Installation wurde mithilfe von Ionic ein neues Projekt auf der Grundlage von Angular generiert. Dieser Schritt bietet den Vorteil, dass Ionic eine Grundstruktur für das Projekt vorgibt. Dies erleichtert den Entwicklungsprozess erheblich, da bestimmte, immer wiederkehrende Aufgaben, wie die Einrichtung von Verzeichnisstrukturen oder die Integration von Basiskomponenten, automatisiert werden.

5.1 Implentierung des Datenmodelle und der Datenbank

In der Entwicklung der „Pollution Detection“-App war es notwendig, die Datenstruktur adäquat im Backend abzubilden. Ein wichtiges Werkzeug, das dabei zum Einsatz kam, war *SQLAlchemy*, ein SQL Toolkit und Object-Relational Mapping (ORM) System für Python3. Es ermöglicht das Mapping von in Python3 definierten Klassen zu Datenbanktabellen und umgekehrt.

Innerhalb dieses Systems gibt es Unterschiede in der Implementierung bestimmter Entitäten. Zum Beispiel wurden für die Klassen `User` und `Pollution` UUIDs mit einer

Hexadezimaldarstellung für ihre IDs verwendet. Das `__tablename__`-Attribut in *SQLAlchemy* bestimmt den Namen der Tabelle in der Datenbank, und in unserem Fall entspricht der Klassenname dem Tabellennamen. Fremdschlüssel werden durch `ForeignKey` und die zugehörigen Relationen durch `relationship` repräsentiert. Primärschlüssel werden durch das `primary_key=True` Argument in der Spaltendefinition angegeben.

Es ist wichtig zu erwähnen, dass sowohl die `Pollution`- als auch die `Geoarea`-Klasse vom Kunden übernommen wurden. Während `Pollution` eine neuere Implementierung darstellt und daher UUIDs in Hexadezimalform verwendet, nutzt `Geoarea` aufgrund ihres Alters Ganzzahlen als IDs.

Abschließend sollte angemerkt werden, dass der Kunde ein Backup bereitstellte, das eine Anzahl von Testgebieten enthielt. Diese wurden in der Entwicklungsphase intensiv zum Testen und Verifizieren der Funktionalität des Systems genutzt.

5.2 Implementierung der Geschfäftslogik

Die Geschäftslogik fokussiert sich auf die Abbildung und Verarbeitung der Daten entsprechend der fachlichen Anforderungen. Entsprechend den in Abschnitt 3.3 (Anwendungsfälle) erläuterten Anwendungsfällen, wurden die Hauptservices in der Service-Schicht realisiert. Hierbei wurden spezielle Service-Dateien, wie der `auth_service` für Authentifizierungsprozesse, der `geoarea_service` zur Abfrage von geografischen Gebieten und der `pollution_service` zur umfassenden Verwaltung von Pollution-Daten, erstellt.

Zur Implementierung der in den Services definierten Methoden und zur Sicherstellung eines effizienten Zugriffs auf die Daten, wurde *SQLAlchemy* als ORM genutzt. Durch die Verwendung von *SQLAlchemy* wird eine klare Trennung zwischen Datenzugriff und Geschäftslogik erreicht. Dies bietet auch die Möglichkeit, bei Bedarf einfach auf andere Datenbank-Implementierungen umzusteigen.

In den Services wird der direkte Zugriff auf die Datenmodelle und deren Methoden gehandhabt. So wird beispielsweise im `pollution_service` eine Schnittstelle zur Datenmanipulation genutzt, welche die CRUD-Operationen auf Pollution-Daten in der Datenbank ermöglicht. Ein zentraler Bestandteil der Logik innerhalb des `Pollution`-Services besteht darin, die Anzahl der gemeldeten Umweltverschmutzungen zu aktualisieren. Bei der Aktualisierung wird besonders darauf geachtet, dass die Werte korrekt und konsistent sind. Der `PollutionCount`-Resource verwendet entsprechende Schnittstellen, um die Anzahl der Verschmutzungen zu ändern. Bei einer Anfrage zum Aktualisieren eines `Pollution`-Eintrags wird zuerst überprüft, ob dieser Eintrag in der Datenbank existiert. Wenn der Eintrag gefunden wird, wird die `count`-Eigenschaft des `Pollution`-Objekts mit dem neuen Wert aktualisiert. Sollte während dieses Prozesses ein Fehler auftreten, etwa bei Datenbankkonflikten, wird die Transaktion rückgängig gemacht, um die Integrität der Daten zu gewährleisten. Im Anhang A.12: Listing von Python-Code ist der entsprechende Code der `PollutionCount`-Klasse zu sehen.

Ein besonderes Augenmerk wurde auf den `auth_service` gelegt, welcher nicht nur die Funktionen für Login und Registrierung bietet, sondern auch sicherstellt, dass Benutzerdaten sicher und vertraulich behandelt werden. Dies beinhaltet unter anderem die sichere

Speicherung von Passwörtern und die Generierung von Tokens für authentifizierte Sessions.

5.3 Implementierung und Durchführung der Tests

Die Tests in unserer Anwendung können in zwei Hauptkategorien unterteilt werden: Tests für Hilfsfunktionen und Tests für API-Endpunkte. Die Hilfsfunktionen, die im Modul `test_utils` getestet werden, überprüfen primär, ob E-Mail-Adressen als gültig erkannt werden und ob Passwörter korrekt gehasht und überprüft werden können.

Für die Tests der Hilfsfunktionen ergänzende Codeausschnitte im Anhang A.13: Listing von unittest.

Im Modul `test_endpoints` konzentrieren sich die Tests auf verschiedene API-Endpunkte. Bevor jeder Test durchgeführt wird, bereitet die Methode `setUp` die Testumgebung vor. Hier wird die Datenbank initialisiert und Testdaten werden eingefügt. Nachdem ein Test abgeschlossen ist, sorgt die Methode `tearDown` dafür, dass alle Änderungen rückgängig gemacht werden, sodass die Datenbank in ihren ursprünglichen Zustand zurückversetzt wird. Dies stellt sicher, dass Tests unter den gleichen Voraussetzungen durchgeführt werden und sich nicht gegenseitig beeinflussen.

Zu den Integrationstests gibt es ergänzende Codeausschnitte im Anhang A.13: Listing von unittest, die Einblicke in die Implementierung bieten.

5.4 Implementierung der Oberfläche

Auf Basis der in Abschnitt **4.3** (Entwurf der Benutzeroberfläche) dargestellten Mockups wurde die Benutzeroberfläche der mobilen Anwendung entwickelt. Die Implementierung erfolgte unter Verwendung des *Ionic-Frameworks* auf *Angular*-Basis, wie in Abschnitt **4.2** (Architekturdesign) beschrieben. Ionic bietet eine Sammlung von UI-Komponenten, die für mobile Anwendungen optimiert sind. Diese Komponenten sind in Web-Technologien wie HTML, CSS und Typescript geschrieben und bieten eine konsistente, plattformübergreifende Benutzererfahrung.

Ein zentrales Konzept in Angular ist das *Two-Way Data Binding*, welches eine direkte Interaktion zwischen der Benutzeroberfläche und den zugrunde liegenden Datenmodellen ermöglicht. Ein praktisches Beispiel aus der Anwendung zeigt ein Eingabefeld, das mit einer Variablen im Datenmodell verknüpft ist:

```
<ion-input color="light" type="number" [(ngModel)]="item.count"
inputmode="numeric" (ionChange)="onCountChange(item)"></ion-input>
```

In diesem Ausschnitt wird die Variable `item.count` direkt mit dem Wert des Eingabefeldes synchronisiert. Wenn der Benutzer einen Wert in das Feld eingibt, wird `item.count` automatisch aktualisiert und umgekehrt. Der Event `(ionChange)` sorgt zudem dafür, dass bei jeder Änderung die Methode `onCountChange(item)` aufgerufen wird.

Ein besonderes Augenmerk wurde auf die Benutzerfreundlichkeit und die Fehlertoleranz der Anwendung gelegt. Bei möglichen Fehlern oder Unklarheiten erhält der Benutzer aussagekräftige Fehler- und Informationsmeldungen. Dies stellt sicher, dass der Benutzer stets

informiert ist und gegebenenfalls notwendige Korrekturen oder Aktionen vornehmen kann.

Die Implementierung folgt allgemeinen Richtlinien der Software-Ergonomie, um eine intuitive und effiziente Benutzerführung zu gewährleisten. Dies beinhaltet beispielsweise eine Minimierung der notwendigen Interaktionen und die Bereitstellung von klaren, selbsterklärenden Steuerelementen.

5.5 Testen der Anwendung

6 Abnahme und Deploymentphase

7 Dokumetation

8 Fazit

Literaturverzeichnis

A Anhang

A.1 Detaillierte Zeitpanung

Analysephase	6h
• Durchführen der Ist-Analyse	2h
• Durchführen der Wirtschaftlichkeitsanalyse und Erstellung einer Amortisationsrechnung	2h
• In Zusammenarbeit mit dem Fachbereich, die Erstellung eines Lastenheftes	2h
Entwurf	12h
• Entwerfen der Benutzeroberfläche inkl. Erstellen von Mock-Ups	4h
• Entwerfen der Datenbankstruktur inkl. Erstellen eines ER-Modells	3h
• Planung der Architektur	3h
• Erstellen des Pflichtenheftes	2h
Implementierung des Backends und der Datenbank	26h
• Anlegen der Datenbank inkl. Tabellen	3h
• Anlegen des Backend-Projekts	1h
• Implementieren der RESTful API	2h
• Herstellen der Datenbankverbindung inkl. Konfiguration	2h
• Implementieren der Datenbankmodelle	6h
• Implementieren der Services	12h
Implementierung und Durchführung der Tests	11h
• Anlegen einer Testdatenbank	1h
• Implementieren der API-Tests	10h
Implementierung der Oberfläche	17h
• Anlegen des Frontend-Projekts	2h
• Umsetzung der Benutzeroberfläche	15h
Deployment	4h
• Code-Review mit dem Ausbilder	2h
• Deployment des Prototyps in eine Testumgebung	2h
Dokumentation	3h
• Erstellen des Benutzerhandbuchs	3h
Gesamt	80h

A.2 Verwendete Ressourcen

Hardware

- Büroarbeitsplatz mit Lenovo ThinkPad Laptop

Software

- Git – Verteilte Versionsverwaltung
- Gradle – Build-Management-Automatisierungstool
- MiKTeX – Distribution des Textsatzsystems TEX
- PGAdmin4 – Verwaltungstool für PostgreSQL-Datenbanken
- PostgreSQL – Relationales Datenbanksystem
- Visual Studio Code (VSCode) – Quellcode-Editor
- Windows 11 Pro – Betriebssystem

Personal

- Informatiker – Review des Codes
- Auszubildender – Umsetzung des Projekts
- Ansprechpartner der Kantonspolizei Zürich

A.3 Amortisation

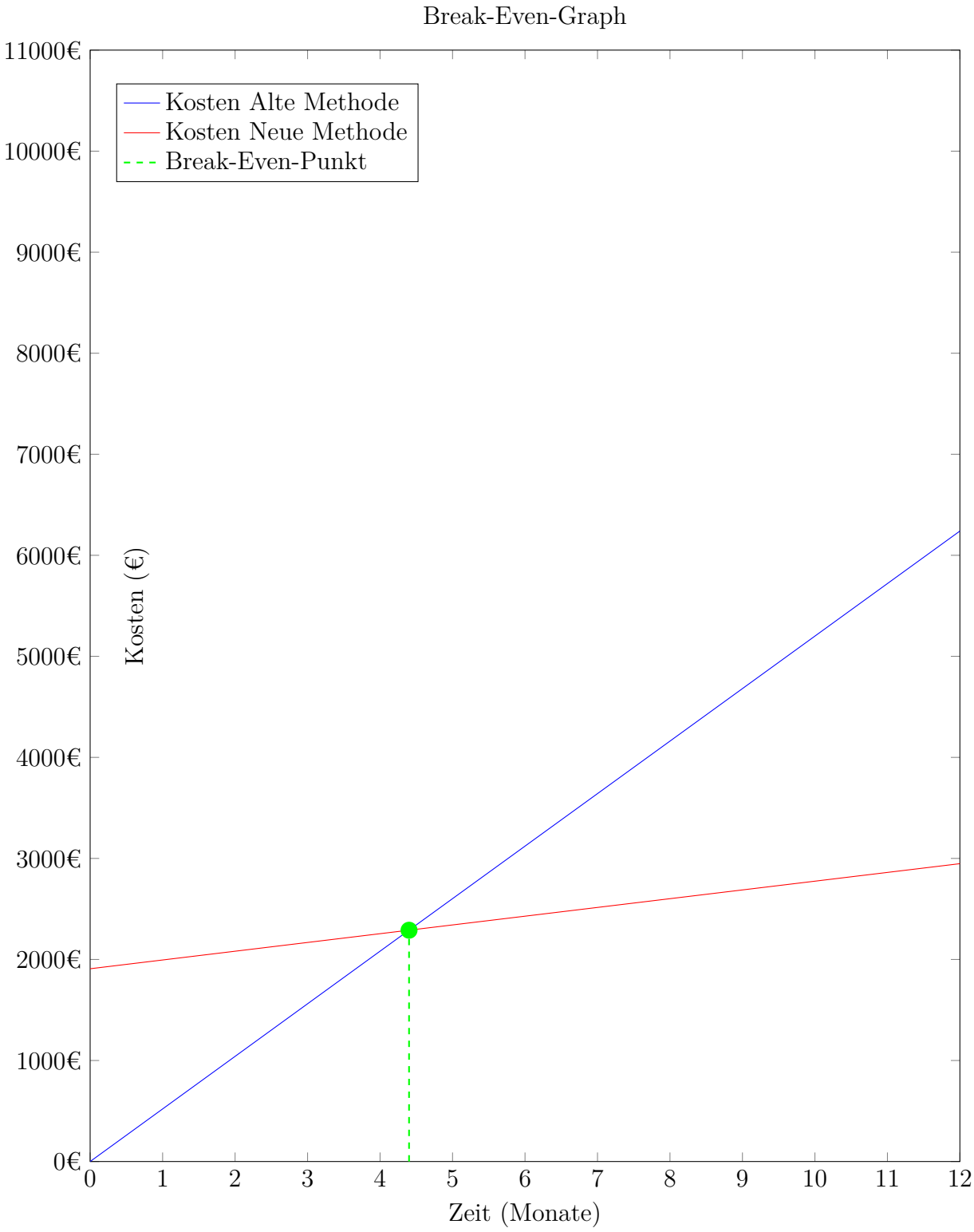

Abbildung 1: Amortisation

A.4 Use-Case-Diagramm

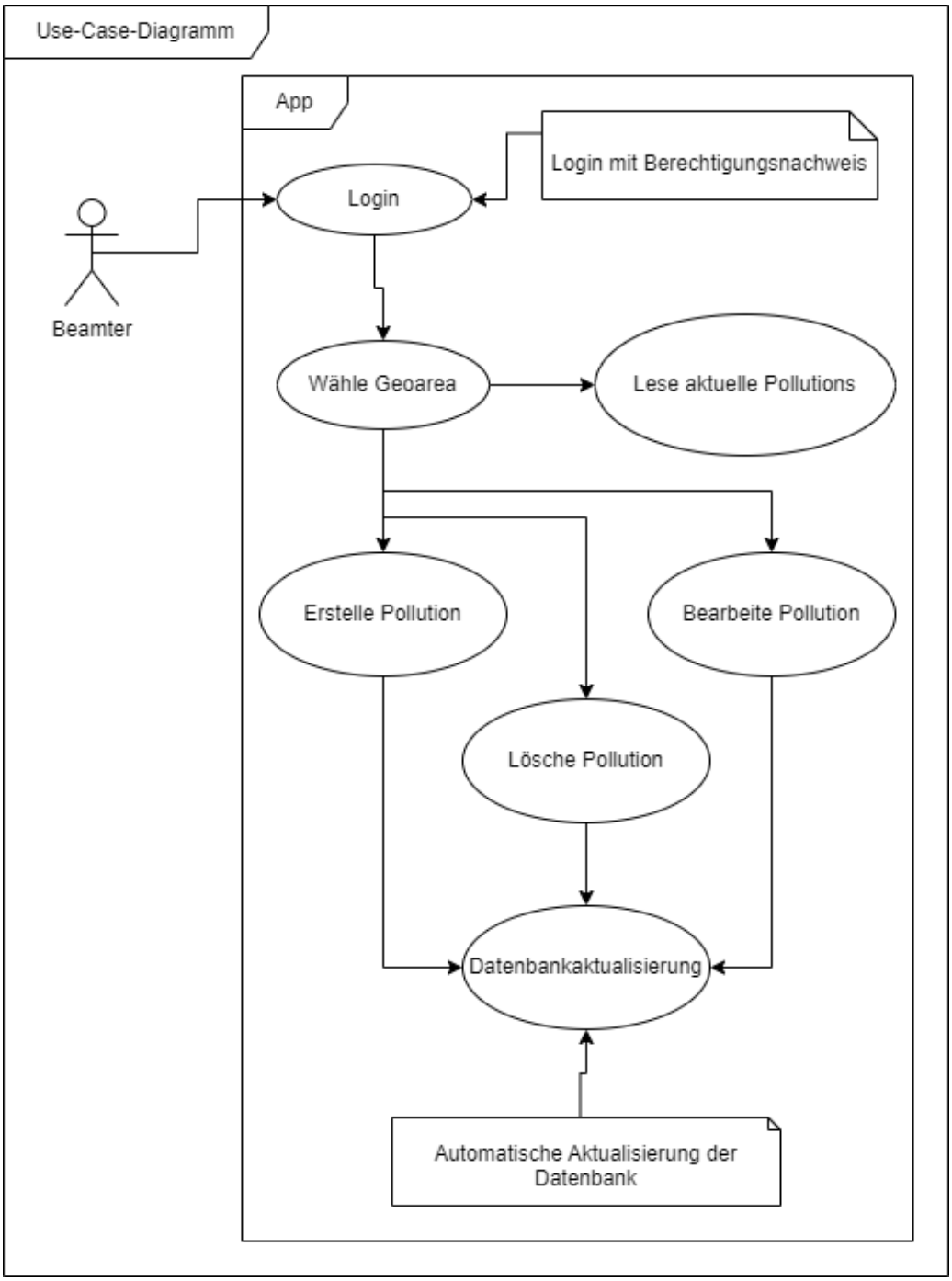

Abbildung 2: Use-Case-Diagramm der „Pollution Detection“-App

A.5 Lastenheft

Für die Entwicklung der „Pollution Detection“-App für die Kantonspolizei Zürich sind folgende Anforderungen festgelegt:

1. Datenverarbeitung und Integration

- Die App **muss** Verschmutzungsdaten in Echtzeit erfassen und kategorisieren.
- Es **muss** möglich sein, spezifische Verschmutzungsarten hinzuzufügen, die für bestimmte geografische Gebiete (Geoareas) relevant sind.
- Es **muss** eine Funktion geben, die den Beamten erlaubt, Verschmutzungsarten zu bearbeiten und zu löschen.
- Die erfassten Daten **könnten** mit anderen Datenquellen oder Systemen der Kantonspolizei synchronisiert werden.

2. Benutzeroberfläche und Interaktivität

- Die App **muss** über eine intuitive Benutzeroberfläche verfügen, die eine schnelle und einfache Erfassung von Verschmutzungen ermöglicht.
- Die App **könnte** in der Lage sein, die genaue Position der Verschmutzung mithilfe von GPS oder anderen geografischen Tools zu erfassen.

3. Zugänglichkeit

- Die App **muss** auf Mobilgeräten funktionieren.
- Die App **sollte** eine sichere Authentifizierungsmethode haben.

4. Sonstige Anforderungen

- Es **sollten** regelmäßige Updates und Wartungen durchgeführt werden, um die App auf dem neuesten Stand zu halten und Sicherheitsrisiken zu minimieren.
- Es **wäre** gut, über kontinuierlichen Support und mögliche Einweisungen zu verfügen.

A.6 Komponentendiagramm

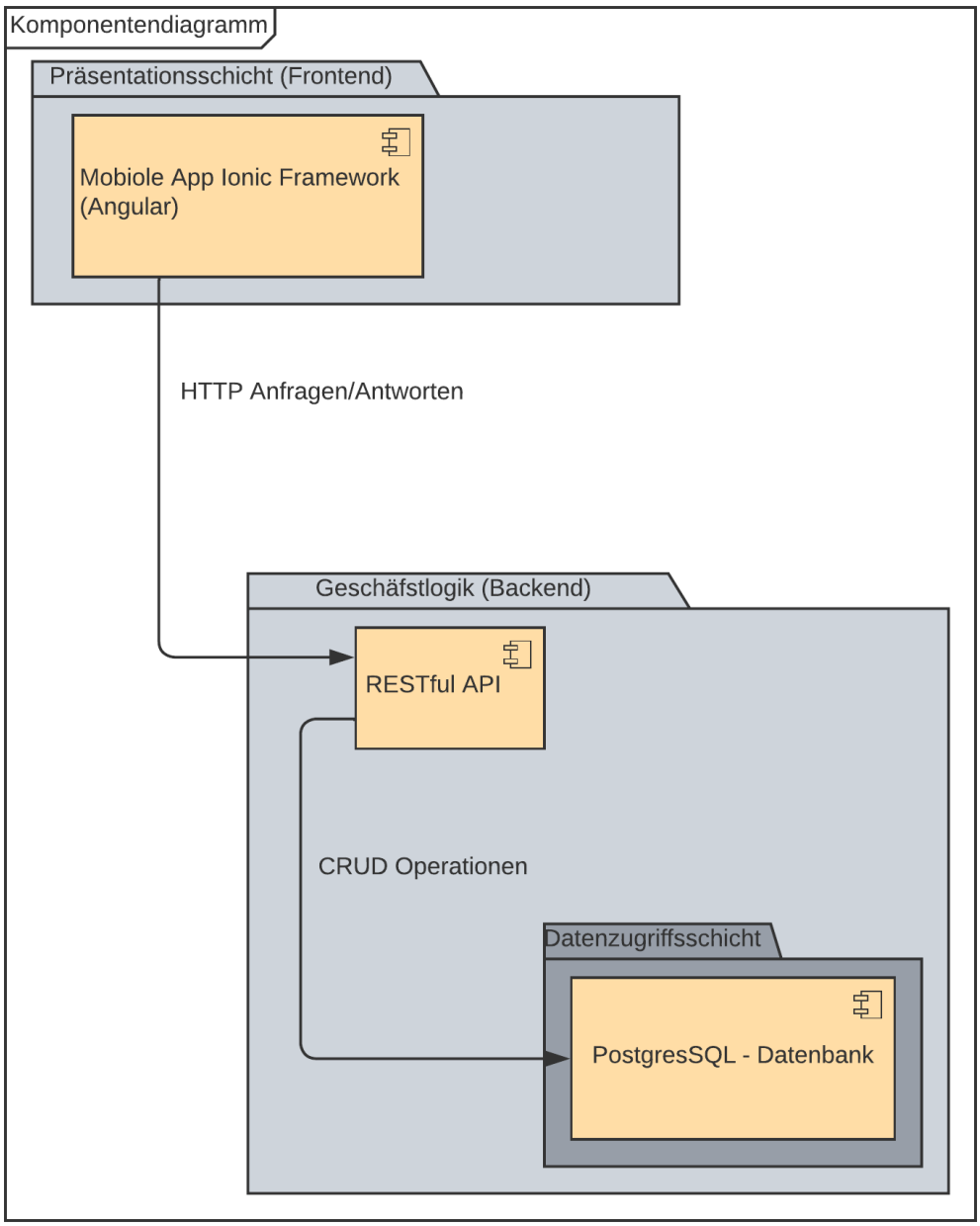

Abbildung 3: Komponentendiagramm

A.7 Mockups

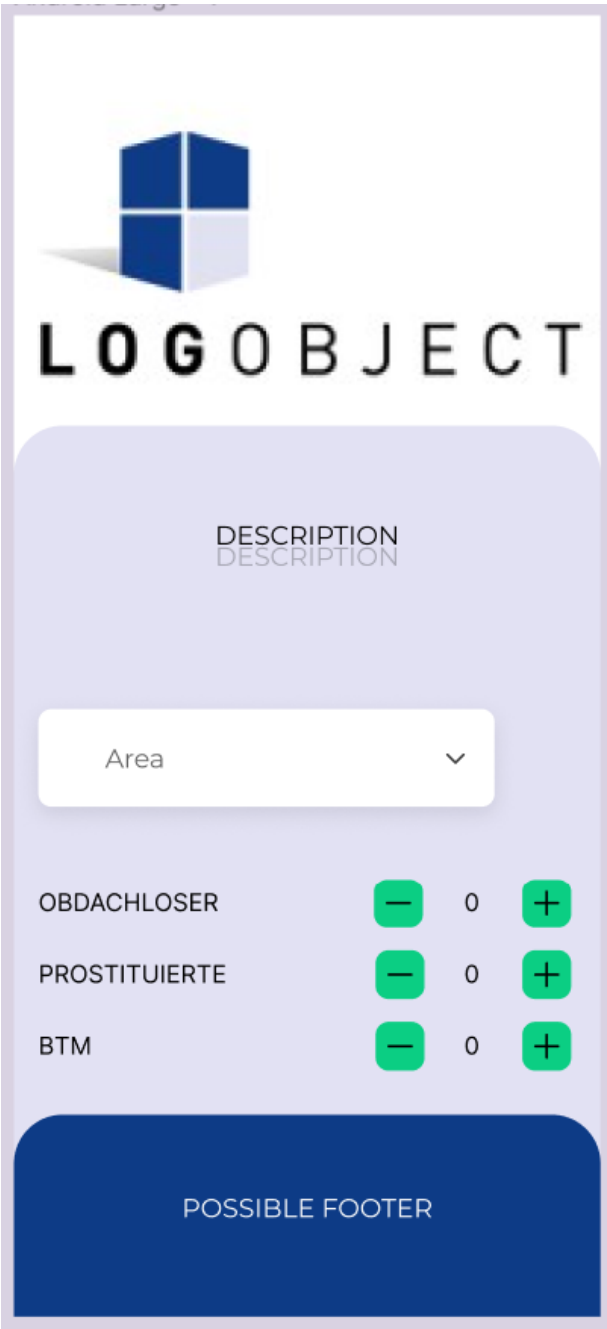

Abbildung 4: Mockups

A.8 Entity-Relationship-Modell

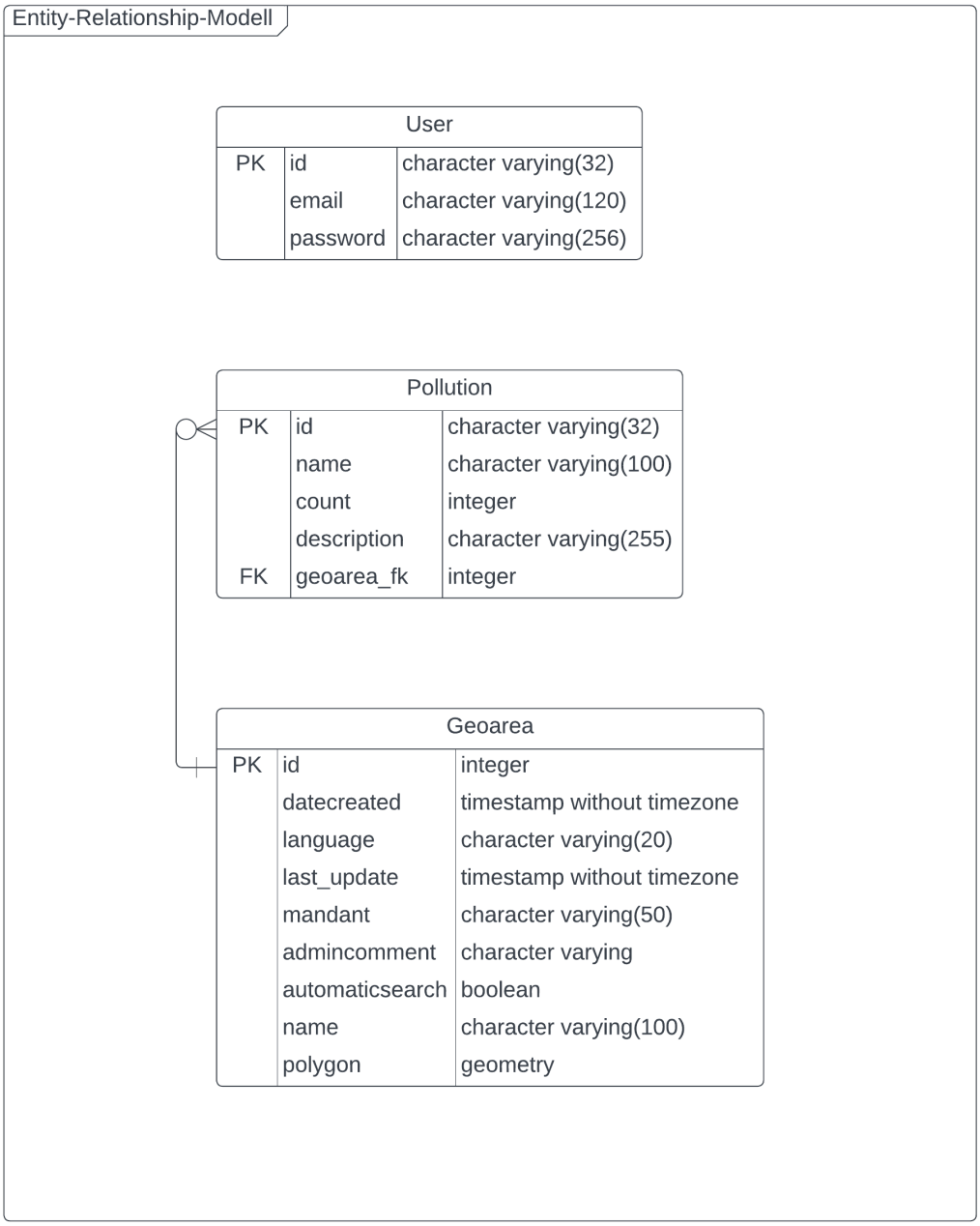

Abbildung 5: Entity-Relationship-Modell

A.9 Klassendiagramm Auszug

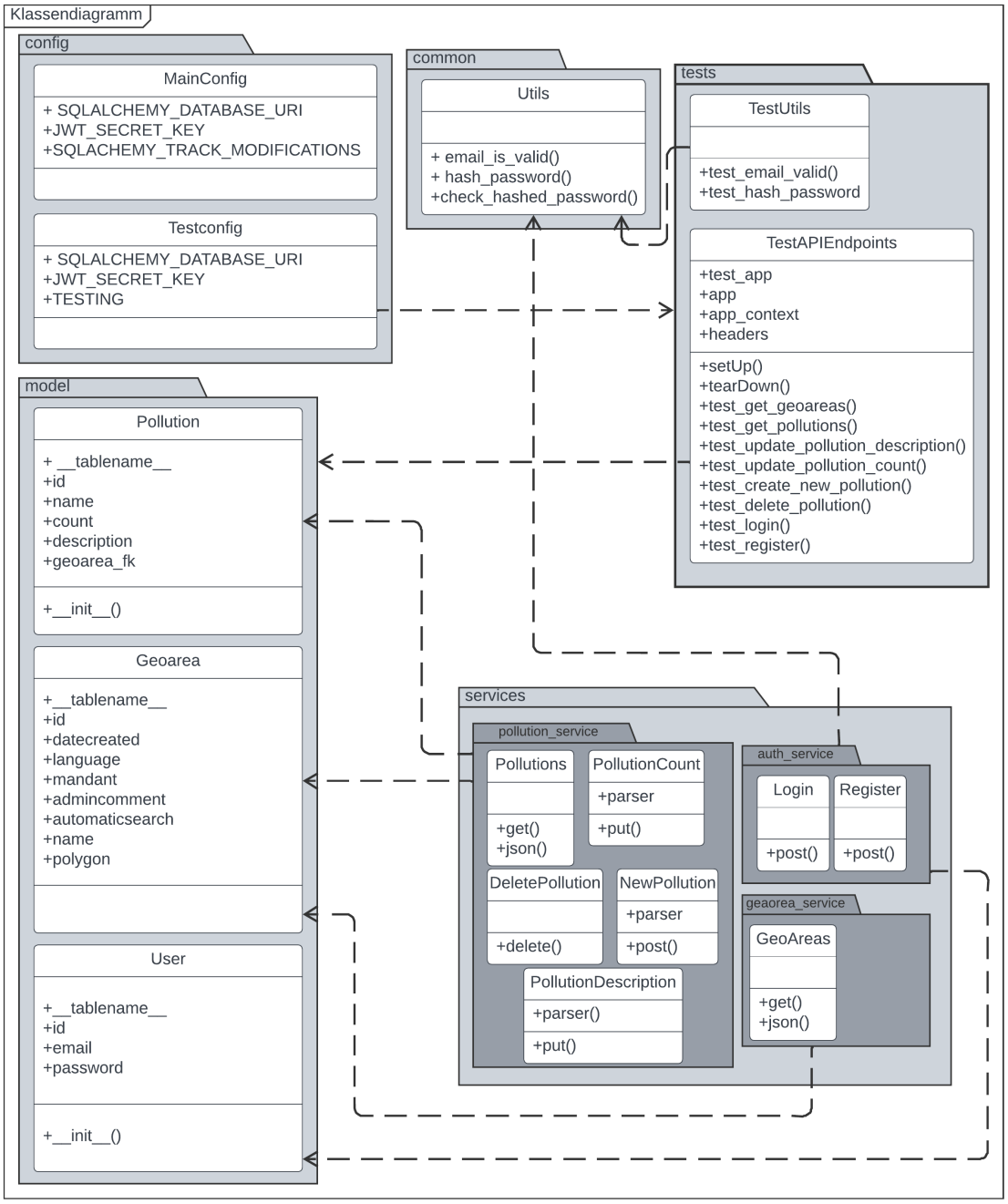

Abbildung 6: Auszug des Klassendiagramms

A.10 Statische Codeanalyse

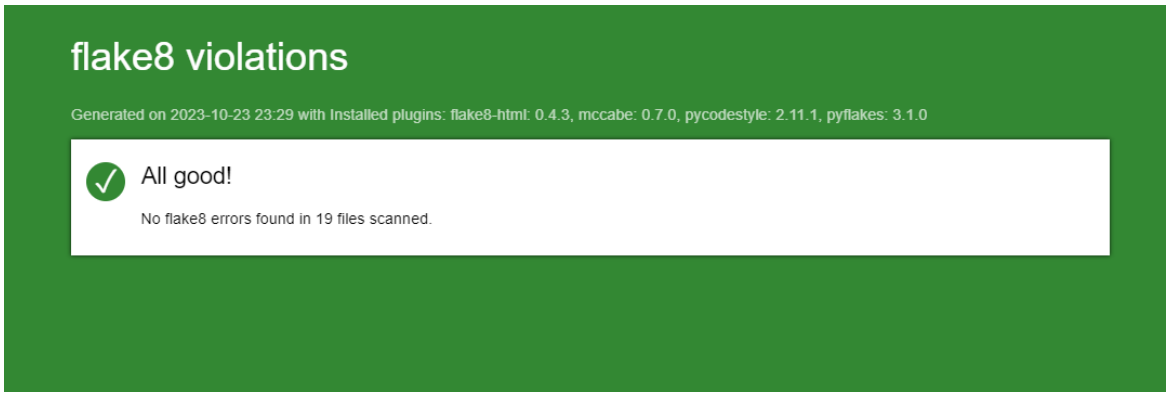

Abbildung 7: Statische Code-Analyse mit flake8

A.11 Pflichtenheft (Auszug)

Der folgende Abschnitt bietet einen Auszug aus dem Pflichtenheft und beschreibt die geplante Umsetzung der im Lastenheft definierten Anforderungen:

1. Entwicklungsumgebung und Plattform

- Als Entwicklungsumgebung wird VSCode verwendet.
- Die API wird mit Python3 unter Verwendung des Flask-Frameworks entwickelt.
- Es wird eine Schnittstelle implementiert, über die Verschmutzungsdaten erfasst, kategorisiert, bearbeitet und gelöscht werden können.
- Die Anwendung wird auf einem Laptop im firmeninternen Netzwerk deployed und dort getestet.
- Für die CI wird GitHub Actions eingesetzt.

2. Datenbank

- Die Daten werden in einer Postgres-Testdatenbank gespeichert.
- Sowohl die Datenbank als auch die API werden in Docker-Containern gehostet.

3. Oberfläche und Interaktivität

- Für die Entwicklung der App-Oberfläche wird das Ionic Framework auf Basis von Angular verwendet.
- Die Benutzeroberfläche wird für eine schnelle Erfassung von Verschmutzungen optimiert.
- Die Oberfläche wird automatisch gebaut und in eine native App umgewandelt.

4. Zugänglichkeit und Sicherheit

- Die App wird für Mobilgeräte optimiert.
- Es wird eine Authentifizierungsmethode implementiert.

5. Geschäftslogik, Tests und Wartung

- Für die Application Programming Interface (API) werden Unittests in Python durchgeführt.
- Ein regelmäßiges Update- und Wartungssystem wird eingerichtet.
- Ein Support-System wird bereitgestellt.

A.12 Listing von Python-Code

```python
class PollutionCount(Resource):
    parser = reqparse.RequestParser()
    parser.add_argument('count', type=int, required=True, help="Count field is required")

    @jwt_required()
    def put(self, pollution_id: str):
        data = PollutionCount.parser.parse_args()
        pollution = Pollution.query.get(pollution_id)

        if not pollution:
            return {"message": "Pollution not found"}, 404

        pollution.count = data['count']

        try:
            db.session.commit()  # Commit the changes to the database
            return {"message": "PollutionCount updated successfully"}, 200
        except Exception as e:
            db.session.rollback()  # Rollback changes in case of an error
            return {"message": f"An error occurred while updating pollution: {str(e)}"}, 500
```

Listing 1: Implementation der PollutionCount-Klasse

A.13 Listing von unittest

```
import unittest
from common.utils import Utils

class TestUtils(unittest.TestCase):
    def test_email_valid(self):
        self.assertTrue(Utils.email_is_valid("example@example.com"))
        self.assertFalse(Utils.email_is_valid("invalid_email"))

    def test_hash_password(self):
        password = "password123"
        hashed_password = Utils.hash_password(password)
        self.assertTrue(Utils.check_hashed_password(password, hashed_password))
        self.assertFalse(Utils.check_hashed_password("wrong_password",
    hashed_password))


if __name__ == '__main__':
    unittest.main()
```

Listing 2: Implementation der Utils-Tests

```
class TestAPIEndpoints(unittest.TestCase):
    def setUp(self):
        print("start setting up test environment...")

        warnings.filterwarnings("ignore")

        self.test_app = create_app(TestConfig, db)
        self.app = self.test_app.test_client()

        self.app_context = self.test_app.app_context()
        self.app_context.push()

        # Create the tables using the application context
        with self.test_app.app_context():
            db.create_all()

    def tearDown(self):
        # Drop the tables using the application context
        with self.test_app.app_context():
            print("tearDown")
            db.session.close_all()
            db.drop_all()

        self.app_context.pop()

    def test_update_pollution_count(self):
        geoarea = GeoArea(
            id=5,
            name='Area 5',
```

```
            datecreated='2023-01-01 00:00:00',
            language='German',
            last_update='2023-08-25 00:00:00',
            mandant='Mandant A',
            admincomment='Comment 1',
            automaticsearch=True,
            polygon="POLYGON((1 2,2 3, 3 4, 5 6, 1 2))"
        )

        pollution = Pollution(
            name='PollutionDelete',
            count=10,
            description='Description',
            geoarea_fk=5
        )
        db.session.add_all([pollution, geoarea])
        db.session.commit()

        expires = timedelta(days=7)
        access_token = create_access_token(identity=geoarea.id, expires_delta=expires)
        self.headers = {'Authorization': f'Bearer {access_token}'}

        with self.test_app.test_client() as client:
            response = client.put('/pollution/pollutionCount/' + pollution.id,
                                  json={'count': 15}, headers=self.headers)

            self.assertEqual(response.status_code, 200)

            updated_pollution = db.session.query(Pollution).get(pollution.id)
            self.assertEqual(updated_pollution.count, 15)
```

Listing 3: Implementation der Integrationstests