Hochschule Bremerhaven

Fachbereich II
Management und Informationssysteme
Wirtschaftsinformatik B.Sc.

Modul
Qualitätsmanagement

Semesteraufgabe

Entwicklung einer Hausverwaltung

Vorgelegt von:	Junior Lesage Ekane Njoh	MatNr. 40128
	Steve Aguiwo II	MatNr. 40088
	Franck Majeste Silatsa Dogmo	MatNr. 38555
Vorgelegt am:	11. März 2025	
Dozent:in:	Prof. Dr. Karin Vosseberg	

Inhaltsverzeichnis

1 Einleitung

Die Verwaltung von Gebäuden und deren Energieverbrauch stellt in der Praxis eine zentrale Herausforderung dar. Insbesondere in Mehrfamilienhäusern oder Wohnanlagen ist eine effiziente und übersichtliche Erfassung von Zählerständen erforderlich, um Verbrauchsdaten transparent zu machen und eine gerechte Abrechnung zu ermöglichen. Im Rahmen dieses Projekts entwickeln wir als Gruppe einen Prototyp für eine Hausverwaltungssoftware, die sich auf die digitale Erfassung, Verwaltung und Analyse von Zählerständen konzentriert.

Die Umsetzung erfolgt als webbasierte Anwendung mit einer intuitiven Benutzeroberfläche und einer zuverlässigen Datenverarbeitung. Unsere Hausverwaltung ermöglicht es, Gebäude, Zähler und Verbrauchsdaten zu verwalten, Zählerablesungen zu dokumentieren und historische Verbrauchswerte grafisch darzustellen. Dabei werden sowohl technische als auch organisatorische Aspekte berücksichtigt, um eine realitätsnahe und funktionale Lösung zu entwickeln.

Ein wesentlicher Bestandteil des Projekts ist das Review der Anforderungen sowie die Entwicklung eines fundierten Testkonzepts, um sicherzustellen, dass der Prototyp stabil, fehlerresistent und effizient arbeitet. Im Rahmen unserer Ausarbeitung dokumentieren wir die einzelnen Projektschritte detailliert und analysieren die gewonnenen Erkenntnisse. Unser Ziel ist es, ein praxisnahes und gut strukturiertes System zu entwerfen, das die wesentlichen Funktionen einer Hausverwaltung abbildet.

Die Entwicklung des Prototyps folgt einem iterativen Ansatz. Zu Beginn wurden die Anforderungen überprüft und überarbeitet, um Widersprüche oder Unklarheiten zu beseitigen. Anschließend wurden konkrete Testfälle definiert, um die Kernfunktionen zu validieren. Die Tests umfassen funktionale Prüfungen, negative Tests sowie Leistungstests, um sowohl die korrekte Funktionalität als auch die Systemgrenzen zu ermitteln. Schließlich wurde der Prototyp entsprechend der definierten Anforderungen und Testfälle umgesetzt und evaluiert.

Mit dieser Arbeit dokumentieren wir den gesamten Entwicklungsprozess, von der Anforderungsanalyse über die Testkonzeption bis hin zur Implementierung und Evaluation des Prototyps.

2 Anforderungsanalyse

2.1 Review der Anforderungen

Im Rahmen dieses Projekts haben wir ein `technisches Review` nach ISO 20246 durchgeführt. Diese Methode wurde gewählt, da sie eine frühe Fehlererkennung in der Anforderungsphase ermöglicht und sich besonders für dokumentenbasierte Analysen eignet.

Das Review-Team bestand aus allen drei Projektmitgliedern, die Analyse erfolgte in zwei Schritten:

1. **Individuelle Prüfung:** Jedes Teammitglied hat alleine für sich die Anforderungen unabhängig nach definierten Kriterien überprüft.
2. **Gemeinsame Konsolidierung**: In einer Sitzung wurden die identifizierten Probleme besprochen und Verbesserungsvorschläge erarbeitet.

Die Überprüfung erfolgte anhand folgender Kriterien:

- `Vollständigkeit:` Sind alle relevanten Aspekte der Hausverwaltung abgedeckt?
- `Eindeutigkeit:` Sind die Anforderungen so formuliert, dass keine Missverständnisse entstehen?
- `Widerspruchsfreiheit:` Gibt es logische oder inhaltliche Widersprüche?
- `Testbarkeit der Anforderungen:` Lassen sich die Anforderungen in konkrete Testfälle überführen?

Nach Überprüfung wurden alle 11 Anforderungen analysiert. Während einige Anforderungen lediglich präzisiert wurden, waren bei anderen inhaltliche Anpassungen erforderlich, um Unklarheiten zu beseitigen und die Testbarkeit zu gewährleisten. Von den überprüften 11 Anforderungen:

- 5 konnten unverändert übernommen werden,
- 3 wurden konkretisiert,
- 3 mussten inhaltlich angepasst werden `(z. B. neue Fehlermeldungen, Validierungsregeln)`.

Die vollständige Analyse mit konkreten Verbesserungsvorschlägen ist in folgender Tabelle dokumentiert:

Tabelle 2.1: Identifizierte Probleme und Verbesserungsvorschläge

Nr	Anforderung	Problem/ Unklarheit	Verbesserungsvorschlag
1	Gebäudestruktur (1…n Gebäude, Eingänge, Wohnungen, Zähler)	keine klare Definition von „Eingang“ Ist ein Eingang ein Gebäudeteil oder eine logische Struktur?	Definition eines Eingangs hinzufügen (z.B. „Ein Eingang ist eine physische oder logische Einheit, die Zugang zu Wohnungen ermöglicht.“).
2	verschiedene Zählertypen (Strom, Gas, Wasser)	Unklar, ob weitere Typen ergänzbar sind?	Klarstellung, ob die Liste erweiterbar ist und wie neue Zählertypen ergänzbar.
3	Zähler-ID	keine Vorgabe zur Länge oder zum Format der ID	Die Zähler-ID muss eindeutig sein und darf nicht mehrfach vergeben werden. Die ID wird automatisch nach dem Schema Gebäude-Jahr-Random generiert.
4	Datenfilterung	Unklar, welche Filtermöglichkeiten existieren (Gebäude, Zeitraum)?	Ergänzung von Filtern nach Gebäude, Wohnung, Zeitraum und Zählertyp.
5	Ablesewerte	Unklar, ob rückwirkende Korrekturen möglich sind.	Spezifikation: Ablesewerte können nur in der Zukunft oder am aktuellen Tag eingetragen werden. Änderungen nur durch Admins.
6	Zähler sind über ihre ID zu finden	Was passiert, wenn eine ID nicht existiert?	Falls eine Zähler-ID nicht existiert, erscheint die Fehlermeldung: ungültige Zählernummer. Bitte überprüfen Sie Ihre Eingabe
7	Zähler sollen abgelesen werden (Eingabe von Datum und Wert)	Gibt es eine Validierung für vergangene/future Daten?	Klarstellung, ob das Ablesedatum nur in der Vergangenheit oder auch in der Zukunft liegen darf.
8	Zähler und Datum laufen nur vorwärts	Fehlt eine Angabe zu Testfällen (z. B. wie rückdatierte Werte behandelt werden)	Testfälle für Grenzwerte (min/max Werte für Datum) spezifizieren
9	Weitere Ablese-informationen eingeben (Ablesung, Schätzung)	Müssen Nutzer einen Ablesetyp zwingend angeben oder gibt es Standardwerte?	Standardwert oder Pflichtfeld definieren.
10	Ableser-Informationen eingeben (Hauswart, Mieter, Energie-versorger)	Können mehrere Ableser für einen Zähler existieren?	Klärung, ob Mehrfachzuweisungen erlaubt sind.
11	Verbrauch berechnen und anzeigen	Sind historische Verbrauchswerte abrufbar?	Die Verbrauchsanzeige wird nach jeder neuen Ablesung automatisch aktualisiert. Keine manuelle Aktualisierung ist erforderlich. Historische Verbrauchsdaten werden für mindestens 12 Monate gespeichert.

2.2 Verbesserung der Anforderungen

Auf Basis unseres Reviews konnten wir die Anforderungen an das Hausverwaltungsprojekt verbessern. Dabei wurden unklare Definitionen konkretisiert, Testbarkeit verbessert und Validierungsregeln ergänzt.

Im Vergleich zu den ursprünglichen Anforderungen haben sich insbesondere die folgenden Aspekte geändert:

- Definition der Zähler-ID (eindeutig, 14-stellig, festes Format)
- Neue Fehlerbehandlungen für ungültige ID-Eingaben
- Validierungsregeln für vergangene und zukünftige Ablesewerte
- Optimierung der Verbrauchsanzeige mit Berücksichtigung fehlender Werte
- Skalierbarkeit für größere Datenmengen mit 5000+ Zählern

Nach Überlegung fanden wir es gut funktionalen von nicht funktionalen Anforderungen zu trennen. Die Trennung zwischen funktionalen und nicht-funktionalen Anforderungen ist essenziell, um eine klare Strukturierung der Systemanforderungen zu gewährleisten.
Unsere funktionalen Anforderungen definieren, was das System tun soll, also welche konkreten Funktionen es bereitstellt. Sie sind direkt testbar und beschreiben die Interaktionen zwischen Nutzern und System.
Nicht-funktionale Anforderungen hingegen spezifizieren wie das System diese Funktionen bereitstellen soll, also Qualitätsmerkmale wie Performance, Benutzerfreundlichkeit oder Skalierbarkeit. Durch diese Trennung wird es einfacher, sowohl die funktionale Umsetzung als auch die technischen Rahmenbedingungen des Prototyps gezielt zu überprüfen und zu optimieren.

2.2.1 Funktionale Anforderungen

Die folgende Tabelle enthält die funktionalen Anforderungen unseres Hausverwaltungsprototyps. Diese Anforderungen legen fest, welche Funktionen das System bieten muss, um eine effektive Verwaltung von Gebäuden, Zählern und Verbrauchsdaten zu ermöglichen. Dazu gehören unter anderem das Erfassen von Zählerständen, die Filterung von Daten sowie die Berechnung und Anzeige des Verbrauchs. Jede Anforderung ist so formuliert, dass sie klar verständlich und testbar ist.

Tabelle 2.2: Funktionale Anforderungen

Nr.	Anforderung	Beschreibung
F1	Gebäudestruktur verwalten	Gebäude können mehrere Eingänge haben, jede Wohnung hat eine eindeutige ID.
F2	Zählertypen verwalten	Unterstützte Typen: Strom, Gas, Wasser. Die Liste ist erweiterbar, indem neue Typen über eine Konfigurationsdatei durch Entwickler hinzugefügt werden.
F3	Zählerverwaltung	Jeder Zähler hat eine eindeutige ID im Format `Gebäude-Jahr-Random` (14-stellig). Jeder Zähler gehört zu einer Wohnung und einem Zählertyp. Er speichert den letzten Ablesewert, das letzte Ablesedatum und die Ablesemethode.
F4	Datenfilterung	Filter nach Gebäude, Wohnung, Zählertyp und Zeitraum.
F5	Zählerablesung	Zählerwerte können nur mit aktuellem oder zukünftigen Datum erfasst werden. Negative Werte sind nicht zulässig. Falls der neue Wert kleiner als der vorherige ist, gibt es eine Fehlermeldung. Admins können jedoch rückwirkende Korrekturen vornehmen, falls ein Fehler festgestellt wird.
F6	Fehlermeldungen	Falls eine Zähler-ID nicht existiert, erscheint „Die eingegebene ID existiert nicht“. Falls eine Wohnung keiner ID zugeordnet ist, erscheint „Dieser Zähler ist keiner Wohnung zugeordnet.“
F7	Verbrauchsanzeige	Historische Verbrauchswerte sind für die letzten 12 Monate abrufbar. Eine grafische Darstellung ist möglich.
F8	Ableser-Informationen	Ableser können Hauswart, Mieter oder Energieversorger sein. Falls keine Information vorhanden ist, wird „Unbekannt“ eingetragen.
F9	Bearbeiten und Löschen von Gebäuden	Gebäude können direkt bearbeitet oder gelöscht werden.
F10	Zurück-Buttons auf allen Seiten	Verbesserte Navigation in der Anwendung.
F11	Gebäude auswählen vor Verbrauchsanzeige	Nutzer müssen erst ein Gebäude wählen, bevor Verbrauchsdaten angezeigt werden.

Fortsetzung auf nächster Seite

Tabelle 2.2: Funktionale Anforderungen

F12	Direkte Weiterleitung bei nur einem Gebäude	Wenn nur ein Gebäude existiert, wird die Verbrauchsanzeige sofort geladen.
F13	Unterschiedliche Speicherung für aktuelle	historische Verbrauchsdaten: `verbrauch_aktuell_X.png` und `verbrauch_historie_X_YYYY-MM-DD.png` werden getrennt gespeichert.

2.2.2 Nicht-funktionale Anforderungen

Neben der funktionalen Umsetzung muss das System bestimmte nicht-funktionale Anforderungen erfüllen. Diese betreffen Aspekte wie Systemperformance, Skalierbarkeit, Fehlerbehandlung und Benutzerfreundlichkeit. Während funktionale Anforderungen definieren, „was“ das System tun soll, beschreiben nicht-funktionale Anforderungen, „wie gut“ es das tun muss. Besonders wichtig sind hier Antwortzeiten der Verbrauchsanzeige, die visuelle Darstellung der Verbrauchsdaten sowie Datenschutzaspekte im Umgang mit Zählerwerten.

Tabelle 2.3: Nicht-Funktionale Anforderungen

Nr.	Anforderung	Beschreibung
NF1	Zeitraum für die Verbrauchsanzeige im Diagramm sichtbar	Das Diagramm zeigt den Zeitraum der Messung an (z.B. „März 2024 - Februar 2025“)und wird automatisch aktualisiert, sobald neue Verbrauchsdaten eingegeben werden.
NF2	Letzte 12 Monate immer anzeigen (auch ohne Werte)	Die Verbrauchsanzeige berücksichtigt automatisch die letzten 12 Monate. Fehlende Werte werden als „0“ dargestellt.
NF3	Farbliche Kennzeichnung der Zähler in der Verbrauchsanzeige	Jeder Zähler erhält eine eindeutige Farbe zur besseren Unterscheidung.
NF4	Optimierung der Antwortzeiten	Das System soll Verbrauchsdaten in unter 2 Sekunden berechnen und anzeigen. Die Berechnung muss auch bei einer Last von 5000 Zählern stabil bleiben.
NF5	Datenintegrität und Konsistenz	Ablesewerte dürfen nicht rückwirkend geändert werden (außer durch Admins).
NF6	Speicherung von Verbrauchsdaten gemäß Datenschutzbestimmungen	Verbrauchsdaten dürfen nur von autorisierten Nutzern eingesehen werden.
NF7	System skalierbar für große Datenmengen	Unterstützung für mindestens 100 Gebäude und 5000 Zähler.

3 Testkonzept

Ein strukturiertes Testkonzept ist essenziell, um die Qualität und Stabilität der entwickelten Hausverwaltungssoftware sicherzustellen. Da es sich um einen Prototype handelt, fokussieren wir uns auf technische Tests zur Funktionsprüfung und verzichten auf umfassende Usability- oder Systemtests. Unser Ziel ist es, sicherzustellen, dass die Kernfunktionen fehlerfrei funktionieren, Daten korrekt verarbeitet werden und das System auch unter Last stabil bleibt. Die Tests orientieren sich an etablierten Softwaretestverfahren und wurden so konzipiert, dass sie eine möglichst hohe Abdeckung der Anforderungen gewährleisten.

3.1 Testziele und Strategie

Für unser Testkonzept haben wir uns klare Testziele definiert, damit wir den Fokus behalten können und nicht unnötig viele Szenarien testen müssen. Unsere Hauptziele sind:

- Sicherstellen, dass die Kernfunktionen der Software korrekt arbeiten.
- Überprüfung, ob Module und Komponenten korrekt zusammenarbeiten.
- Validierung der Fehlerbehandlung durch gezielte Eingabe ungültiger Werte.
- Sicherstellung der Performance unter hoher Last.

Unser Testkonzept folgt einer auf `bottom-up-ansatz`basierten Strategie, bei der zunächst einzelne Komponente geprüft und anschließend System- und Performancetests durchgeführt werden:

- **Zunächst Unit-Tests**: Isolierte Tests einzelner Funktionen (z.B. Validierung von Zähler-IDs oder Verbrauchsdaten).
- **Danach Funktionstests**: Prüfung der Geschäftslogik, u.a. Zählerverwaltung, Ablesungen und Verbrauchsberechnung.
- **Anschließend negative Tests**: Überprüfung der Fehlerbehandlung durch ungültige Eingaben.
- **Schließlich Performance-Tests**: Simulation von hoher Last, um die Skalierbarkeit zu überprüfen.

Wir wissen aus der Vorlesung, dass sich Softwaretests grundsätzlich in drei Kategorien unterteilen lassen:

- **White-Box-Testing:** Interne Logik der Software wird geprüft, Code-Abdeckung ist entscheidend.

- **Black-Box-Testing:** Tests erfolgen ohne Kenntnis des Quellcodes, Fokus liegt auf den Ein- und Ausgaben des Systems.
- **Gray-Box-Testing:** Kombination aus White-Box und Black-Box, teilweise Kenntnisse über den Code werden verwendet.

Aus diesem Grund haben wir uns eine Kombination aus **Black-Box-Testing** und **Gray-Box-Testing** entschieden, um die Funktionalität aus Nutzersicht zu prüfen und gezielt Fehlerfälle im Code zu analysieren. Wir setzen auch die **Äquivalenzklassenbildung** ein, um Eingaben in Gruppen zu testen und mit wenigen Tests eine hohe Abdeckung zu erreichen. Enorm hat uns auch **Boundary-Value-Testing** geholfen, Grenzwerte (z.B. minimale und maximale Verbrauchswerte) gezielt zu überprüfen.

Unsere Entscheidung, uns auf diese Techniken zu konzentrieren, basiert auf der Tatsache, dass unser Prototyp eine Webanwendung ist, wobei der Fokus auf der Funktionalität der Schnittstellen und Datenverarbeitung liegt. Die Äquivalenzklassenbildung reduziert die Anzahl der Testfälle, während trotzdem eine breite Abdeckung erreicht wird. Letztendlich ist Boundary-value-Testing essenziell für die Überprüfung von Verbrauchswerten und Ablesedaten.

3.2 Testarten

Um unser System effizient zu testen, haben wir vier Teststufen definiert:

Tabelle 3.1: Testarten

Testverfahren	Ziel	Begründung
Unit-Tests	Einzelne Funktionen wie Datenvalidierung, ID-Format, Speicherung von Ablesewerten	Frühes Erkennen von Fehlern in einzelnen Modulen.
Funktionstests	Überprüfung der gesamten Funktionalität wie Zählerverwaltung, Ablesungen, Filterung, Verbrauchsanzeige	Sicherstellung der korrekten Umsetzung der Anforderungen.
Performance-Tests	Simulation hoher Last durch 1000+ gleichzeitige Ablesungen	Sicherstellung, dass das System auch mit vielen Gebäuden und Zählern performant bleibt.
Negative Tests	Eingabe ungültiger Werte (z.B. leere Felder, falsche ID, negatives Datum)	Überprüfung der Fehlerbehandlung und Robustheit des Systems.

3.3 Testumgebung und Testdaten

3.3.1 Testumgebung

Unsere Testumgebung haben wir versucht, so einfach wie möglich zu halten, damit wir nicht den Rahmen überspringen. Hier sind die Kernpunkte unserer Testumgebung aufgelistet:

- Der Prototyp wird in einer lokalen Entwicklungsumgebung als Flask-Anwendung entwickelt und getestet.
- Die Tests werden mithilfe von pytest automatisiert durchgeführt.
- Performance-Tests erfolgen durch Simulation hoher Anfragen über eine Flask-Testumgebung.
- Erstellung von Testfällen erfolgt nach den Prinzipien von Äquivalenzklassenbildung und Grenzwertanalyse.

3.3.2 Testdaten

Zur Testen gehören auch Testdaten zur Simulation, weil wir noch bei einem Prototyp sind, dessen Einsatz in einer produktiven Umgebung geplant ist.

- Eine Testdatenbank mit Dummy-Daten wird verwendet.
- Persistenz der Daten erfolgt über JSON-Dateien.
- Für Lasttests werden 1000 simultane Ablesungen simuliert.
- Testfälle für Grenzwerte und ungültige Werte (z.B. negative Ablesungen) wurden vorbereitet.

3.4 Ableitung konkreter Testfällen

Die gründliche Kontrolle der Funktionen ist entscheidend für die Qualität eines Softwareprojekts: Sie bestimmt, ob das Projekt erfolgreich ist oder nicht. Daher haben wir aus unseren Anforderungen gezielt abgeleitet, um sicherzustellen, dass der Prototyp alle Anforderungen erfüllt und fehlerfrei arbeitet. Wir haben wie oben schon aufgelistet, vier Testarten gewählt

Tabelle 3.2: Testfälle für die Hausverwaltungssoftware

Test-ID	Beschreibung	Eingabe	Erwartetes Ergebnis	Testtyp
TC-F6-01	Zähler-ID existiert nicht	‘999-9999-9999‘	Fehlermeldung: „Die eingegebene ID existiert nicht.“	Negative Test
TC-F5-02	Negativer Ablesewert	‘-10‘	Fehlermeldung: „Ungültiger Ablesewert“	Negative Test
TC-F3-03	Zählerlänge	‘1-2024-4567823‘	Fehlermeldung: „Zähler-ID muss genau 14 Zeichen haben!“	Negative Test
TC-F5-04	Ablesedatum rückdatiert	‘2000-01-01‘	Fehlermeldung: „Datum darf nicht in der Vergangenheit liegen!“	Negative Test
TC-F5-05	Ablesewert kleiner als vorheriger Wert	Neuer Wert: ‘50‘, alter Wert: ‘100‘	Fehlermeldung: „Neuer Wert muss größer sein als der vorherige.“	Negative Test
TC-F3-06	Gültige Zähler-ID über die Suchfunktion eingeben	‘1-2025-5487‘	Zählerdetails werden angezeigt	Funktionstest
TC-F5-07	Korrekte Ablesung speichern	Alter Wert: 100, Neuer Wert: ‘250‘	Wert wird korrekt gespeichert	Funktionstest
TC-F5-08	Ablesedatum in der Zukunft	Datum: ‘01.01.2030‘	Wert wird gespeichert	Funktionstest
TC-F8-09	Standard-Ableser bei fehlender Eingabe	Ableser nicht eingetragen	Standardwert „Unbekannt“ wird gespeichert	Funktionstest
TC-F7-10	Historische Verbrauchswerte anzeigen	Es wird die Schnittstelle für Historiographer mit der Gebäude-ID ‘1′ abgerufen	Ablesungen sollen als Liste zurückgegeben werden oder als Grafik in der Weboberfläche	Funktionstest
TC-F3-11	Suchfunktion mit Teilstring	Eingabe: ‘123‘	Zeigt alle Zähler mit ‘123‘ in der ID	Funktionstests

Fortsetzung auf nächster Seite

Tabelle 3.2: Testfälle für die Hausverwaltungssoftware

TC-NF4-12	Massive Ablesungen	10000 Ablesungen	Es werden alle Ablesungen in maximal 60 Sekunden gespeichert und keine Daten gehen verloren	Performance-Tests
TC-NF4-13	Antwortzeit-Test	Es wird die index-Seite aufgerufen	Innerhalb von wenigen Millisekunden eine Antwort geliefert	Performance-Test
TC-NF7-14	Massive Zählererstellung	10000 Strom-Zähler	Alle Zähler werden hinzugefügt ohne zu lange Wartezeit	Performance-Test
TC-NF7-15	Massive Gebäude erstellen	10000 Gebäude	Alle Gebäude werden hinzugefügt ohne zu lange Wartezeit	Performance-Test
TC-F1-16	Datenspeicherung und Datenabruf im JSON-Format	Dummy-Gebäude-Daten	Gebäude-Daten sollen gespeichert und abgerufen werden können.	Unit-Test
TC-F3-17	Zähler-ID-Generierung	7 als Gebäude-ID und das aktuelle Jahr	Es soll eine gültige Zähler-ID generiert werden.	Unit-Test

Unser Testkonzept ist also sehr umfassend, deckt die meisten möglichen Szenarien ab und stellt sicher, dass der Prototyp funktionsfähig, robust und performant ist. Durch die gewählte Kombination aus unseren Testarten wollen wir eine praxisnahe Qualitätssicherung erreichen.

4 Prototypische Umsetzung der Hausverwaltung

4.1 Software-Architektur und Technologien

Die prototypische Umsetzung der Hausverwaltung basiert auf einer webbasierten Client-Server-Architektur, bei der das Backend die Geschäftslogik verwaltet und das Frontend die Benutzeroberfläche bereitstellt.

Diese Architektur ermöglicht eine klare Trennung von Datenverarbeitung und Präsentation, wodurch die Wartbarkeit und Skalierbarkeit der Anwendung verbessert wird.

4.1.1 Verwendete Technologien

Für die Umsetzung des Prototyps haben wir bewusst Technologien gewählt, die eine einfache Entwicklung, Testbarkeit und Skalierbarkeit unterstützen. Die folgende Tabelle gibt einen Überblick über die eingesetzten Technologien:

Tabelle 4.1: Verwendete Technologien

Technologie	**Einsatzbereich**	**Begründung**
Python 3	Backend-Logik, API	Einfache Entwicklung und große Auswahl an Bibliotheken für Web- und Datenverarbeitung
Flask	Web-Framework für das Backend	Leichtgewichtiges Framework für die schnelle Entwicklung von Webanwendungen
HTML, CSS, Jinja2	Frontend und Templating	Ermöglicht dynamische Weboberflächen mit serverseitigem Rendering
JSON	Datenspeicherung	Einfache persistente Speicherung von Gebäuden, Zählern und Ablesewerten
Pytest	Testautomatisierung	Framework zur strukturierten Implementierung und Ausführung von Unit- und Funktionstests
Matplotlib	Visualisierung von Verbrauchsdaten	Erstellung von Diagrammen zur Verbrauchsanalyse

4.1.2 Architekturübersicht

Unsere Anwendung folgt dem Model-View-Controller (MVC)-Ansatz, um eine strukturierte Trennung zwischen Daten, Logik und Darstellung zu gewährleisten:

- **Model (M):** Datenverwaltung erfolgt über JSON-Dateien, die Gebäude-, Zähler- und Ablesedaten speichern.
- **View (V):** Die Weboberfläche wird über HTML und Jinja2-Templates dynamisch generiert.
- **Controller (C):** Die Geschäftslogik ist in Flask implementiert und verarbeitet Nutzeranfragen.

Die folgende Abbildung zeigt die grundlegende Architektur unserer Hausverwaltungssoftware:

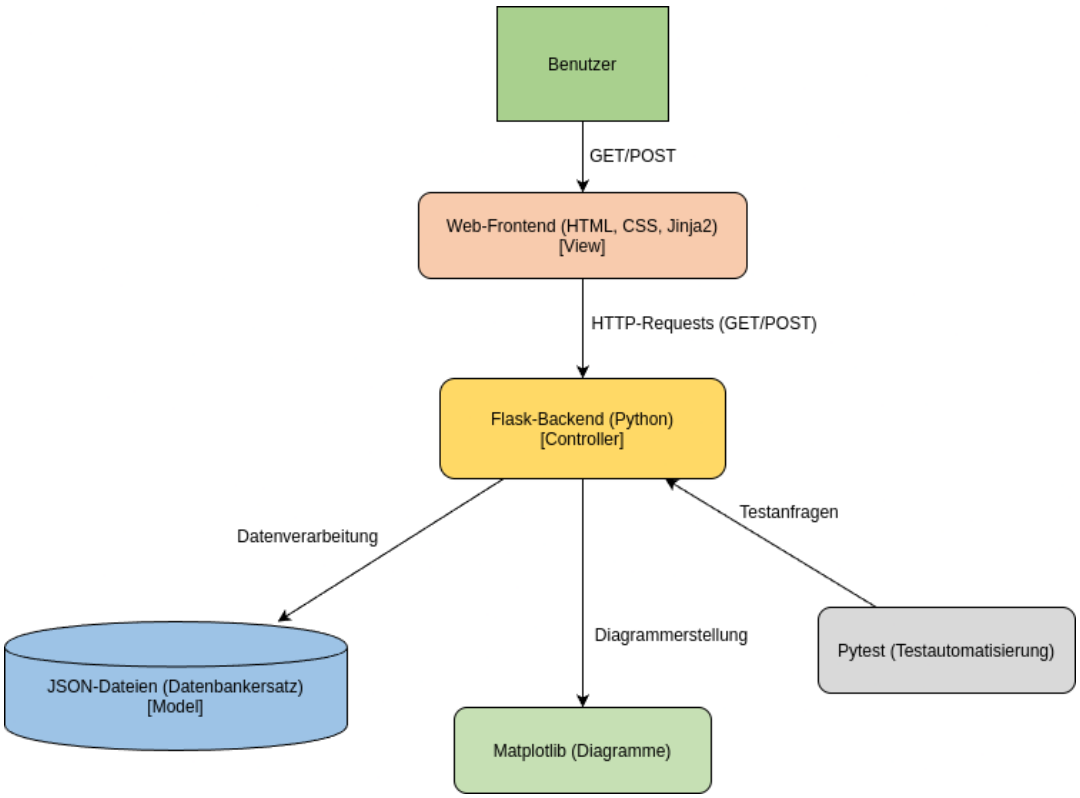

Abbildung 4.1: Architektur der Hausverwaltungssoftware

Unsere Architektur ist speziell für einen Prototyp konzipiert, kann aber mit minimalem Aufwand für eine produktive Umgebung weiterentwickelt werden.

4.2 Implementierung

Unser Prototyp haben wir modular aufgebaut und er besteht aus den folgenden Kernkomponenten:

Tabelle 4.2: Kernkomponente

Modul	Beschreibung
Flask-Backend	Verantwortlich für die Geschäftslogik der Anwendung, einschließlich der Verarbeitung von Ablesedaten und der Verwaltung der JSON-Datenbank.
Web-Frontend	HTML/CSS und Jinja2 für die Benutzeroberfläche zur Verwaltung von Gebäuden, Zählern und Ablesewerten.
Datenverwaltung	Speicherung und Abruf von Daten in JSON-Dateien (gebäude.json, zähler.json, ablesungen.json).
Ablesungshistorie	Verwaltung und Validierung der Ablesungen sowie Berechnung von Verbrauchswerten.
Diagrammerstellung	Generierung von Verbrauchsdiagrammen mit Matplotlib zur Visualisierung der historischen Daten.
Testautomatisierung	Automatische Tests mit Pytest zur Validierung der wichtigsten Funktionen (Unit-Tests, Funktionstests, Performance-Tests).

4.2.1 Wichtige Funktionen und Codeausschnitte

Unsere Geschäftslogik besteht aus mehreren Funktionen, die wir hier nicht alle auflisten können, daher haben wir uns entschieden nur die wichtigsten zu zeigen.

Funktion: `ablesung_hinzufuegen()`

Die Funktion verarbeitet die eingehende Ablesedaten, validiert sie und speichert sie in der JSON-Datenbank. Es erfolgt zuerst eine Überprüfung, ob die erforderlichen Werte vorhanden und gültig sind.
Danach wird geprüft, ob der Zähler zur richtigen gebäude-ID gehört, wenn ja dann erfolgt eine Validierung, ob der neue Ablesewert nicht kleiner als der vorherige ist. Sind alle Bedingungen erfüllt, wird die Ablesung gespeichert und eine Erfolgsmeldung zurückgegeben.

```
1 @app.route("/ablesung/hinzufuegen", methods=["POST"])
2     def ablesung_hinzufuegen():
3     ablesungen = load_json(ABLESUNG_FILE)
4     zaehler = load_json(ZAEHLER_FILE)
5     gebaeude = load_json(GEBAEUDE_FILE)
6
```

```
    # JSON-Daten aus der Anfrage abrufen
    data = request.get_json()
    if not data:
        return jsonify({"error": "Fehlende oder ungültige JSON-Daten"}), 400

    print("Empfangene JSON-Daten:", data)

    try:
        gebaeude_id = data.get("gebaeude_id")
        zaehler_id = data.get("zaehler_id")
        datum = data.get("datum")
        wert = int(data.get("wert"))
        ableser = data.get("ableser", "Unbekannt")

        if not gebaeude_id or not zaehler_id or not datum or wert is None:
            return jsonify({"error": "Fehlende Eingaben"}), 400

        heutiges_datum = datetime.now().date()
        eingabe_datum = datetime.strptime(datum, "%Y-%m-%d").date()

        print(heutiges_datum)

        if eingabe_datum < heutiges_datum:
            return jsonify({"error": "Datum darf nicht in der Vergangenheit
            ↪  liegen!"}), 400


    except ValueError:
        return jsonify({"error": "Ungültiger Zahlenwert für Ablesung"}), 400

    # Prüfen, ob der gewählte Zähler wirklich zu diesem Gebäude gehört
    if not any(z["id"] == zaehler_id and str(z["gebaeude_id"]) == str(gebaeude_id)
    ↪  for z in zaehler):
        return jsonify({"error": "Ungueltiger Zaehler fuer dieses Gebaeude!"}), 400

    # Validierung des Ablesewerts
    if wert < 0:
        return jsonify({"error": "Ungueltiger Ablesewert"}), 400

    # Überprüfung auf vorherige Ablesewerte
    vorherige_ablesungen = [a for a in ablesungen if a["zaehler_id"] == zaehler_id]
```

```
    if vorherige_ablesungen:
        letzter_wert = max(a["wert"] for a in vorherige_ablesungen)
        if wert < letzter_wert:
            return jsonify({"error": "Neuer Ablesewert muss groesser sein als der
            ↪  vorherige"}), 400

    # Ablesung speichern
    neue_ablesung = {
        "gebaeude_id": gebaeude_id,
        "zaehler_id": zaehler_id,
        "datum": datum,
        "wert": wert,
        "ableser": ableser
    }
    ablesungen.append(neue_ablesung)
    save_json(ABLESUNG_FILE, ablesungen)

    return jsonify({"message": "Ablesung erfolgreich gespeichert", "ableser":
    ↪  ableser}), 201
```

Listing 4.1: Python Code *ablesung_hinzufuegen*

Funktion: `verbrauchsanzeige()`

Diese Funktion berechnet den Verbrauch und generiert eine grafische Darstellung für den User. Die Funktionsweise haben wir so einfach wie möglich gehalten. Es werden zuerst die Verbrauchsdaten aus der JSON-Datei abgerufen und basierend auf der Gebäude-ID gefiltert dann wird ein Diagramm
mit Matplotlib zur Visualisierung des Verbrauchs erstellt. Anschließend wird das Diagramm gespeichert und auf der Weboberfläche angezeigt.

```
@app.route("/verbrauch", methods=["GET"])
    def verbrauchsanzeige():
    ablesungen = load_json(ABLESUNG_FILE)
    gebaeude = load_json(GEBAEUDE_FILE)
    selected_gebaeude = request.args.get("gebaeude_id")

    if not selected_gebaeude:
        return render_template("verbrauch.html", gebaeude=gebaeude,
        ↪  selected_gebaeude=None, no_data=True)

    try:
        selected_gebaeude = int(selected_gebaeude)
    except ValueError:
        return "Fehler: Ungültige Gebäude-ID!", 400

    # Verbrauchsdaten filtern
    ablesungen = [a for a in ablesungen if str(a["gebaeude_id"]) ==
    ↪  str(selected_gebaeude)]
    if not ablesungen:
        return render_template("verbrauch.html", gebaeude=gebaeude,
        ↪  selected_gebaeude=selected_gebaeude, no_data=True)

    # Diagramm generieren
    plt.figure(figsize=(10, 5))
    for zaehler_id in set(a["zaehler_id"] for a in ablesungen):
        daten = sorted([a for a in ablesungen if a["zaehler_id"] == zaehler_id],
        ↪  key=lambda x: x["datum"])
        x = [datetime.strptime(a["datum"], "%Y-%m-%d") for a in daten]
        y = [a["wert"] for a in daten]
        plt.plot(x, y, marker="o", linestyle="-", label=f"Zähler {zaehler_id}")

    plt.xlabel("Datum")
    plt.ylabel("Verbrauch")
```

```
30      plt.legend()
31      plt.grid(True)
32
33      save_path = os.path.join("static", f"verbrauch_{selected_gebaeude}.png")
34      plt.savefig(save_path)
35      plt.close()
36
37      return render_template("verbrauch.html", gebaeude=gebaeude,
        ↪  selected_gebaeude=selected_gebaeude, verbrauchspfad=save_path)
```

Listing 4.2: Python Code *Verbrauchsanzeige*

4.2.2 Herausforderungen während der Umsetzung

Während der entwicklung unseres Prototyps sind wir auf mehrere Herausforderungen gestoßen, die wir mit verschiedenen Lösungsansätzen bewältigt haben:

Tabelle 4.3: Herausforderungen

Herausforderung	**Lösung**
Validierung der Ablesedaten	Implementierung von Prüfungen für ID-Format, Wertebereiche und Duplikate
Simulation von Lasttests	Nutzung von Pytest, um 1000+ Ablesungen gleichzeitig zu simulieren
Darstellung der Verbrauchsdaten	Speicherung und Abruf von Daten in JSON-Dateien (gebäude.json, zähler.json, ablesungen.json)
Fehlermeldungen und UI-Feedback	Klare Fehlermeldungen und strukturierte JSON-Antworten implementiert

Durch die Erfahrung, die wir bei der Umsetzung vom Prototyp gemacht haben, können wir bereits sagen, dass unsere JSON-basierte Datenverwaltung für diesen Fall ausreicht, jedoch für eine produktive Umgebung wäre eine Umstellung auf eine relationale Datenbank sinnvoll.

4.3 Anwendung des Testkonzepts

4.3.1 Überblick über die Testergebnisse

In diesem Abschnitt werden die durchgeführten Tests dokumentiert. Dabei wurden verschiedene Testarten angewandt, um die Funktionsweise und Stabilität des Systems zu validieren. Die Testergebnisse basieren auf einer Kombination aus automatisierten Tests mit pytest und manuellen Überprüfungen in der Weboberfläche.

Tabelle 4.4: Testfälle für die Hausverwaltungssoftware

Test-ID	**Eingabe**	**Erwartetes Ergebnis**	**Tatsächliches Ergebnis**	**Status**
TC-F6-01	‘999-9999-9999‘	Fehlermeldung: „Die eingegebene ID existiert nicht.“	Fehlermeldung wurde korrekt ausgegeben	Bestanden
TC-F5-02	‘-10‘	Fehlermeldung: „Ungültiger Ablesewert“	Fehlermeldung wurde korrekt ausgegeben	Bestanden
TC-F3-03	‘1-2024-4567823‘	Fehlermeldung: „Zähler-ID muss genau 14 Zeichen haben!“	Fehlermeldung wurde korrekt ausgegeben	Bestanden
TC-F5-04	‘2000-01-01‘	Fehlermeldung: „Datum darf nicht in der Vergangenheit liegen!“	Fehlermeldung wurde korrekt ausgegeben	Bestanden
TC-F5-05	Neuer Wert: ‘50‘, alter Wert: ‘100‘	Fehlermeldung: „Neuer Wert muss größer sein als der vorherige.“	Fehlermeldung wurde korrekt ausgegeben	Bestanden
TC-F3-06	‘1-2025-5487‘	Zählerdetails werden angezeigt	Zählerdetails wurden angezeigt	Bestanden
TC-F5-07	Alter Wert: 100, Neuer Wert: ‘250‘	Wert wird korrekt gespeichert	Die Ablesewerte wurden gemäß den Spezifikationen korrekt gespeichert	Bestanden

Fortsetzung auf nächster Seite

Tabelle 4.4: Testfälle für die Hausverwaltungssoftware

TC-F5-08	Datum: ‘01.01.2030‘	Wert wird gespeichert	Die Ablesewerte wurden gemäß den Spezifikationen korrekt gespeichert	Bestanden
TC-F8-09	Ableser nicht eingetragen	Standardwert „Unbekannt“ wird gespeichert	Die Ablesewerte wurden gemäß den Spezifikationen korrekt gespeichert	Bestanden
TC-F7-10	Es wird die Schnittstelle für Verbauchshistorie mit der Gebäude-ID ”1äbgerufen	Ablesungen sollen als Liste zurückgegeben werden oder als Grafik in der Weboberfläche	Ablesungen wurden als List über die API und als Grafik über die Weboberfläche zurückgegeben	Bestanden
TC-F3-11	Eingabe: ‘123‘	Zeigt alle Zähler mit ‘123‘ in der ID	Alle Zähler beinhaltend ”123”wurden angezeigt	Bestanden
TC-NF4-12	10000 Ablesungen	Es werden alle Ablesungen in maximal 60 Sekunden gespeichert und keine Daten gehen verloren	Es sind keine Daten verloren gegangen	Bestanden
TC-NF4-13	Es wird die index-Seite aufgerufen	Innerhalb von wenigen Millisekunden eine Antwort geliefert	Index-Seite wurde innerhalb von 0.2 Millisekunden angezeigt	Bestanden

Fortsetzung auf nächster Seite

Tabelle 4.4: Testfälle für die Hausverwaltungssoftware

TC-NF7-14	10000 Strom-Zähler	Alle Zähler werden hinzugefügt ohne zu lange Wartezeit	Die erwartete Antwortzeit sollte unter 1 Sekunde bleiben, tatsächlich lag sie bei durchschnittlich 0.7 Sekunden, was innerhalb der akzeptablen Grenze liegt. Die Testdaten wurden vollständig und korrekt gespeichert	Bestanden
TC-NF7-15	10000 Gebäude	Alle Gebäude werden hinzugefügt ohne zu lange Wartezeit	Die erwartete Antwortzeit sollte unter 1 Sekunde bleiben, tatsächlich lag sie bei durchschnittlich 0.8 Sekunden, was innerhalb der akzeptablen Grenze liegt. Die Testdaten wurden vollständig und korrekt gespeichert	Bestanden
TC-F1-16	Dummy-Gebäude-Daten	Gebäude-Daten sollen gespeichert und abgerufen werden können.	Daten konnten korrekt gespeichert und wieder ausgelesen werden	Bestanden
TC-F3-17	7 als Gebäude-ID und das aktuelle Jahr	Es soll eine gültige Zähler-ID generiert werden.	ID wurde korrekt generiert	Bestanden

4.3.2 Analyse der Testergebnisse

Die Testergebnisse zeigen, dass der Prototyp die definierten Anforderungen weitestgehend erfüllt. Tests haben verschiedene Erkenntnisse geliefert, die für zukünftige optimierungen genutzt werden können. Die hohe Erfolgsquote zeigt, dass der Prototyp stabil und zuverlässig

arbeitet. Insbesondere die korrekte Verarbeitung von Zähler-IDs und Verbrauchswerten konnte nachgewiesen werden.

Erfolgreiche Tests

- Alle Unit-Tests wurden bestanden, was zeigt, dass die Kernfunktionen (z.B.: ID-Generierung, Datenspeicherung) korrekt arbeiten.
- alle Funktionstests sind erfolgreich, sodass die Hausverwaltung ihre Grundfunktionen fehlerfrei ausführt.
- Alle negative Tests wurden auch erfolgreich ausgeführt, was zeigt, dass der Prototyp keine unzulässigen Eingabe akzeptiert.
- Performance-Tests bestätigen, dass das System stabil mit großen Datenmengen umgehen kann. IDes könnte nützlich sein, wenn der Prototyp in einer realen Umgebung für eine große Verwaltungsfirma eingesetzt werden soll.

Verbesserungsbedarf Allerdings ist uns auch beim Testen einiges aufgefallen und zwar gibt es möglicherweise auch Verbesserungsbedarf, was mana an unserem Prototyp kritisieren könnte. Da könnte man auf die folgende Punkte eingehen:

- Bei einem versuchten Massentest von 1.000.000 gab es 7 fehlerhafte Einträge, da Zähler-IDs nicht doppelt existieren dürfen. Hierfür haben wir uns als Lösung ausgedacht, dass wir eine bessere ID-Prüfung vor dem Speichern einführen könnten.
- Die Verarbeitungsgeschwindigkeit war insgesamt gut, aber für noch größere Datenmengen könnte eine optimierte Datenbankstruktur erforderlich sein. Darüber hinaus könnte man mit dem Punkt Sicherheit unsere Datenpersistenz kritisieren, da die Daten ungeschützt gespeichert werden. Unsere Lösung hierfür wäre eine Umstellung auf eine sicherere Datenbank wie etwas MariaDb für eine produktive Umgebung.

Angesichts dieser Analyse ist festzustellen, dass unser Prototyp die wichtigsten anforderungen erfüllt und eine stabile Grundlage für eine erweiterte Version bietet. Allerdings wäre für eine produktive Umgebung der Wechsel von JSON zu einer relationalen Datenbank sinnvoll, um eine effizientere Abfrage und bessere Skalierbarkeit zu gewährleisten.

5 Qualitätsmanagement-Methoden in der Softwareentwicklung

Aufgrund der zentralen Rolle, was Qualitätssicherung in der Softwareentwicklung spielt, sollten wir uns diese Gelegenheit nicht entgehen lassen, eine reflektierende und wissenschaftliche Betrachtung der angewendeten QS-Methoden in unserem Projekt zu liefern, daher sollten wir auch in diesem Zusammenhang die Bedeutung von Qualitätssicherung herausarbeiten und dann konkret auf die Anwendung von QS-Methoden in unserem Projekt eingehen.

5.1 Relevanz der Qualitätssicherung

In der Softwareentwicklung spielt die Qualitätssicherung (QS) eine entscheidende Rolle, um sicherzustellen, dass Anwendungen fehlerfrei funktionieren, effizient arbeiten und benutzerfreundlich sind.

Mangelhafte QS kann zu Fehlfunktionen, Sicherheitslücken und höheren Kosten führen, da nachträgliche Fehlerkorrekturen aufwendiger sind als eine frühzeitige Fehlervermeidung.

Gerade in unserem Projekt, einem Hausverwaltungssystem, ist eine präzise Qualitätssicherung unerlässlich. Die Anwendung muss zuverlässig arbeiten, um sicherzustellen, dass Verbrauchsdaten korrekt verarbeitet und angezeigt werden, damit eine fehlerfreie Abrechnung möglich ist. Außerdem müssen Fehlermeldung konsistenz und verständlich sein, damit Nutzer wissen, was schiefgelaufen ist und wie sie das Problem beheben können. Zudem soll die Anwendung auch bei hoher Nutzerlast stabil bleiben, insbesondere wenn viele Gebäude und Zähler verwaltet werden müssen. Um diese Risiken zu minimieren, haben wir gezielt QS-Maßnahmen in unser Projekt integriert.

Unsere wichtigsten Maßnahmen zur Sicherstellung der Softwarequalität sind:

- Ein Anforderung-Review nach **ISO 20246**, um Unklarheiten frühzeitig zu identifizieren und widersprüchliche Anforderungen zu vermeiden.
- Ein strukturiertes Testkonzept, das verschiedene Testarten wie Unit-Tests, Funktionstests, Negative Tests und Performance-Tests umfasst.
- Traceability zwischen Anforderungen und Testfällen, um sicherzustellen, dass jede relevanten Anforderungen auch tatsächlich getestet wird.

5.2 Anwendung von QS-Methoden im Projekt

Die Qualitätssicherung unseres Projekts erfolgte in mehreren Phasen, um eine hohe Softwarequalität und Stabilität sicherzustellen. Dabei haben wir gezielt Maßnahmen ergriffen, die Fehler frühzeitig identifizieren und die Nachverfolgbarkeit der Anforderungen verbessern.

Ein zentraler Bestandteil unserer Qualitätssicherung war die direkte Verknüpfung der Anforderungen mit den Testfällen. Dadurch konnten wir sicherstellen, dass jede implementierte Funktion einer definierten Anforderung entspricht und getestet wird. Alle Testfälle wurden aus den überarbeiteten Anforderungen abgeleitet und unser technisches Review half uns, Anforderungen von der Implementierung zu präzisieren und spätere Fehler zu vermeiden.

Anforderungen sind ein wichtiger Punkt als Basis für die Implementierung vom Prototyp, dennoch sollten wir auch nicht vergessen, dass es noch wichtiger ist, die Prozesse zu dokumentieren, besonders die Tests. Eine strukturierte Testdokumentation war essenziell, um die Testergebnisse systematisch zu erfassen und nachvollziehbar zu machen. Die Testergebnisse haben wir in einer Tabelle festgehalten, sodass jede Test-ID direkt mit einer Anforderung verknüpft ist. Außerdem haben wir die fehlerhaften Tests analysiert, was zu gezielten Verbesserungen im Code führte. Durch diese automatisierte Nachverfolgbarkeit konnten wir effizient Schwachstellen identifizieren und beheben, wodurch sich die Qualität des Prototyps erheblich verbesserte.

5.3 Lessons Learned

Wir können es nicht in Abrede stellen, dass uns die Anwendung von Qualitätssicherungsmethoden wertvolle Erkenntnisse gebracht hat, die über den reinen Testprozess hinausgehen. Wie wir diese Qualitätsmaßnahmen konkret in unserem Projekt umgesetzt haben, wird im Folgenden erläutert:

- **Früheres Review der Anforderungen hilft, Fehler zu vermeiden:** Unklare oder fehlerhafte Anforderungen führen zu einem erhöhten Testaufwand und Nachbesserungen. Durch unser strukturiertes Anforderung-Review konnten wir dies minimieren und von Anfang an präzisere Anforderungen definieren.
- **automatisierte Tests sparen Zeit und erhöhen die Testabdeckung:** Manuelle Tests sind zeitaufwendig und fehleranfällig beispielsweise durch Konfigurationsfehler oder Eingabefehler. Mit `pytext` konnten wir wiederholbare Tests automatisieren und schneller fehler identifizieren.
- **Performance-Tests sind essenziell für die Skalierbarkeit:** Unsere Lasttests zeigten, dass das System stabil mit großen Datenmengen umgehen kann. Allerdings haben wir erkannt, dass für eine echte Skalierbarkeit eine relationale Datenbank vorteilhafter wäre als die aktuelle JSON-basierte Speicherung.

- **Fehlermeldungen sind genauso wichtig wie korrekte Funktionen:** Nicht nur fehlerfreie Funktionen, sondern auch verständliche Fehlermeldungen tragen zur Benutzerfreundlichkeit bei. Durch gezielte Tests konnten wir sicherstellen, dass Fehlermeldungen für ungültige Zähler-IDs, falsche Ablesewerte oder negative Werte klar und nachvollziehbar sind.

6 Fazit

Im Rahmen dieses Projekts haben wir eine Hausverwaltungssoftware prototypisch entwickelt und dabei einen umfassenden softwaretechnischen Prozess durchlaufen – von der Anforderungsanalyse über die Implementierung bis hin zur Qualitätssicherung. Unser Ziel war es, ein System zu entwerfen, das die Erfassung, Verwaltung und Analyse von Verbrauchsdaten effizient unterstützt und dabei stabil, fehlerresistent und benutzerfreundlich ist.

Die Anforderungsanalyse bildete die Grundlage für die Entwicklung unseres Prototyps. Durch ein technisches Review nach ISO 20246 konnten wir potenzielle Unklarheiten in den ursprünglichen Anforderungen identifizieren und gezielt verbessern. Die Trennung zwischen funktionalen und nicht-funktionalen Anforderungen hat sich als vorteilhaft erwiesen, da sie die Strukturierung und Testbarkeit unseres Systems erleichtert hat.

Auf Basis dieser überarbeiteten Anforderungen wurde ein strukturiertes Testkonzept entwickelt. Durch die Kombination von Unit-Tests, Funktionstests, negativen Tests und Performance-Tests konnten wir die wichtigsten Systemfunktionen validieren und sicherstellen, dass das System auch unter Last stabil bleibt. Die durchgeführten Tests bestätigten die Korrektheit der Kernfunktionen und deckten potenzielle Optimierungsansätze auf, insbesondere in Bezug auf die Skalierbarkeit und Datenspeicherung.

Die Implementierung unseres Prototyps folgte einer modularen Architektur mit einem Flask-Backend, einer JSON-basierten Datenspeicherung und eine webbasierte Benutzeroberfläche. Diese Architektur ermöglichte eine schnelle Entwicklung und eine einfache Testbarkeit. Herausforderungen, wie die Validierung von Ablesedaten oder die Simulation von Lasttests, wurden durch gezielte Anpassungen und Optimierungen erfolgreich gemeistert.

Unsere Qualitätssicherungsmaßnahmen haben sich als effektiv erwiesen, um Fehler frühzeitig zu identifizieren und das System iterativ zu verbessern. Die Rückverfolgbarkeit zwischen Anforderungen und Testfällen trug wesentlich dazu bei, dass keine kritische Funktion ungetestet blieb. Zudem zeigten unsere Performance-Tests, dass das System mit hohen Datenmengen umgehen kann, auch wenn eine relationale Datenbank für eine produktive Umgebung empfehlenswert wäre.

Zusammenfassend hat unser Prototyp die wesentlichen Anforderungen erfolgreich umgesetzt und bietet eine solide Grundlage für eine Weiterentwicklung. Falls das System für eine produktive Umgebung ausgebaut werden soll, wären folgende Maßnahmen sinnvoll:

- Migration von JSON zu einer relationalen Datenbank für bessere Skalierbarkeit und Sicherheit.
- Erweiterung der Performance-Tests mit größeren Datenmengen und realistischen Nutzungsszenarien.
- Einführung automatisierter UI-Tests zur Überprüfung der Benutzerfreundlichkeit.

- Implementierung weiterer Fehlerbehandlungen und Sicherheitsmechanismen.

Dieses Projekt hat uns gezeigt, wie entscheidend eine strukturierte Anforderungsanalyse, ein fundiertes Testkonzept und iterative Qualitätssicherungsmaßnahmen für die erfolgreiche Entwicklung einer Software sind. Die gewonnenen Erkenntnisse und Erfahrungen aus diesem Projekt werden uns in zukünftigen Softwareentwicklungsprojekten von großem Nutzen sein.

Tabellenverzeichnis

Listingverzeichnis

Selbstständigkeitserklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit habe ich mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegt.

Bremerhaven, den 11. März 2025 Unterschrift: