Dokumentation zur betrieblichen Projektarbeit

Backend-Entwicklung von API-Endpunkten mit Restful-API, CRUD-Funktionen und Datenbankanbindung für Full-Stack-Anwendung

Prüfungsteilnehmer: Krpan Nikola
Kontumazgarten 26
90429 Nürnberg
Mobil: 0179 448 01 67

Prüfungsnummer: 158 22112

Fachrichtung: Fachinformatiker - Anwendungsentwicklung

Prüfungsjahr: Winter 2023

Ausbildungsbetrieb: Berufsförderungswerk Nürnberg GmbH
Schleswiger Str. 101
90427 Nürnberg

Abgabedatum: 15.12.2023

Inhaltsverzeichnis

1. Einleitung

1.1 Zu meiner Person

Ich heiße Nikola Krpan. Ich bin in Kroatien geboren und lebe seit 6 Jahren in Deutschland. Seit Januar 2022 besuche ich das Berufsförderungswerk Nürnberg, um eine betriebliche Umschulung zum Fachinformatiker-Anwendungsentwicklung zu absolvieren. Mein Praktikum führte ich bei der Firma Empuron AG durch. In dieser Dokumentation werde ich Ihnen meine Erfahrungen und gesammelten Fähigkeiten während der 3-monatigen Praktikumszeit beschreiben.

Fachbegriffe sind unterstrichen und im Glossar in chronologischer Reihenfolge erklärt. Klassennamen sind mit ***bold/italic*** und Terminalbefehle nur mit **bold** gekennzeichnet. Die Methoden sind in der Schriftart `Consolas` dargestellt.

1.2 Der Praktikumsbetrieb

Empuron AG ist eine Softwarefirma, die innovative, technische Lösungen für die erneuerbaren Energiesysteme, für Smart Grids und zur Steigerung der Energieeffizienz entwickelt. Empuron versteht sich als ein kunden- und marktorientiertes Unternehmen. Sie benutzen eine umfangreiche und funktionsstarke Plattform, die im Bereich des Energiemanagments und der Energieeffizenz eingesetzt wird. Empuron wurde 2008 gegründet. Das Unternehmen ist ein kleines Unternehmen mit ca. 10 Mitarbeitern.

1.3 Ausgangssituation

Im Rahmen meiner Ausbildung zum Fachinformatiker Anwendungsentwicklung habe ich bei der Empuron AG mein Praktikum durchgeführt. Es existiert bereits eine firmeninterne Applikation (C@rtan), die auf dem Flask Framework basiert. Dieses ist in der Lage den Energieverbrauch in Echtzeit auszulesen und ermöglicht einen zeitnahen Überblick über den aktuellen Energieverbrauch und über die entsprechenden Kosten. C@rtan verfügt über eine eigene Datenbank und kann Zähler im Netzwerk automatisch finden.

1.4 Projektumfeld

Wie in Punkt 1.2 beschrieben beschäftigt sich die Empuron mit erneuerbaren Energien. Der Hauptentwickler meines Projektes war Herr Sascha Kuhlmann. Wir waren in Kontakt und hatten mehrmals pro Woche Updates bezüglich meines Codes. Es wurden regelmäßige Gesprächsrunden mit dem Chef über den Stand meines Projekts etabliert. Dabei wurden auch allgemeine und fachspezifische Themengebiete des Unternehmen besprochen.

1.5 Projektabgrenzung

Aufgrund des Umfangs des Gesamtprojekts ist es wichtig, dessen Umsetzung zu unterteilen und hier klar abzugrenzen. Das Projekt kann grob in die Erstellung von Frontend und Backend geteilt werden.

Die Erstellung des Backends für die User Zähler, User Benachrichtigungen, User Statistik und User Graphen wird durch den Autor übernommen, und dies gilt auch für das Aufsetzen und Erstellen von CRUD-Funktionen, der API-Endpunkte durch den RESTful API und

den Datenbankstrukturen. Das Frontend und die Hauptlogik der Anwendung wurde und wird von anderen Kollegen umgesetzt.

In cartan:	***user_dash_series, user_dashboard, user_gauge, user_graph, user_notification, user_statistic***
In cartan/rest:	***gauges***

Tabelle 1: von mir entwickelte Modulen

1.6 Projektziel

Das Ziel des Projekts bestand darin, die wichtige Teile des Backends zu schaffen, um für ein zu erstellendes Frontend mit Hilfe von API Routen Zugriff zu ermöglichen. Dadurch wird die Erweiterung einer Anwendung mit grafischer Benutzeroberfläche realisierbar. Die Entwicklung des Frontend war kein Ziel meines Projektes. Die Anwendung wird es dem Benutzer ermöglichen, alle Arten von Informationen wie täglichen/wöchentlichen/monatlichen Stromverbrauch, aktuellen Stand der Energieeinsparung und -ausgaben, tägliche Informationsnachrichten usw. zu visualisieren.

Screenshot von UI Prototype liegt im Anhang/Anlagen (1.6)

2. Projektplanung

Das folgende Kapitel beschreibt, wie das weitere Vorgehen bestimmt wurde. Hierzu sind die Projektphasen geplant worden. Die benötigten Ressourcen und der Entwicklungsprozess wurden ermittelt und festgelegt.

Insgesamt sollte der Arbeitsaufwand für die betriebliche Projektarbeit 80 Stunden umfassen und diesen nicht überschreiten. Die Ressourcen mussten vor Beginn der Durchführung des Projekts auf verschiedene Arbeitsphasen aufgeteilt und strukturiert verplant werden.

2.1 Ressourcenplanung

Die Umsetzung von Projekten im Rahmen einer produktiven Individualanwendung ist in der Regel sehr ressourcenintensiv, daher befindet sich eine detaillierte Auflistung der für dieses Projekt benötigten Hardware-, Software- und Personalressourcen im *Anhang/Anlagen (2.1)* liegt.

Für den gesamten Entwicklungsprozess wurden ausschließlich Open-Source-Software und -Werkzeuge verwendet. Die Entwicklung des Backends erfolgte an einem Desktop-PC mit zwei Monitoren, auf dem das Betriebssystem Linux – Manjaro lief. Das war mein gewöhnlicher Arbeitsplatz. Hier waren keine weiteren Anschaffungen notwendig.

2.2 Projektphasen

Projektphasen	Dauer
Analyse	11
• Ist-Analyse	2
• Analyse von Codebase und Struktur der C@rtan Anwendung	5
• „Lernen“ (Überblick über das Flask-Framework verschaffen)	4
Entwurf	10
• Erstellung eines UML-Diagramms	2
• Entwurf der Testdatenbank	1
• Planung der Architektur	7
Implementierung inkl. Tests	43
• Implementierung der Requirements	1
• Erstellen der Klassen-Modelle und Testdatenbank-Anbindung	5
• Entwicklung von API-Endpunkten und CRUD-Funktionen	25
• Server-Logging und Fehlerbehebung	4
• API Route Testen	4
• Code-Review	2
• Code-Refaktorisierung	2
Abnahme und Einführung	2
Dokumentation	14
• Erstellung eines Testfallhandbuchs	6
• Erstellung von Projektdokumentation	8
Projektdauer	80

Tabelle 2: Zeitplanung

2.3 Technische Maßnahmen

Es mussten zur Erfüllung des Soll-Zustandes folgende Aspekte durchgeführt werden.

- MVC Design Pattern umsetzen
- Entwicklung von API-Endpunkten
- RESTfull API Regeln folgen
- Erstellung von Server Logging für User Aktivitäten
- Generierung von Python-Programmcode
- Erweiterung der SQL Datenbank durch SQLAlchemy
- Database Migration Script mithilfe von Flask Migration
- Der generierte Programmcode muss innerbetrieblichen Coderichtlinien folgen
- Testfälle für Benutzerdokumentation schreiben

2.4 Organisatorische Maßnahmen

Um die Ziele des Projekts zu erreichen, war es notwendig, der etablierten Struktur der Codebase von C@rtan Applikation zu folgen. Ich nutzte alle verfügbaren Ressourcen des Unternehmens und hielt regelmäßige Rücksprachen mit dem leitenden Entwickler und dem Chef über den Status der aktuellen Aufgabe und über mögliche Änderungen und Aktualisierungen seitens des Kunden halten. Das gesamte Projekt wird in Apache Subversion versioniert und alle Schritte im Redmine Ticketsystem dokumentiert.

3. Analyse

3.1 Ist-Analyse

Die Projektabsicht ist es gewesen, für die firmeninterne Applikation C@rtan eine Erweiterung zu entwickeln. Ein wichtiger Systembaustein davon war der Datenpunkt. Ein Datenpunkt wird als Datenadresse in der Datenbank (auch "Technologische Adresse") definiert, die eine technische Zeitreihe eines beliebigen Zyklus (von 1 Sekunde bis 1 Jahr, bzw. als Spontanwert) aufnimmt.

Bspw. kann ein elektrischer Leistungswert in "kW" in einem Zyklus von einer Minute aufgezeichnet werden. Dann wird für jede Minute ein Wert in die Datenbank eingetragen. Diese Datenadresse kann ich als Datenobjekt aufrufen und meine API-Architektur testen, z.B. API “Anfrage” und API “Antwort” für verschiedene Zyklen.

In den Besprechungen wurde mir vom Hauptentwickler mitgeteilt, dass diese Applikation stetig Stromzähler auslesen und in einer Datenbank speichern soll. Es war beabsichtigt, verschiedene Module zu implementieren. Dies wurde mir als Aufgabe übertragen.

Als ersten konkreten Schritt verschaffte ich mir einen Überblick über die ***Datenobject***-Klasse. Hier war die Erkenntnis über das Verhalten der Klasse sowie die Parameterbewertung von Wichtigkeit.

Da alle weiteren von mir entwickelten Module mit dieser Klasse kommunizieren werden müssen, war die Kenntnis über diese Klasse entscheidend.

3.2 Soll-Konzept

Ziel der Programmierung war es, die von mir geschriebenen Module in das C@rtan zu implementieren. Um die verschiedenen Varianten dieser Parameterdaten aus einer Datenbank abfragen zu können, z.B. „zeige mir nur den Strom, den ich in der letzten Woche verbraucht habe“ wurden verschiedene Module benötigt.

Die Applikation sollte perfekt integriert und nahtlos in der Lage sein, kontinuierlich Stromzähler auszulesen und diese Daten in Echtzeit in der Datenbank dauerhaft zu speichern. Sie sollte, über eine intuitive Benutzeroberfläche verfügen, die es Benutzern möglich macht, den Stromverbrauch in verschiedenen Zeiträumen zu überwachen und detaillierte Analysen abzurufen.

Alle API-Routen sollen funktionieren und Abfragen von Daten aus der Datenbank reibungslos abrufen lassen. Clients sollen in der Lage sein, ihre Statistik mit diesen abgerufenen Daten graphisch darzustellen. User Benachrichtigungen sollen Clients über Ihre täglichen/wöchentlichen/monatlichen Stromverbrauch informieren. Ziel ist es auch, Tipps über Sparmöglichkeiten ausbringen zu können.

3.3 Analyse von Codebase und Struktur die C@rtan Anwendung

Am Anfang meines Projektes muss ich die firmeninterne Applikation C@rtan kennenlernen. Es handelt sich um eine Applikation, die das Verwalten und Zugreifen auf Datenobjekte mit verschiedenen Eigenschaften und Zugriffssteuerungsregeln umfasst.

Die Hauptklasse in Applikation ist die ***Datenobject***-Klasse. Diese Klasse definiert ein Datenbankmodell für ein Datenobjekt System, einschließlich zugehöriger Tabellen und Hilfsfunktionen für Abfragen und die Arbeit mit den Daten aus dem Client Stromzähler.

Für meinen Teil des Projekts war dies die wichtigste Klasse, und meine Module würden ständig mit ihr kommunizieren, da sie Datenobjekte repräsentiert, die aus den Zählern gelesen werden. Mit den Eigenschaften dieser Klasse konnte ich API-Aufrufe in meinen Modulen tätigen und die verschiedenen Arten von Daten darstellen, die vom Client angefordert wurden, wie z.B. die Anzeige von täglichen und wöchentlichen Ausgaben.

3.4 „Lernen“ (Überblick über das Flask-Framework verschaffen)

Die C@rtan Applikation basiert auf dem Python Flask Framework. Daher war es für das Projekt sehr wichtig, dass ich die Funktionsweise nachvollziehen konnte. Somit war eine Bewertung und Anwendung des Frameworks gegeben.

Flask ist ein Micro-Web-Framework, d. h. es ist einfach und leichtgewichtig konzipiert. Es bietet die grundlegenden Werkzeuge und Komponenten, die für die Erstellung von Webanwendungen benötigt werden. Es legt keine strenge Struktur oder eine Menge eingebauter Funktionen fest. Das macht es für mich praktisch, Code zu schreiben, der leicht zu lesen ist und mir erlaubt, die Daten zu kapseln.

Es folgt dem WSGI-Standard (Web Server Gateway Interface), der es Entwicklern ermöglicht, verschiedene Erweiterungen und Bibliotheken zu verwenden, um ihren Anwendungen spezifische Funktionen hinzuzufügen. Es ermöglicht zu C@rtan Applikation die Daten von den Zählern schnell zu lesen und in der Datenbank zu speichern.

Eine der wichtigsten und wertvollsten Komponenten im Flask-Framework sind die sogenannten "Blueprints". Flask Blueprints kapseln Funktionalitäten wie Ansichten (Views), Vorlagen (Templates) und andere Ressourcen. Anstatt “Views” und anderen Code direkt bei einer Anwendung zu registrieren, erfolgt die Registrierung bei einer Blueprint. Um effizientere Code zu entwickeln, war es notwendig diese Blueprint zu entwickeln und im Projekt zu implementieren.

```
user_dashboard_app = Blueprint("user_dashboard_app", __name__)
user_dashboard_view = UserDashboardApi.as_view("user_dashboard_api")

user_dashboard_app.add_url_rule(
    "/cartan/user_dashboard/",
    defaults={"user_dashboard_id": None},
    view_func=user_dashboard_view,
    methods=["GET"],
)
user_dashboard_app.add_url_rule(
    "/cartan/user_dashboard/",
    view_func=user_dashboard_view,
    methods=["POST"],
)
user_dashboard_app.add_url_rule(
    "/cartan/user_dashboard/<int:user_dashboard_id>",
    view_func=user_dashboard_view,
    methods=["GET", "DELETE", "POST"],)
```

Abbildung 1: Blueprint von UserDashboard-Klasse

Um eine Flask-Blueprint zu verwenden, war es notwendig sie zu importieren und dann in der Anwendung die Registrierung mit `register_blueprint()` durchzuführen. Durch die Bekanntgabe der Flask-Blueprint Methode, wird die Anwendung um ihre Inhalte erweitert.

```
from app.cartan.user_gauge.views import user_gauge_app
from app.cartan.rest.gauges.views import gauge_app
from app.cartan.user_dash_series.views import user_dash_series_app
from app.cartan.user_notifications.views import user_notifications_app
from app.cartan.user_statistic.views import user_statistic_app
from app.cartan.user_dashboard.views import user_dashboard_app
from app.cartan.user_graph.views import user_graph_app

app.register_blueprint(user_gauge_app)
app.register_blueprint(gauge_app)
app.register_blueprint(user_dash_series_app)
app.register_blueprint(user_notifications_app)
app.register_blueprint(user_statistic_app)
app.register_blueprint(user_dashboard_app)
app.register_blueprint(user_graph_app)
```

Abbildung 2: Blueprint Registration mit dem App von meinen Modulen

Blueprint ermöglicht den Einsatz verschiedener Design-Prinzipien wie der Trennung von Verantwortklichkeiten (Separation of Concerns) und "DRY" (Don't Repeat Yourself), was bedeutet, dass sich Code nicht wiederholen sollte.

4. Entwurf

4.1 Erstellung eines UML-Diagramm

Als erstes habe ich mich in der Entwurfsphase dazu entschlossen, ein UML-Diagramm für meine Module zu erstellen. Die Erstellung eines UML-Diagramms ist wichtig, da es eine visuelle “Blueprint” liefert, die die Struktur, Beziehungen und Funktionalität des Systems verdeutlicht, Fehler reduziert und eine effektive Kommunikation zwischen Teammitgliedern ermöglicht. Dafür habe ich mich für “Visual Paradigm” entschieden, weil es eine benutzer-freundliche Oberfläche besitzt, was den Entwicklungsprozess vereinfacht und verbessert.

Ich habe mich entschieden, die **`UserDashboard`**-Klasse zu erstellen, weil ich eine klare Trennung der Zuständigkeiten (Separation of Concerns) zwischen der **Applikation**(**User**)-Klasse und allen anderen Modulen haben wollte. Außerdem existierten die Applikation und die Datenobjektklasse bereits vorher, und ich wollte den Code nicht ändern, was das O-pen/Closed-Principle verletzen würde. Stattdessen konnte ich mit der **`UserDashboard`**-Klasse die direkte Kommunikation zwischen der Applikation und anderen Modulen nur über sie ermöglichen.

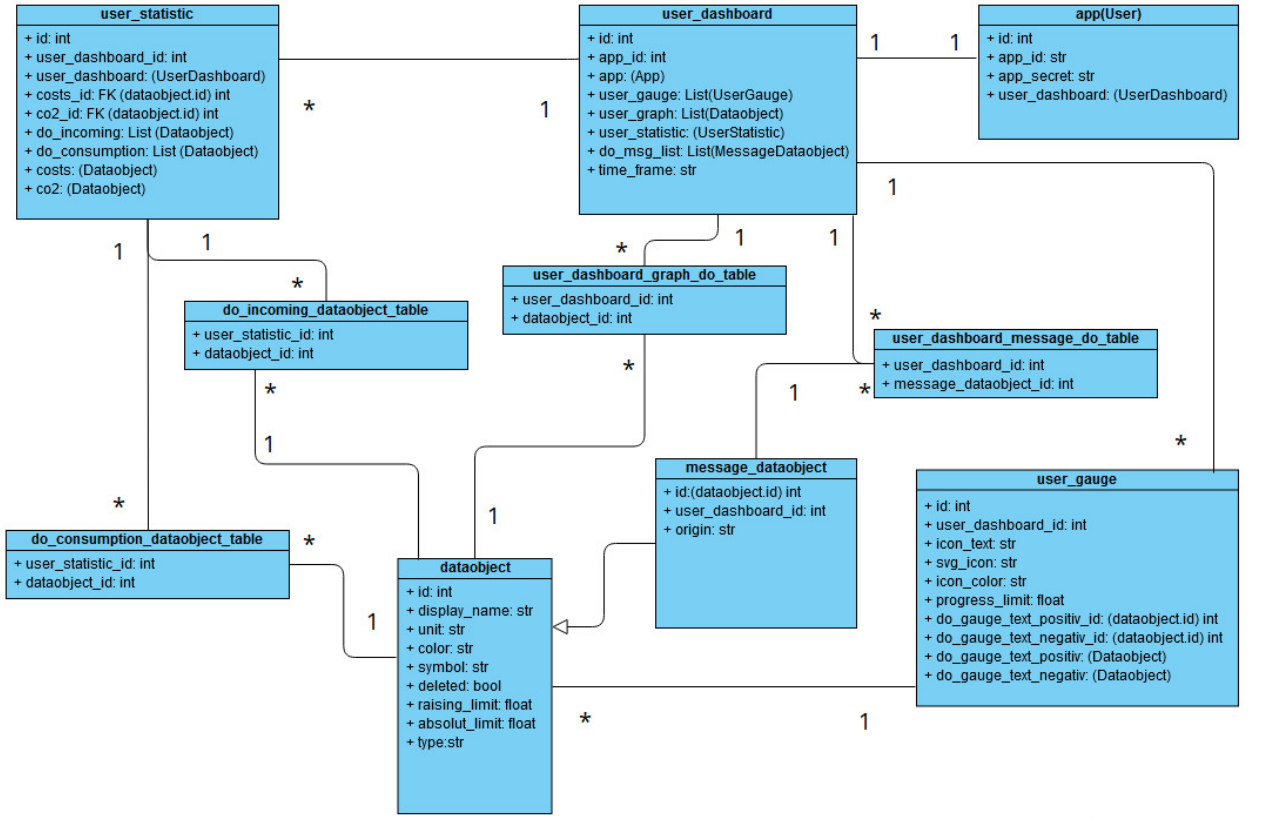

Abbildung 3: UML-Diagramm

4.2 Entwurf der Testdatenbank

Ich habe mich in diesem Fall für SQLite-Datenbank entschieden. SQLite ist ein plattformübergreifendes Open-Source-Datenbankmanagementsystem, das auf allen Betriebssystemen, einschließlich macOS, Windows, Linux usw., ausgeführt werden kann. SQLite erfordert keine Konfiguration. Es benötigt keine Einrichtung oder Administration.

Ich habe auch "Dummy"-Daten erstellt, die den echten Daten entsprechen, die vom Benutzer verwendet werden. Für einige davon wurde eine CSV (Comma Separated Value) -Datei verwendet, während ich die Darstellung von Datenobjekten programmatisch mithilfe von REST-Client Erweiterung heraus gefüllt habe. Außerdem möchte ich hinzufügen, dass alle Daten, mit denen ich in meinem Projekt gearbeitet habe, "Dummy"-Daten waren, aber diese Daten entsprechen strukturell gesehen vollständig dem Aufbau der echten Daten.

Ich benötigte diese Testdatenbank, um die API-Routen zu prüfen, da ich keinen Zugriff auf die tatsächlichen Benutzerdaten hatte.

4.3 Planung der Architektur

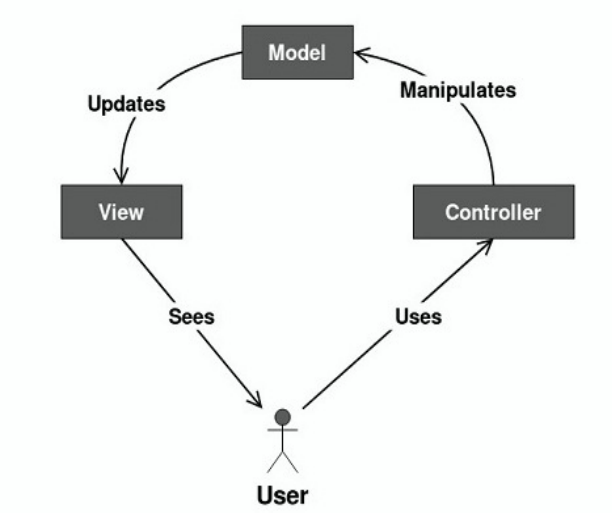

Abbildung 4: MVC-Design Muster

Für den Entwurf und die Implementierung der firmen internen C@rtan Applikation wurde das Model-View-Controller-Pattern (MVC) eingesetzt.

Deshalb habe ich die gleichen Muster für meinen Teil verwendet.

Bei MVC wird ein interaktives System in die Komponenten Model, View und Controller geteilt.

- ✓ Das Model hält und verarbeitet die konkreten Daten
- ✓ Die View stellt die Daten dar und präsentiert die Benutzerschnittstelle
- ✓ Der Controller interpretiert und verarbeitet die Eingaben des Benutzers

Das führt dazu, dass die Daten unabhängig von der Darstellung sind und die Benutzerschnittstelle flexibel ausgetauscht werden kann, beispielsweise, wenn anstelle einer grafische Benutzeroberfläche (GUI) in Zukunft eine webbasierte Benutzeroberfläche verwendet werden soll.

Alle Module, die ich entwickelt habe, verwenden das gleiche Muster und haben die gleichen Dateien mit dem gleichen Namen (der Code darin ist für jedes Modul anders). Also habe ich beschlossen, ein Beispiel nur von einem meiner Module (***user_dashboard***) zu machen.

Anbei ein Beispiel eines von mir entwickelten Modul (***user_dashboard***) eines MVC Musters:

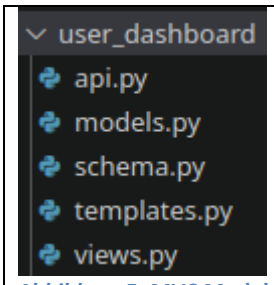

Abbildung 5: MVC Modul

Jeder Dashboard-Ordner wurde in Module unterteilt, wie zum Beispiel: API (Controller), Models (Model), Schema, Templates, Views.

4.4 API Modul

Im API-Modul werden alle für diese Schnittstelle notwendigen CRUD (Create, Read, Update, Delete) -Funktionen vorhanden sein. Die CRUD-Funktionen in der API ermöglichen das Erstellen, Lesen, Aktualisieren und Löschen von Daten in der Datenbank über die Benutzeroberfläche oder Anwendungsprogrammierschnittstelle (API).

4.5 Models Modul

Im Models-Modul habe ich für jede benötigte Schnittstelle eine Modelklasse erstellt. Jede Klasse wird Eigenschaften haben, die es dem Benutzer ermöglichen, mit ihren Daten zu arbeiten. In den Modellen sind auch Datenbankverbindungen zwischen verschiedenen Klassen hergestellt worden. Hierfür habe ich einen Objekt-Relational Mapper (ORM) verwendet.

ORM ist eine Bibliothek, die den für die Datenmanipulation erforderlichen Code kapselt, sodass keine direkten SQL-Befehle genutzt werden müssen. Stattdessen interagieren Sie direkt mit einem Objekt in derselben Sprache, die Sie verwenden.

4.6 Schema Modul

Im Schema-Modul befindet sich ein JSON Schema für die Schnittstelle der DB-Modellen. Das JSON-Schema dient dazu, die Validierung von JSON-Daten sicherzustellen, die mit der Schnittstelle des DB-Modellsarbeiten. Es stellt sicher, dass die Daten die erwarteten Typen und Strukturen haben und den Anforderungen des Modells entsprechen.

4.7 Templates Modul

Im Templates-Modul befinden sich die Funktionen, die dazu verwendet werden, Python-Objekte in das entsprechende Format zu serialisieren. In meinem Fall handelte es sich um JSON. Serialisierung ist der Prozess der Umwandlung des Zustands eines Objekts in einen Byte-Stream. Dieser Byte-Stream kann dann in einer Datei gespeichert, über ein Netzwerk gesendet oder in einer Datenbank gespeichert werden. Es können verschiedene Formate für die Serialisierung verwendet werden, wie JSON, XML und binär. Die Deserialisierung ist der umgekehrte Prozess der Serialisierung. Dabei handelt es sich um die Umwandlung eines Byte-Streams zurück in ein Objekt.

4.8 View Modul

Im View-Modul werden die REST-Endpunkte (API-Routen) für die Schnittstellen definiert.

5. Implementierung inkl. Tests

5.1 Implementierung der Requirements

Zu Beginn der Entwicklungsphase musste ich meine Entwicklungsumgebung einrichten und alle erforderlichen Komponenten mit folgendem Befehl installieren:

```
pip install -r requirements.txt
```

In der Anforderungsdatei befindet sich eine Liste von angegebenen Paketen mit ihren angegebenen Versionen, die es Entwicklern ermöglichen, mit denselben Werkzeugen zu arbeiten.

5.2 Erstellen der Model-Klassen und Testdatenbank Anbindung

Ich begann mit der Erstellung meiner Model-Klassen aus meinem UML-Klassendiagramm und importierte alle erforderlichen Module. Ein Model ist eine Klasse, die eine Tabelle in einer Datenbank darstellt, und bei der jedes Attribut der Klasse ein Feld der Tabelle ist. In der Model-Klasse habe ich alle erforderlichen Attributen aus bestimmten Klassen platziert, die es meinen CRUD-Funktionen ermöglichen, die Zählerdaten zu manipulieren und dem Benutzer die korrekte Darstellung in Form von Grafiken und Diagrammen zu ermöglichen.

Ich habe `primary_key=True` verwendet, was bedeutet, dass es sich um einen Primärschlüssel für die Tabelle handelt und automatisch inkrementiert werden soll. Jede von mir erstellte Model-Klasse muss die Basisklasse von SQLAlchemy namens ***`db.Model`*** erben, weil damit SQLAlchemy Modelle definieren kann und mit der Datenbank integrieren.

Außerdem habe ich alle erforderlichen Datenbankbeziehungen mithilfe der Flask Funktion `relationship()` hergestellt. Einige der Klassen befanden sich in einer m:n-Beziehung, daher musste ich die dritte Tabelle erstellen, die die beiden m:n-Tabellen verknüpfen würde. Ich habe die Regeln des SQLAlchemy Objekt-Relational Mappers genau befolgt, um sicherzustellen, dass meine Model-Klasse und Datenbank Beziehungen ordnungsgemäß funktionieren.

Nachdem ich Modelle und Beziehungen definiert hatte, stellte ich sicher, dass die Attributdatentypen meiner Schema-Modulklassen korrekt sind, damit meine Migration keine Probleme mit der Datenbank verursacht. Anschließend verwendete ich die Flask-Migrate-Erweiterung, um Datenbanktabellen aus meinen Modellklassen zu erstellen.

`flask db init`	fügt einen Ordner für Migrationen hinzu
`flask db migrate -m "Initial"`	generiert eine anfängliche Migration
`flask db upgrade`	Änderungen in der Datenbank übernehmen

Tabelle 3: Migration Befehlen

5.3 Entwicklung von API-Endpunkten und CRUD Funktionen

Alle von mir entwickelten API-Endpunkte und CRUD-Funktionen verwendeten ähnliche Muster und einige Variationen der (GET, POST, PUT, DELETE) -Methoden. Aufgrund dessen denke ich, dass ich den Ablauf der einer API- Klasse erklären könnte, und nicht jede Klasse und jedes Modul separat zu beschreiben.

5.4 GET (READ)-Methode

Diese Methode ruft Daten zu einem ***UserDashboard*** ab. Es gibt verschieden Parametern von ausgewählten Objekt aus, wie z.B Diagramme, Statistiken, Benachrichtigungen.

```
class UserDashboard(MethodView):
    def __init__(self) -> None:
        if (request.method != "GET" and request.method != "DELETE")
                                    and not request.json:
            log_user_dash.error("UserDashboard API failed to initialize")
            abort(400)

    def get(self, user_dashboard_id: int = None) -> Response:
        user: App = request.user
        log_user_dash.info(
             f"User tries to retrieve UserDashboard object:#{user_dashboard_id}")

        user_dashboard: UserDashboard = user.user_dashboard
        if user_dashboard_id:
            user_dashboard: UserDashboard = db.one_or_404(
                db.select(UserDashboard).filter_by(id=user_dashboard_id))

            if not user_dashboard:
                log_user_dash.error(
                   f"User has no rights to access UserDashboard
                           ID#{user_dashboard_id}")
                return jsonify({"error": "User has no access rights"}), 401

            response = {"result": "ok",
                        "user_dashboard": user_dashboard_obj(user_dashboard)}
            return jsonify(response), 200

        if not user_dashboard:
            log_user_dash.info("No UserDashboard object found for the user.")
            return jsonify({"result": "No data fetched", "user_dashboard": []}), 200

        log_user_dash.debug(f"Logging details of object: {user_dashboard}")

        user_gauges = user_gauge_list_obj(user_dashboard.user_gauge)
        user_graphs = dataobjects_obj(user_dashboard.user_graph)
        user_statistic = user_statistic_obj(user_dashboard.user_statistic)
        do_message_list = dataobjects_obj(user_dashboard.do_message_list)
        time_frame = user_dashboard.time_frame

        response = {
            "user_gauges": user_gauges,
            "user_graphs": user_graphs,
            "user_statistic": user_statistic,
            "do_message_list": do_message_list,
            "time_frame": time_frame,}

        log_user_dash.info("UserDashboard object retrieved.")
        return jsonify({"result": "ok", "user_dashboard": response}), 200
```

Abbildung 6: GET (READ)-Methode Quellcode

5.5 POST (WRITE)-Methode

Diese Funktion erstellt neue Einträge in der Datenbank unter Berücksichtigung von verschiedenen Objekten wie Benutzerstatistiken, Diagrammen, und Nachrichten.

```
def post(self) -> Response:
    user_dashboard_json: Dict = request.json
    error = best_match(Draft4Validator(schema).iter_errors(user_dashboard_json))
    if error:
        log_user_dash.error(f"Dataobject Schema validation failed | {error.message}")
        return jsonify({"error": error.message}), 400

    log_user_dash.debug("user_dashboard_json: " + json.dumps(user_dashboard_json))

    user_gauge = user_dashboard_json.get("user_gauge")
    user_graph = user_dashboard_json.get("user_graph")
    user_statistic_id = user_dashboard_json.get("user_statistic")
    do_message_list = user_dashboard_json.get("do_message_list")
    time_frame = user_dashboard_json.get("time_frame")

    user_statistic = db.session.query(UserStatistic)
                 .filter_by(id=user_statistic_id).scalar()
    user_gauge_objects = db.session.query(UserGauge)
                 .filter(UserGauge.id.in_(user_gauge)).all()
    user_graph_object = (db.session.query(Dataobject)
                 .filter(Dataobject.id.in_(user_graph)).all())
    do_message_objects = (db.session.query(MessageDataobject)
                 .filter(MessageDataobject.id.in_(do_message_list)).all())

    user_dashboard = UserDashboard(
        app_id=request.user.id,
        user_gauge=user_gauge_objects,
        user_graph=user_graph_objects,
        user_statistic=user_statistic,
        do_message_list=do_message_objects,
        time_frame=time_frame,)

    try:
        db.session.add(user_dashboard)
        db.session.commit()
        log_user_dash.info(
            "UserDashboard object successfully written to the database.")
        response = {"result": "ok", "user_dashboard": user_dashboard_obj(
            user_dashboard)}
        return jsonify(response), 201

    except exc.IntegrityError as error:
        log_user_dash.exception(
            f"IntegrityError occured while commiting {user_dashboard} to the Database"
            f"| {type(error)}: {error}")

        db.session.rollback()
        response = {
            "error": "user_dashboard object already exists",
            "user_dashboard": user_dashboard_json,}
        return jsonify(response), 400
```

Abbildung 7: POST (WRITE)-Methode Quellcode

5.6 PUT (UPDATE)-Methode

Die POST- und PUT-Methoden sind ziemlich ähnlich, der einzige Unterschied besteht darin, dass die PUT-Methode einen ID-Parameter in der Methodensignatur hat, der dabei hilft, ein Objekt wie Benutzerstatistiken, Diagrammen, und Nachrichten in der Datenbank aktualisieren zu können. Deshalb kam ich zu der Überzeugung, es wäre überflüssig, auch die PUT-Methode zu zeigen.

5.7 DELETE (LÖSCHEN)-Methode

Diese Methode kann mithilfe des ID-Parameters das ausgewählte Objekt aus der Datenbank löschen.

```
def delete(self, user_dashboard_id) -> Response:
    user_dashboard: UserDashboard = db.one_or_404(
        db.select(UserDashboard).filter_by(id=user_dashboard_id))

    if not user_dashboard:
        log_user_dash.error(
            f"UserDashboard ID#{user_dashboard_id} not found in Database.")
        return jsonify({"error": "UserDashboard object not found"}), 404

    db.session.delete(user_dashboard)
    db.session.commit()
    log_user_dash.info(
        f"UserDashboard ID#{user_dashboard_id}
            succesfully deleted from the Database.")
    return jsonify({}), 204
```

Abbildung 8: DELETE (LÖSCHEN)-Methode Quellcode

5.8 Server Logging und Fehlerbehebung

Server-Logging und Fehlerbehebung sind zwei wirklich wichtige Teile des Entwicklungszyklus. Es bezeichnet die Praxis, Ereignisse und Aktivitäten, die auf einem Server oder in einer Applikation auftreten, in einer strukturierten und systematischen Weise aufzuzeichnen. Im Server Log werden Informationen wie Fehler, Warnungen, Informationsmeldungen und andere relevante Daten im Zusammenhang mit dem Betrieb des Servers erfasst.

Jedes Mal, wenn ein Entwickler ein Problem hat oder einen bestimmten Teil des Codes missversteht oder wissen möchte warum ein bestimmter Fehler auftritt, kann er das Server-Log auswerten oder mit Hilfe von "Breakpoints" ein Fehler beheben. Es war für mich während der Entwicklung ein unverzichtbares Werkzeug und hat mir geholfen, spezifische Probleme besser zu verstehen, die während der Codeimplementierung aufgetreten sind.

5.9 API Route Testen

Ein weiterer wichtiger Teil der Entwicklung war, meine REST-API-Routen mithilfe von REST-Client Erweiterung zu testen. Die REST Client Erweiterung ist ein Werkzeug, das in integrierten Entwicklungsumgebungen (IDEs) verwendet wird, um das Testen und die Interaktion mit RESTful-Web-Services direkt innerhalb des Code-Editors zu erleichtern. Es bietet auch Unterstützung für Token-Authentifizierung. Dies stellt sicher, dass Entwickler authentifizierte Anfragen an sichere APIs senden können.

Diese Erweiterung hat es mir ermöglicht, meine Testdatenbank mit Dummy-Daten auszufüllen, die echten Benutzerdaten entsprechen, und verschiedene Datenbankanfragen mit meinen CRUD-Funktionen durchzuführen.

```
GET http://localhost:5000/cartan/gauges/HTTP/1.1
X-APP-TOKEN: {{$dotenv token}}
X-APP-ID: {{$dotenv client}}
content-type: application/json
```

Abbildung 9: Route-Testing GET-Methode von *gauges* Modul

5.10 Code-Review

An diesem Punkt hatte ich ein Online-Meeting mit dem leitenden Entwickler, der eine Überprüfung meines Codes durchgeführt hat.

Code-Reviews sind normalerweise in den vorhandenen Prozess eines Teams integriert. Damit wird sichergestellt, dass der Code-Reviewer den Code auf Probleme überprüft, die Maschinen übersehen. Außerdem wird verhindert, dass sich ungünstige Entscheidungen bezüglich des Codes auf die Hauptentwicklung auswirken.

Es wurde beschlossen, dass ich “Code Refaktorisierung” (Refactoring) dürchführen muss, um einige logische Fehler zu korrigieren und redundanten Code zu löschen.

5.11 Code Refaktorisierung

In dieser Phase habe ich die Änderungen am Code vorgenommen und den Ratschlägen des leitenden Entwicklers gefolgt. Code Refaktorisierung ist der Prozess der Umstrukturierung von Code, ohne dessen ursprüngliche Funktionalität zu verändern. Ziel des Refaktorisierung ist es, den internen Code durch viele kleine Änderungen zu verbessern, ohne das externe Verhalten des Codes zu verändern.

6. Abnahme und Einführung

Nachdem ich die Code Refaktorisierung abgeschlossen hatte, ging ich den geänderten Code mit dem leitenden Entwickler durch und erklärte die Methoden, die ich während des Entwicklungsprozesses verwendet hatte.

Damit wurde festgestellt, dass mein Code den Anforderungen entspricht.

7. Dokumentation

7.1 Erstellung eines Testfallhandbuchs

Nachdem ich mit der Programmierung fertig war, begann ich mit dem Benutzeroberflächentest. Ich öffnete die C@rtan-Anwendung im Browser und ging manuell durch alle Optionen in der Benutzeroberfläche, um zu sehen, ob einige von ihnen nicht ordnungsgemäß funktionieren. Für jeden dieser Versuche musste ich einen Testfall schreiben. Ein Testfall ist eine Reihe von Bedingungen oder Variablen, unter denen ein Tester feststellen wird, ob ein System unter Test die Anforderungen erfüllt oder ordnungsgemäß funktioniert. Jeden Testfall musste ich im Testfall-Handbuch aufschreiben.

Name des Testfalls

2.2.1 Netzwerk Parameterisierung

Testfall Beschreibung

Dieser Testfall ist für den Netzwerk Parameterisierung.

- - - - - - - - - - - - - - -

Erfordernis:

1. Der Benutzer ist auf der Anwendung autorisiert und befindet sich im "Cartan"-Tab auf der persönlichen Konto-Seite.
2. Der Benutzer befindet sich im „Konfiguration“ Menü
3. Der Benutzer befindet sich im „Netzwerk“ Menü

Anzahl der Prüfschritte: 3 auszuführende Schritte.

Keine Vorbedingungen

Test-Schritte

Name	Beschreibung	Erwartetes Ergebnis	Bestanden	Geschei tert Err.Nr.
Schritt 1	Klicken Sie auf das Konfigurations-menü.	Es sollte eine neue Menüleiste geöffnet werden, die Optionen für die Parametrisierung enthält.		
Schritt 2	Klicken Sie auf das Netzwerksme-nü.	Ein Fenster wird geöffnet, das Ihnen die Optionen für die Netz-werk-Parametrisierung und die automatische Suche nach Zählern bietet.		
Schritt 3	Geben Sie Ihre IP-Adresse und Netzwerkmaske ein und klicken Sie auf die Schaltfläche "Zähler su-chen".	Wenn die Daten korrekt eingege-ben wurden, werden Ihnen die bereits existierende gefundene Zähler angezeigt.		

Bemerkung

Abbildung 10: Testfall Beispiel

7.2 Erstellung von Projektdokumentation

Am Ende habe ich meine Projekt-Dokumentation für die IHK geschrieben. Ich musste keine tatsächliche Benutzerdokumentation schreiben. Die Mitarbeiter des Unternehmens haben das getan. Für meine Projektdokumentation habe ich Microsoft Word verwendet.

8. Fazit/Zusammenfassung

8.1 Soll -/ Ist-Vergleich

Weil während der Implementierungsphase bereits alle Klassen und Methoden mit Pyton Docstring-Kommentaren versehen wurden, konnte ich im Dokumentationsphase Zeit spa-ren. Auch in der Abnahme und Einführung wurde Pufferzeit gewonnen, weil ich während des Code-Reviews mit dem Lead Developer über meinen Entwicklungsprozess gespro-chen habe. Die Analysephase fiel in der Summe eine Stunde länger aus, weil ich mehrere Stunden für C@rtan Applikation Analyse gebraucht habe. Für das Server Logging und Fehlerbehebung habe ich eine zusätzliche Stunde gebraucht.

Projektphasen	Soll	Ist	Differenz
Analyse	11	12	+1
• Ist-Analyse	2	2	
• Analyse von Codebase und Struktur der C@rtan Anwendung	5	6	+1
• „Lernen“ (Überblick über das Flask-Framework verschaffen)	4	4	
Entwurf	10	10	
• Erstellung eines UML-Diagramms	2	2	
• Entwurf der Testdatenbank	1	1	
• Planung der Architektur	7	7	
Implementierung inkl. Tests	43	44	+1
• Implementierung der Requirements	1	1	
• Erstellen der Klassen Modelle und Testdatenbank-Anbindung	5	5	
• Entwicklung von API-Endpunkten und CRUD-Funktionen	25	25	
• Server-Logging und Fehlerbehebung	4	5	+1
• API Route Testen	4	4	
• Code-Review	2	2	
• Code-Refaktorisierung	2	2	
Abnahme und Einführung	2	1	-1
Dokumentation	14	13	-1
• Erstellung eines Testfallhandbuchs	6	6	
• Erstellung von Projektdokumentation	8	7	-1
Projektdauer	80	80	

Tabelle 4: Ist-Soll Vergleich

8.2 Fazit

Das Projektziel wurde gemäß allen Anforderungen erreicht. Alle von mir implementierten Module funktionieren wie beabsichtigt. Parallel zu mir hat der Frontendentwickler Teilbereiche der Applikation, die mit dem Backend kompatibel sind, erstellt. Die API-Routen sind mit den Ressourcen der Datenbank verbunden, und der Client kann mithilfe der grafischen Benutzeroberfläche Anfragen stellen und Antworten mit den gewünschten Daten auf seinem Display erhalten. Es ist möglich, die Daten in verschiedenen Diagrammformen zu manipulieren und Benachrichtigungen über Energieeinsparungen oder Stromverbrauch zu erhalten. Die mir für das Projekt zugewiesene Zeit war ausreichend. Es ist möglich die Applikation mit weiteren Modulen zu ergänzen.

8.3 Persönliches Ausblick

Das Projekt war für mich eine hervorragende Möglichkeit, wertvolle Erfahrungen aus den Bereichen Projektarbeit und Entwicklung zu sammeln. Während der Umsetzung des Projektes konnte ich meine Fähigkeiten im Umgang mit der Programmiersprache Python vertiefen. Ich habe festgestellt, dass der Einsatz von Entwurfsmustern wie MVC sinnvoll ist und die Arbeit erleichtert.

Anhang

I. Anlagen

1.6 Projektziel – UI Prototype von C@rtan Applikation

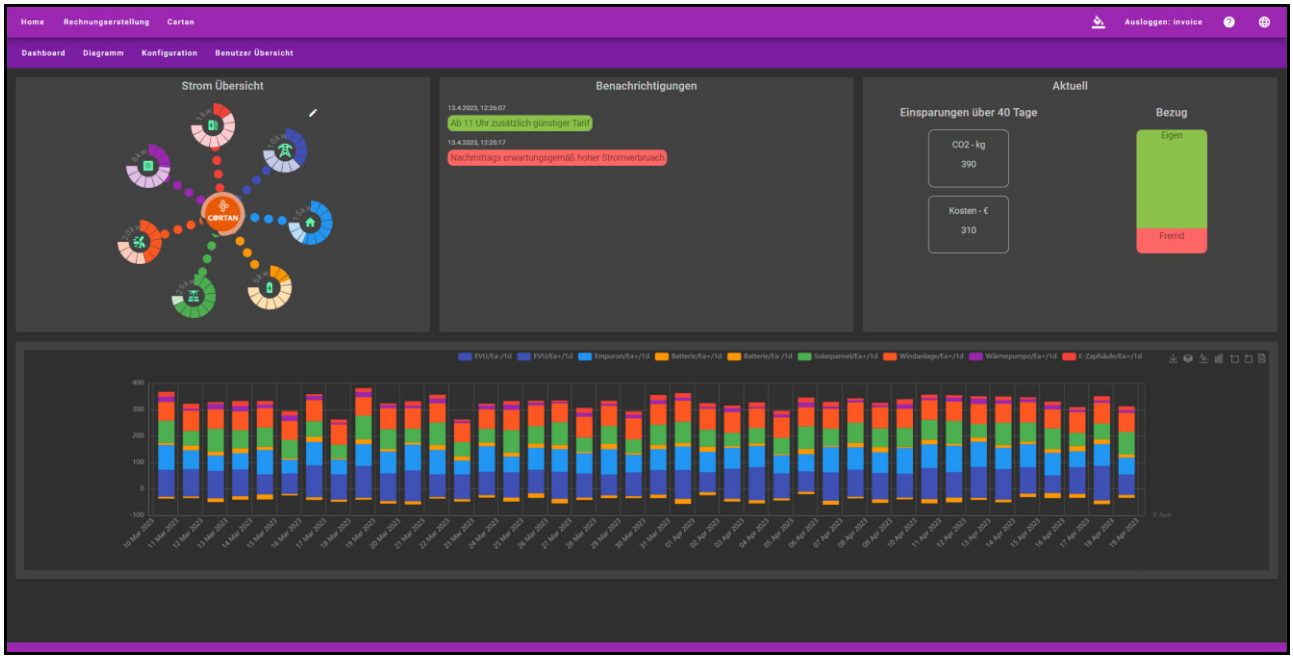

Abbildung 11: UI Prototype

2.1 Ressourcenplanung – Verwendete Ressourcen

Hardware

- Büroarbeitsplatz mit Desktop-PC
- Büroeigener Server – Bereitstellung der Testumgebung

Software

- EMPURON C@rtan Sofware System (Unternehmens eigenes Software)
- Flask-Framework
- SQLAlchemy – ORM (Object Relational Mapper) d.h. erleichtert Verbindung zwischen Python und SQL-Datenbanken
- Influx DB – Datenbanksystem
- SQLite DB - Datenbanksystem
- Visual Paradigm – Programm zur Erstellung von Diagrammen und Modellen
- Waitress – Python-WSGI (Web Server Gateway Interface)-Server
- Dbeaver – Verwaltungswerkzeug für MySQL-Datenbanksysteme
- Redmine – Projekmanagementsoftware
- Manjaro Linux – Betriebssystem
- Visual Studio Code – IDE
- Pip – Package Manager für das Projekt
- Apache Subversion – Version Control System

Personal

- Auszubildender Fachinformatiker für Anwendungsentwicklung – Umsetzung des Projektes
- Anwendungsentwickler/Lead Developer – Festlegung der Anforderungen, Mitgestaltung der Oberflächen und Abnahme der Anwendung
- Geschäftsführer – Beratung, Hilfe bei Fragestellungen

II. Glossar

Begriff	Erklärung
Frontend	bezieht sich auf den Teil einer Software oder einer Anwendung, mit dem Benutzer interagieren
Backend	bezeichnet die nicht sichtbaren Teile einer Software, die im Hintergrund arbeiten, wie die Server, Datenbanken und die Logik
CRUD	steht für Create, Read, Update, Delete und bezieht sich auf die grundlegenden Operationen, die in vielen Datenbanksystemen verwendet werden, um Daten zu erstellen, lesen, aktualisieren und löschen
API-Endpunkte	ist ein Kontaktpunkt zwischen einem API-Client und einem API-Server. API-Clients senden Anfragen an API-Endpunkte, um auf die Funktionalität und die Daten der API zuzugreifen
REST-API	ist eine Schnittstelle, die es Applikationen ermöglicht, miteinander zu kommunizieren und Informationen auszutauschen, indem sie HTTP-Methoden wie GET, POST, PUT und DELETE verwenden
API-Routen	sind spezifische Endpunkte oder Pfade in einer API, die definierte URLs sind, um auf bestimmte Funktionen oder Ressourcen zuzugreifen und Aktionen wie das Abrufen, Aktualisieren, Löschen oder Erstellen von Daten durchzuführen
Open-Source Software	bezeichnet Programme, deren Quellcode öffentlich zugänglich ist
Linux Manjaro	ist eine auf Arch-Linux basierende, benutzerfreundliche Linux-Distribution
Design Pattern	sind bewährte Lösungsansätze für häufig auftretende Designprobleme in der Softwareentwicklung, die als generische Vorlagen dienen, um wiederkehrende Probleme strukturiert zu lösen und eine bessere Code-Architektur zu ermöglichen.
API-Anfrage	ist eine spezifische Aufforderung von einer Anwendung an eine API, um auf bestimmte Daten oder Funktionen zuzugreifen, indem sie die definierten Endpunkte und Parameter nutzt, um Informationen zu senden oder zu empfangen
API-Antwort	ist die Rückmeldung oder das Ergebnis, das von der API an eine Anwendung gesendet wird, nachdem diese eine Anfrage gestellt hat; sie enthält oft Daten, Statuscodes und Informationen, die als Reaktion auf die Anfrage bereitgestellt werden
Design Prinzipien	sind Richtlinien und Konzepte, die bei der Gestaltung von Systemen oder Anwendungen angewendet werden, um Struktur, Verständlichkeit und Wartbarkeit zu verbessern
UML-Diagramm	ist eine grafische Darstellung, die die Struktur und das Verhalten eines Systems mithilfe von Symbolen und Notationen visualisiert, um dessen Architektur, Interaktionen und Prozesse zu beschreiben
Open/Closed Principle	Bezeichnet dass, Klassen offen für Erweiterungen sein sollten, aber geschlossen für Modifikationen

Dummy Daten	sind fiktive oder generierte Informationen, die verwendet werden, um Platzhalter- oder Testdaten in einer Anwendung zu simulieren
JSON	(JavaScript Object Notation) ist ein leichtgewichtiges Datenformat, das verwendet wird, um strukturierte Daten in einem leicht lesbaren Textformat darzustellen, oft für den Datenaustausch zwischen verschiedenen Systemen
M:N Beziehung (many-to-many)	beschreibt eine Verknüpfung zwischen Entitäten in der Datenbank, in der jedes Element der einen Entität mehreren Elementen der anderen Entität zugeordnet werden kann und umgekehrt, was zu einer komplexen, indirekten Beziehung führt
IDE	ist eine Softwareanwendung, die Entwicklern Werkzeuge und Funktionen bereitstellt, um Code zu schreiben, zu debuggen, zu testen und zu verwalten, oft mit Editor, Compiler, Debugger und anderen Entwicklungs-Tools in einer integrierten Umgebung

Tabelle 5: Glossar

III. Tabellenverzeichnis

IV. Abbildungsverzeichnis

V. Literaturverzeichnis/Quellenverzeichnis

Internet Seite von Unternehmen, wo ich mein Praktikum durchgeführt habe:

https://www.empuron.com