Pflichtenheft

POS

1 Softwarevorraussetzungen

Haupt-Abhängigkeiten: flutter flutter_staggered_animations: ^1.1.1 cupertino_icons: ^1.0.8 shared_preferences: ^2.5.3 permission_handler: ^11.0.0 path_provider: ^2.1.1 flutter_map: ^6.2.1 latlong2: ^0.9.0 geolocator: ^10.1.0 flutter_native_splash: ^2.4.0 provider: ^6.1.3 pedometer: ^4.0.0 http: ^1.2.1

Dev-Abhängigkeiten: flutter_test flutter_lints: ^5.0.0 dcdg: ^4.1.0

2 Architektur

2.1 Klassen

2.1.1 Datenklassen (Schemes)

- Account (abstrakt)
  - Admin
  - User
- City (enum)
  - CityExtension
- Item
- Quest

2.1.2 Manager

- Inventory (quasi ItemManager)
- QuestManager
- UserManager

2.1.3 Controller

- AppController

2.1.4 Services

- APIService
- ItemService
- Locationchecker
- QuestService
- StepCounter
- UserService

2.1.5 Klassenhierarchie

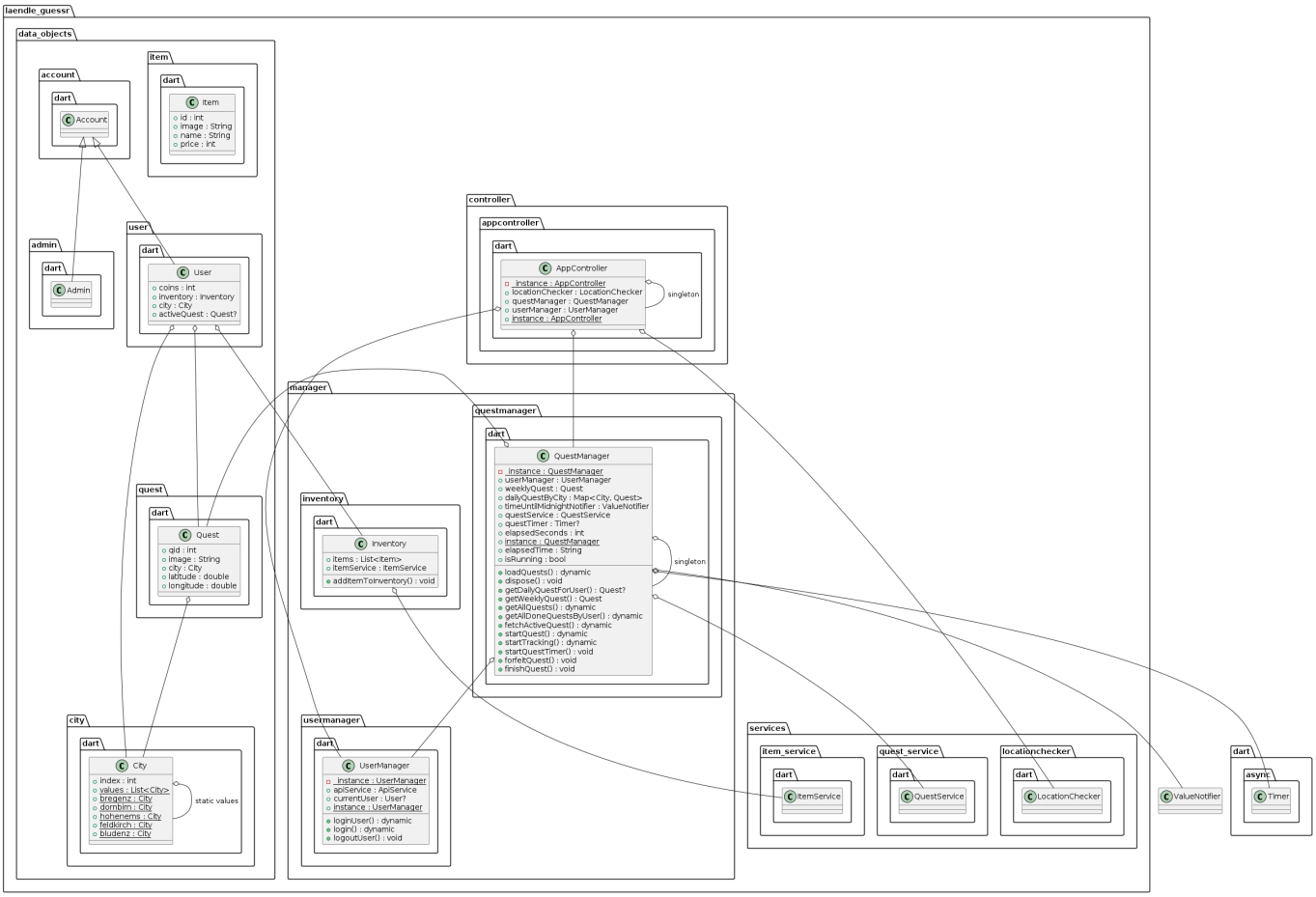
2.1.5.1 Beschreibung

Wir haben uns für folgende Hierarchie entschieden:

- Services, welche von mehreren Stellen im Programm erreichbar sind/benutzt werden
- Datenklassen, welche nur Schemen darstellen und quasi nichts können
- Manager, welche diese Datenklassen verwalten und ihnen die Funktionalität bieten
- Controller (AppController), welcher die Manager verwaltet

2.1.5.2 Visualisierung im Klassendiagramm

Automatisch generiert mithilfe eines Paketes in flutter.



2.2 Aufbau der UI

- MainPage, auf welcher die Quests für den angemeldeten User angezeigt werden (Startpunkt der App)
- MapsPage, auf welcher die Map angezeigt wird
- ProfilePage, auf welcher das Profil des Users angezeigt wird
- SignUpPage, auf welcher der User sich registriert
- LoginPage, auf welcher der User sich einloggt
- WillkommenPage, auf welcher der User am Start landet und aussuchen kann, ob er sich anmelden oder registrieren will

3 Umsetzung der Anforderungen

3.1 Organisatorisch

- zweier Gruppe: Bilal Ensar Bugday, Marlon Pichler
- Arbeit im Unterricht und zubhause (siehe GIT-Repository)

3.2 Technischer Inhalt

3.2.1 Must Haves

- Git Repository aktiv verwendet
    - Immer wieder committet und die Issues aktuell gehalten.
    - Kanban Board genutzt.
- Klassendiagramme vor dem Start der Programmierung
    - Wie per Teams ersichtlich, haben wir bevor wir mit dem Programmieren angefangen haben einen Sketch für die Klassenhierarchie bzw. Klassen gehabt.
- Grafische Anwendung:
    - Wie in Punkt 2.2 ersichtlich haben wir mindestens 3 Pages verwendet.
- Vererbung:
    - Wie in Punkt 2.1.5 ersichtlich haben wir eine abstrakte Klasse verwendet.
- API Dokumentation:
    - Wir haben das komplette FrontEnd mithilfe von KI dokumentiert.
- Unit Tests:
    - Wir haben uns für alle unsere relevanten Klassen Unit Tests geschrieben und damit unsere Klassen getestet

3.2.2 Nice To Haves

- Paralelle Programmierung:
    - Wir haben in unserem Programm oft Datenbankabfragen benutzt und somit auch auf Parallele Programmierung zurückgegriffen.
- Neues Framework bzw. neue Programmiersprache:
    - Wir haben unser Programm in Flutter bzw. Dart umgesetzt. Dies war für uns eine völlig neue und gewöhnungsbedürftige Erfahrung.

3.3 Beschreibung der Anwendung

Erstmal landet der User auf der "WillkommenPage". Auf dieser kann er sich dann entweder anmelden oder sich registrieren und wird auf die jeweilige Page navigiert ("SignUpPage"/"LoginPage").

Wenn er sich nun erfolgreich angemeldet hat kommt der User auf die "MainPage". Unten auf der Page befindet sich in userer gesamten Anwendung ab hier eine NavBar, mit welcher zwischen den Pages gewechselt werden kann.

So stehen einem nun alle Wege offen. Man kann Quests starten, auf der Map sehen wo man sich gerade befindet, im Shop mit seinen verdienten Coins Sammelstücke erwerben und vieles mehr!

4 Mögliche Probleme und ihre Lösung

- Natürlich hat bei unserem Projekt nicht alles auf Anhieb geklappt und wir hatten allfällige Schwierigkeiten. Spezifische Beispiele und deren Lösungen werden hier näher besprochen

4.1 Automatische Questverteilung

- Speziell bei der Questverteilung hatten wir recht große Schwierigkeiten, welche uns einige Gehirnzellen und Arbeitszeit abverlangt haben.
- Zuerst mussten wir uns hier nämlich überlegen, wie wir denn überhaupt die Quests automatisch verteilen, damit sich die DailyQuests und WeeklyQuests automatisch aktualisieren.
- Nach mehreren Stunden Überlegzeit und Hirnzerbrechen fanden wir dann endlich die Lösung: Cron Jobs
    - Mithilfe von Cron Jobs können in SupaBase nämlich automatisch in bestimmten Intervallen bzw. zu bestimmten Zeiten SQL-Snippets ausgeführt werden
    - Nun mussten wir nur noch das SQL-Snippet erstellen und testen und wir waren Good-To-Go!

4.2 Integrieren der Map in das Programm

- Anfangs wollten wir in unserem Programm die Karte von Google Maps verwenden, jedoch machte diese viele Probleme:
    - Die Verwendung dieser Karte ist sehr kompliziert, was uns an sich nicht aufgehalten hätte, ABER:
    - Später ist uns dann aufgefallen, dass die Verwendung mit der Zeit nicht mehr gratis sein würde. Also mussten wir uns eine Alternative suchen
- Diese Alternative war dann OpenStreetMap. Diese war sehr einfach zu bedienen und noch viel wichtiger: vollkommen gratis.

DBI

1 Softwarevorraussetzungen

Python3 pip

(Die Dependencies aus dem venv) aiohappyeyeballs==2.6.1 aiohttp==3.12.13 aiosignal==1.3.2 annotated-types==0.7.0 anyio==4.9.0 attrs==25.3.0 bcrypt==4.3.0 certifi==2025.6.15 charset-normalizer==3.4.2 click==8.2.1 clickclick==20.10.2 colorama==0.4.6 connexion==2.14.2 coverage==7.9.1 deprecation==2.1.0 Flask==2.1.1 Flask-Testing==0.8.1 frozenlist==1.7.0 gotrue==2.12.0 h11==0.16.0 h2==4.2.0 hpack==4.1.0 httpcore==1.0.9 httpx==0.28.1 hyperframe==6.1.0 idna==3.10 inflection==0.5.1 iniconfig==2.1.0 itsdangerous==2.2.0 Jinja2==3.1.6 jsonschema==4.24.0 jsonschema-specifications==2025.4.1 MarkupSafe==3.0.2 multidict==6.5.0 packaging==25.0 pluggy==1.6.0 postgrest==1.0.2 propcache==0.3.2 py==1.11.0 pydantic==2.11.7 pydantic_core==2.33.2 PyJWT==2.10.1 pytest==7.1.3 pytest-cov==6.2.1 pytest-mock==3.14.1 pytest-randomly==3.16.0 python-dateutil==2.9.0.post0 PyYAML==6.0.2 realtime==2.4.3 referencing==0.36.2 requests==2.32.4 rpds-py==0.25.1 setuptools==80.9.0 six==1.17.0 sniffio==1.3.1 storage3==0.11.3 StrEnum==0.4.15 supabase==2.15.3 supafunc==0.9.4 swagger-ui-bundle==0.0.9 tomli==2.2.1 typing-inspection==0.4.1 typing_extensions==4.14.0 urllib3==2.5.0 websockets==14.2 Werkzeug==2.2.3 yarl==1.20.1

2 Architektur

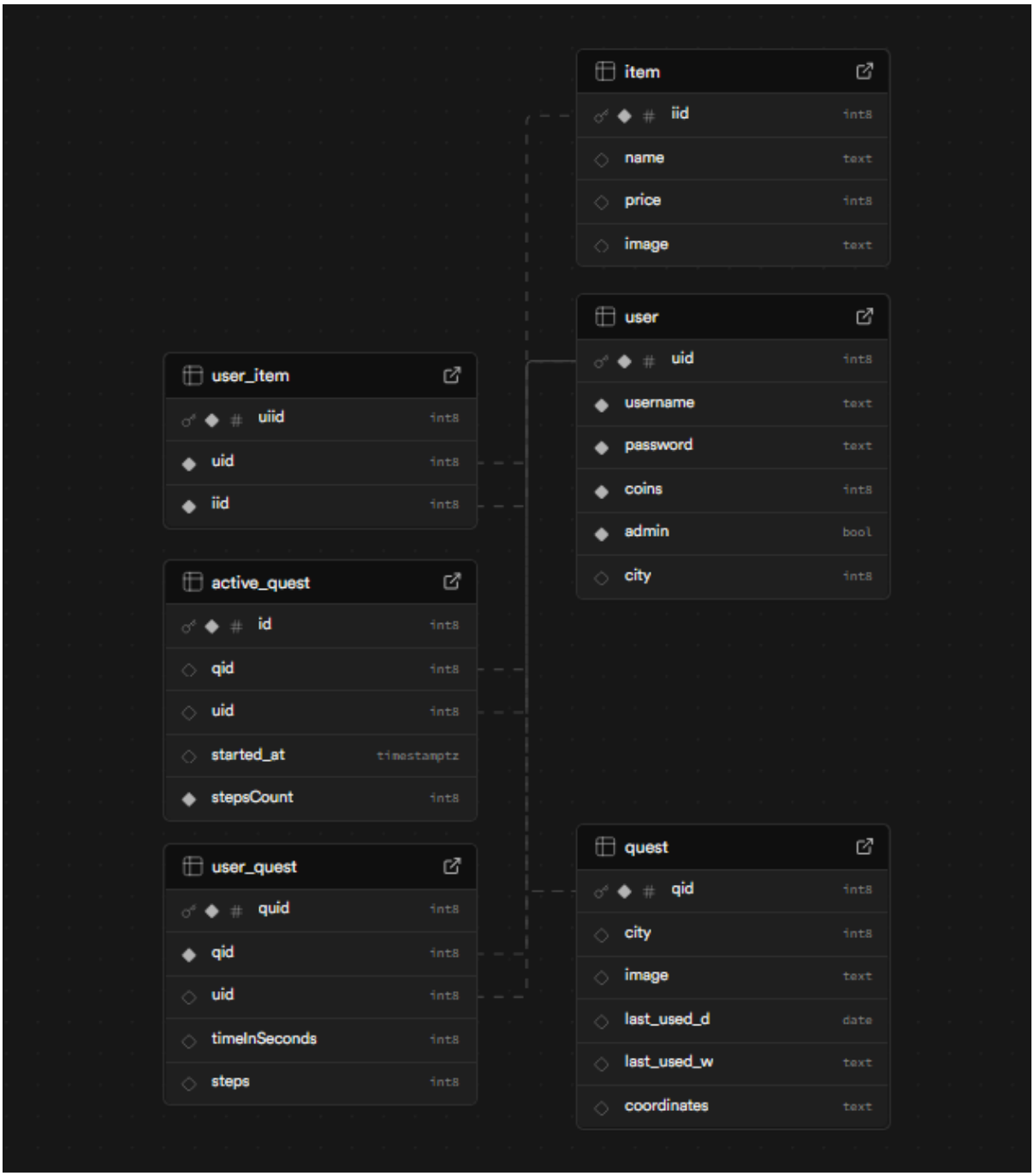
- Allgemein war unser Backend so aufgebaut:
    - Frontend (API-Service) -> REST-API -> Supabase-API -> Supabase-Datenbank
- Das Frontend schickt mithilfe des API-Services http-Requests an unsere REST-API, welche dann Anfragen an die Supabase-Api bzw. die Supabase-Datanbank schickt.

2.1 REST-API

- Wir haben für unsere REST-API Swagger (in Python) verwendet und mit "Docker" auf unserem yaml-Schema einen Server generiert.
- In diesem gibt es dann:
    - Controller, welche die Funktionalitäten der Endpunkte darstellen
    - Und Models, welche von den Controllern benutzt werden
- In den Controllern haben wir wie folgt unterteilt:
    - user_controller.py, dieser managed alle Endpunkte, welche mit Usern zu tun haben
    - item_controller.py, dieser managed alle Endpunkte, welche mit Items zu tun haben
    - quest_controller.py, dieser managed alle Endpunkte, welche mit Quests zu tun haben
    - security_controller.py, dieser managed die Sicherheit unserer REST-API durch beispielsweise Authenticaiton
- In unseren Models gibt es einige Baupläne für Objekte wie beispielsweise:
    - quest_user.py
    - oder user.py

2.2 Datenbank

- Wir haben uns dazu entschieden unsere Datenbank über SupaBase zu erstellen und laufen zu lassen. Dies hatte mehrere Gründe:
    - Sehr einfache Kommunikation zu Programmen
    - Sehr gute Dokumentation
    - Gratis
    - Gutes Scaling
    - ...
- Im Endeffekt sah unsere Datenbankstruktur wie folgt aus:



3 Umsetzung der Anforderungen

3.1 Organisatorisch

- zweier Gruppe: Bilal Ensar Bugday, Marlon Pichler
- Arbeit im Unterricht und zubhause (siehe GIT-Repository)

3.2 Technischer Inhalt

3.2.1 Must-Haves

- Mind. 3 Tabellen in 3.NF:
  - Wir haben insgesamt, in welchen wir (hoffentlich) die dritte Normalform beachtet haben.
- Queries mit Select, Joins Aggregation
  - Haben wir alles verwendet (siehe REST-API)

- Schreibender und lesender Zugriff:
  - Wir haben sowohl GET und POST, als auch PUT und DELETE für unsere Vorhaben gebraucht
- mind. 2 Rollen
- SQL-Datenbank:
  - Wir haben eine globale Datenbank verwendet (siehe 2.2)
- REST-Interface:
  - Wir haben mit Swagger (in Python) eine REST-API erstellt (siehe 2.1)
- API-Dokumentation + JSON Schema für Objekte:
  - Wir haben mithilfe von KI eine komplette API-Dokumentation gemacht und haben in unseren Models (siehe 2.1) Schemen erstellt
- Unit Tests:
  - WIr haben Unit Tests für die relevanten Funktionen geschrieben und unsere funktionen damit getestet
- Logging:
  - Wir haben das Projekt komplett geloggt

3.2.2 Nice-To-Haves

- Globale Datenbank (Supabase)

3.3 Beschreibung der Anwendung

In unserer REST-API gibt es viele verschiedene Endpunkte, welche alle Daten aus unserer Datenbank auslesen/löschen/verändern/hinzufügen. Diese sind nach Kategorien in Files eingeteilt und auf Funktionalität getestet.

4 Mögliche Probleme und ihre Lösung

- Natürlich hat bei unserem Projekt nicht alles auf Anhieb geklappt und wir hatten allfällige Schwierigkeiten. Spezifische Beispiele und deren Lösungen werden hier näher besprochen

4.1 Supabase

- Trotz der vielen Vorteile gab es bei Supabase für uns trotzdem einige Probleme.
  - Beispielsweise waren die Server von Supabase für mehrere Tage down und wir konnten somit nicht an unserem Backend weiterarbeiten (Blind-Coding ohne testen bringt nichts)
  - Zudem gab es teilweise Diskrepanzen in den Fehlern. An einem Tag kann alles einwandfrei funktionieren und am nächsten Tag funktioniert der 1-zu-1 gleiche Code nicht mehr aufgrund eines Fehlers in SUpabase selbst.
- Lösung gibt es hier keine, außer abzuwarten und Däumchen zu drehen bis die Server wieder laufen

4.2 Rollen

- Da wir zuvor noch nie mit Rollen gearbeitet haben, war es sehr schwer diese umzusetzen und wir sind auf viele Fehler und unerwartetes Verhalten gestoßen