Entwicklerdokumentation

Mobiles Logbuch für Modellflugplatz (E09)

Inhaltsverzeichnis

1. Entwurfsdokumentation . . . 1
   1.1. Architecture Notebook . . . 1
      1.1.1. Zweck . . . 1
      1.1.2. Architekturziele und Philosophie . . . 1
      1.1.3. Annahmen und Abhängigkeiten . . . 1
      1.1.4. Architektur-relevante Anforderungen . . . 2
      1.1.5. Entscheidungen, Nebenbedingungen und Begründungen . . . 2
      1.1.6. Architekturmechanismen . . . 3
      1.1.7. Wesentliche Abstraktionen . . . 4
      1.1.8. Schichten oder Architektur-Framework . . . 4
      1.1.9. Architektursichten (Views) . . . 5
   1.2. Design . . . 11
      1.2.1. Diagramm . . . 11
      1.2.2. Verwendete Technologien . . . 12
      1.2.3. Schnittstellen . . . 13
      1.2.4. Komponenten und Struktur . . . 16
2. Softwaredokumentation . . . 18
3. Am Projekt entwickeln . . . 19
   3.1. Entwicklungsumgebung . . . 19
      3.1.1. Backend . . . 19
      3.1.2. Frontend . . . 19
      3.1.3. Datenbank . . . 19
   3.2. Starten der Anwendung in der Entwicklungsumgebung . . . 20
      3.2.1. Konfiguration nach dem ersten Start . . . 20

1. Entwurfsdokumentation

1.1. Architecture Notebook

1.1.1. Zweck

Dieses Dokument beschreibt die Philosophie, Entscheidungen, Nebenbedingungen, Begründungen, wesentliche Elemente und andere übergreifende Aspekte des Systems, die Einfluss auf Entwurf und Implementierung haben.

1.1.2. Architekturziele und Philosophie

Die Software ist eine Webanwendung, welche zum Protokollieren von Flügen auf einem Modellflugplatz dient. Diese Anwendung muss von allen private Endgeräten der Piloten aufrufbar sein - und ist somit auch **öffentlich im Internet zugänglich**. Die Anwendung sollte eine deutlich höhere Toleranz an gleichzeitig aktiven Nutzern aufweisen als die meisten Modellflugclubs an Mitgliedern haben.

Dabei ist es wichtig, dass eine gesicherte Verbindung aufgebaut werden kann. Weiterhin dürfen alle Kernfunktionalitäten (bis auf die Anmeldung) erst nach der erfolgreichen **Authentifizierung der Piloten***über ihren Nutzeraccount verfügbar sein.

Da das Logbuch eines Modellflugplatz auf **gesetzlichen Regelungen** basiert, ist es zwingend erforderlich dass diese eingehalten werden. Darüber hinaus müssen die ohnehin geltenden Bestimmungen, wie unteranderem die DSGVO, beachtet werden.

Da der Modellflugplatz ein Outdoor-Gelände ist, muss die Anwendung vor allem auf **mobile Endgeräte** abgestimmt sein. Da sich diese Endgeräte in ihren Spezifikationen teils stark unterscheiden, muss die Anwendung flexibel skalieren und schmale Bildschirmbreiten korrekt handhaben. Da manche Bedienungsschritte der Anwendung außerhalb der WLAN-Reichweite des Clubs stattfinden, muss die Anwendung möglichst **wenig mobile Daten** verbrauchen.

Zusätzlich zu dem allgemeinen Ziel der **einfachen Bedienbarkeit** kommt der Umstand, dass viele Modellflugclubs auch eine Reihe an älteren Mitgliedern haben, sodass kleine UI-Elemente und komplexe Strukturen vermieden werden sollten. Die Hauptfunktionalität der Protokollierung sollte mit wenigen Klicks möglich sein. Wo es möglich ist sollen Standardwerte bereits eingetragen sein.

Es ist davon auszugehen, dass der Auftraggeber oder andere Modellflugclubs das System in Zukunft **erweitern möchten**, weswegen eine nachvollziehbare Struktur und Dokumentation ins Gewicht fallen.

1.1.3. Annahmen und Abhängigkeiten

Annahmen:

- Internetverbindung
    - Auf dem Modellflugplatz besteht eine gute Abdeckung an mobilen Daten.

    - Im Clubgebäude gibt es WLAN.
- Smartphonefeatures (oder anderes Endgerät)
    - Jeder Pilot hat ein Smartphone mit HTML5 und JavaScript fähigem Browser.
    - Das Smartphone des Piloten verfügt über eine Kamera.
- Die verwendeten Frameworks sind stabil und werden in Zukunft durch ihre Autoren gewartet.
- Der Verein des Modellflugplatz verfügt über einen Admin (Person).

Abhängigkeiten:

- Das Projekt ist darauf angewiesen, dass die Verwendeten Frameworks (für Frontend und Backend) und die Datenbank in Zukunft gepflegt werden.
- Der Verein benötigt einen möglichst dauerhaft laufenden Linux Server, sowie einen Admin welcher diesen in Stand hält.

1.1.4. Architektur-relevante Anforderungen

Funktional

- **SWFA-1:** Das System muss Accountdaten sowie Protokolle dauerhaft speichern
- **SWFA-2:** Piloten müssen ihre eigenen Protokolle bearbeiten können, aber nicht die anderer Nutzer
- **SWFA-3:** Das System muss die Daten zu den Protokollen korrekt und vollständig erfassen und speichern
- **SWFA-4:** Die Flugleiter-Regelungen müssen eingehalten werden, ein Admin oder ein Protokolleur soll das überprüfen können

Effizienz (Performance)

- **NFAP-1:** Das System sollte in der Lage sein, Protokolle sofort abzuspeichern und auszugeben
- **NFAP-2:** Seitenwechsel in der Anwendung müssen schnell erfolgen, sodass kein gravierender Unterschied zu einer nativen Mobile-App feststellbar ist

Wartbarkeit (Supportability)

- **NFAS-1:** Das System sollte mit wenig Zeitaufwand administrierbar sein

1.1.5. Entscheidungen, Nebenbedingungen und Begründungen

1. Wir nutzen für das Frontend das Webframework Vue.js, da
    a. es weit verbreitet ist, gut dokumentiert und höchstwahrscheinlich auch in Zukunft gut gepflegt
    b. wir durch Vorgängerprojekte inspiriert wurden
    c. die Einstiegshürde gering ist
    d. kein großer Eingriff in die Standard HTML Struktur stattfindet

2. Wir nutzen TypeScript im Frontend, da
    a. somit häufige Fehlerquellen aus JavaScript vermieden werden
    b. die IDE Unterstützung besser ist
3. Wir nutzen die relationale SQL Datenbank PostgreSQL, da
    a. alle Teammitglieder und der Auftraggeber mit der Query-Language vertraut sind
    b. die zukünftige Wartung wahrscheinlich sicher ist
    c. diese Software als robust und gut optimiert gilt
    d. die Einrichtung und Wartung einfach und gut dokumtiert ist, und es viel Hilfe im Internet gibt
4. Wir nutzen Kotlin mit Ktor im Backend, da
    a. die Erfahrung aus Java mitgenutzt werden kann
        i. die Sprache aber flexibler als Java ist
    b. das Framework von Kotlin offiziell gepflegt wird
    c. manche Teammitglieder dort bereits Erfahrung haben
5. Wir nutzen standard SQL Queries für die Datenbankabfragen und SQLDelight um diese type-safe aufzurufen, da
    a. somit keine direkte Abhängigkeit zu einer ORM-Bibliothek besteht
    b. die Anfragen leichter von Personen ohne Kenntnissen in anderen Programmiersprachen bearbeitet werden können
6. Wir definieren serialisierbare Klassen anhand von Typen, um somit zwischen Frontend und Backend zu kommunizieren
7. Wir nutzen die Component-Architektur im Frontend, um die einzelnen UI-Elemente und deren Logik zu trennen und leichter wiederverwertbar zu machen
8. Wir nutzen Browser-Sessions, um die Nutzer der Anwendung nach dem Anmelden zu identifizieren

1.1.6. Architekturmechanismen

Doku "Concept: Architectural Mechanism"

1. Die Daten der Anwendung müssen persistent sein (SWFA-1) und jeder Nutzer darf nur seine eigenen Daten bearbeiten (SWFA-2). Daher wird eine Datenbank benötigt, bei der unsere Entscheidung wie im Punkt "Entscheidungen" erläutert für eine SQL-Datenbank ausgefallen ist. Es ist noch nicht klar, ob das finale Projekt SQLite oder PostgreSQL verwenden wird.
2. Nach SWFA-2 wird eine Nutzerverwaltung und Loginfunktionalität benötigt. Dabei haben wir die Möglichkeit der eigenen simplen Implementierung oder OAuth mit third-party Services wie Google, Facebook usw. Vorerst werden wir nur eine eigene Nutzerverwaltung verwenden, da der Verein einen Admin hat, welcher die Nutzeraccounts manuell erstellt und somit viele Sicherheitsprobleme nicht auftreten können. Die Möglichkeit der 2-Faktor-Authentifizierung ziehen wir in Betracht.

3. Da die Webanwendung sehr viel mit sich veränderndem State arbeitet, werden wie die Reactivity Funktionen von Vue.js in Kombination mit der Composition API verwenden. Dadurch wird es in der Entwicklung deutlich einfacher das UI immer aktuell zu halten.
4. Damit der Server seine Anforderungen durchsetzen kann, wie zum beispiel SWFA-4, ist der Client ständig über eine WebSocket Verbindung mit dem Server verbunden. Somit kann der Server Hinweise und Anfragen verschicken, wie z.B. dass ein neuer Flugleiter benötigt wird.

1.1.7. Wesentliche Abstraktionen

Typen: Typen werden im Frontend und Backend zur Laufzeit benötigt um die Programmdaten an den gewünschten Stellen zur Verfügung zu haben.

- Protokoll: enthält alle gesetzlich vorgeschrieben Eckdaten (siehe Use Case 03)
- User: enthält Session, Name, weiter Nutzerinformationen
  - Flugleiter: enthält Informationen zum Zeitraum, in dem die Rolle ausgeübt wurde
  - Admin: wie User, aber die Session enthält einen Admin Vermerk
- Modell: Information über ein spezielles Modellflugzeug, Bild
- Flugzeugtyp: allgemeiner Flugzeugtyp, Beschreibung

Routes: Werden vom Backend offeriert, damit das Frontend (mit valider Session) Daten abfragen und manipulieren kann.

Components: Sind gekapselte UI-Element eim Frontend, welche Style, Logik und State enthalten.

1.1.8. Schichten oder Architektur-Framework

Frontend

Single-File-Components

Die Trennung von Funktionen und Inhalten ist ein wichtiges Prinzip in der Softwareentwicklung. In traditionellen Web-Entwicklungsansätzen wird dies oft durch die Trennung von HTML, CSS und JavaScript erreicht.

Im Kontext zunehmend komplexer Frontend-Anwendungen ist diese Trennung jedoch nicht immer sinnvoll. Das Component-Pattern bietet eine alternative Lösung, die die Vorteile der Trennung von Funktionen und Inhalten mit den Anforderungen an die Wartbarkeit komplexer Anwendungen vereint.

Beim Component-Pattern werden die UI-Elemente in lose gekoppelte Komponenten unterteilt. Jede Komponente übernimmt eine bestimmte Aufgabe, z.B. die Darstellung eines Buttons oder die Eingabe eines Formulars. Die Vorlage, Logik und Stile einer Komponente sind intrinsisch miteinander verbunden. Dies macht die Komponente kohärenter und wartbarer.

Die Komponenten können zu komplexeren UI-Strukturen zusammengesetzt werden. Dies ermöglicht eine flexible und wiederverwendbare Architektur.

Backend

Funktionaler Stil

Die einzelnen Routes auf die das Backend reagiert werden vollständig funktional gehandhabt. Dies ermöglicht zusätzlich eine einfache Parallelisierung.

Des weiteren wird somit der Control Flow nachvollziehbarer.

1.1.9. Architektursichten (Views)

Logische Sicht

Die logische Sicht betrachtet Klassen und Schnittstellen, welche bei dem System miteinander interagieren bzw. Einfluss aufeinander haben. Diese sind architekturrelevant und spiegeln sich unteranderem in den Typen wieder. Der Großteil der Interaktion geht hier von dem Pilot aus, was sich mit dem Ansatz deckt, dass dieser im zentralen Fokus unserer Anwendung und Nutzeroberfläche steht.

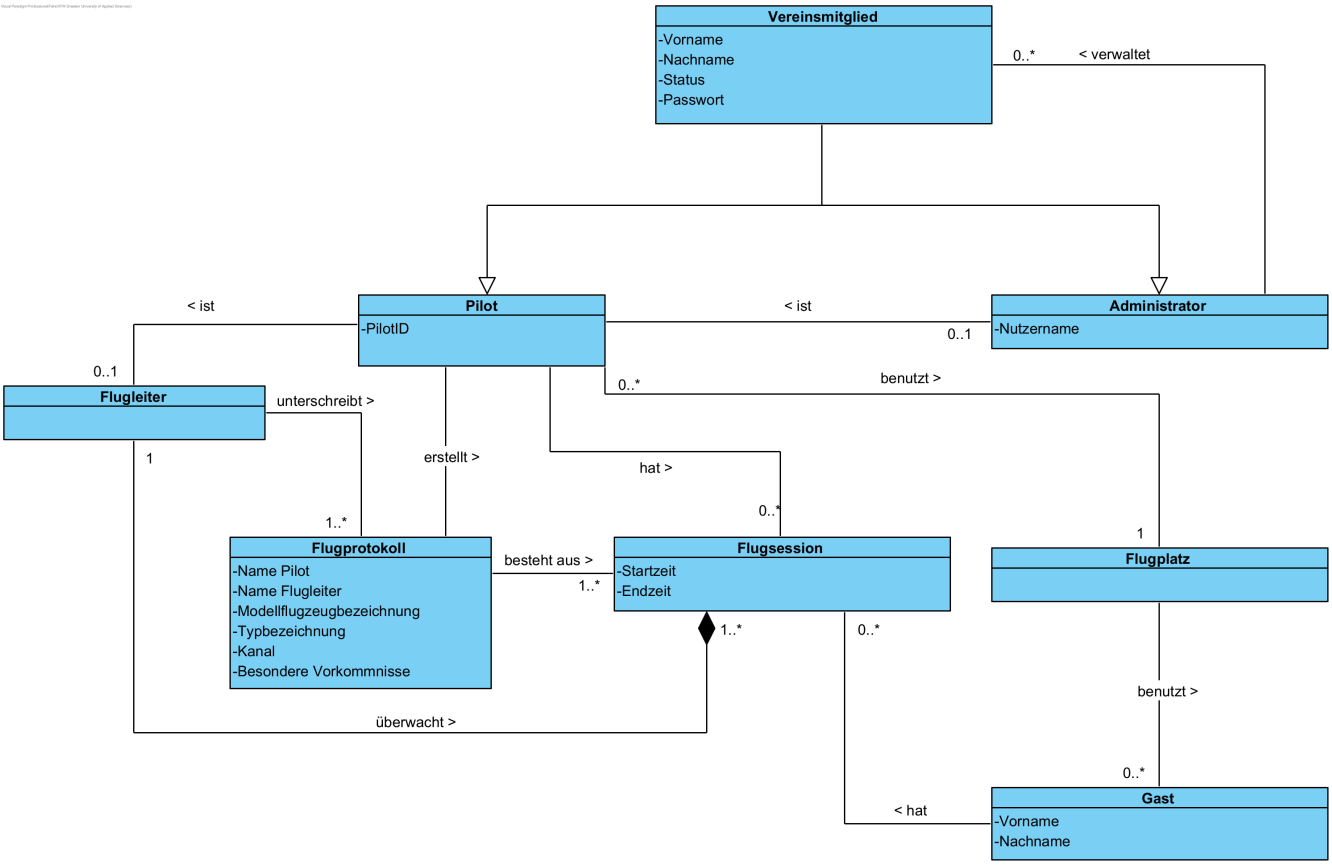


Abbildung 1. Domainmodell

Physische Sicht (Betriebssicht)

Aus physischer Sicht steht klar das Backend im Mittelpunkt, da es Wetterstation und Datenbank zusammenfasst, die meiste Logik handhabt und bei jeglicher Interaktion mit einbezogen wird.

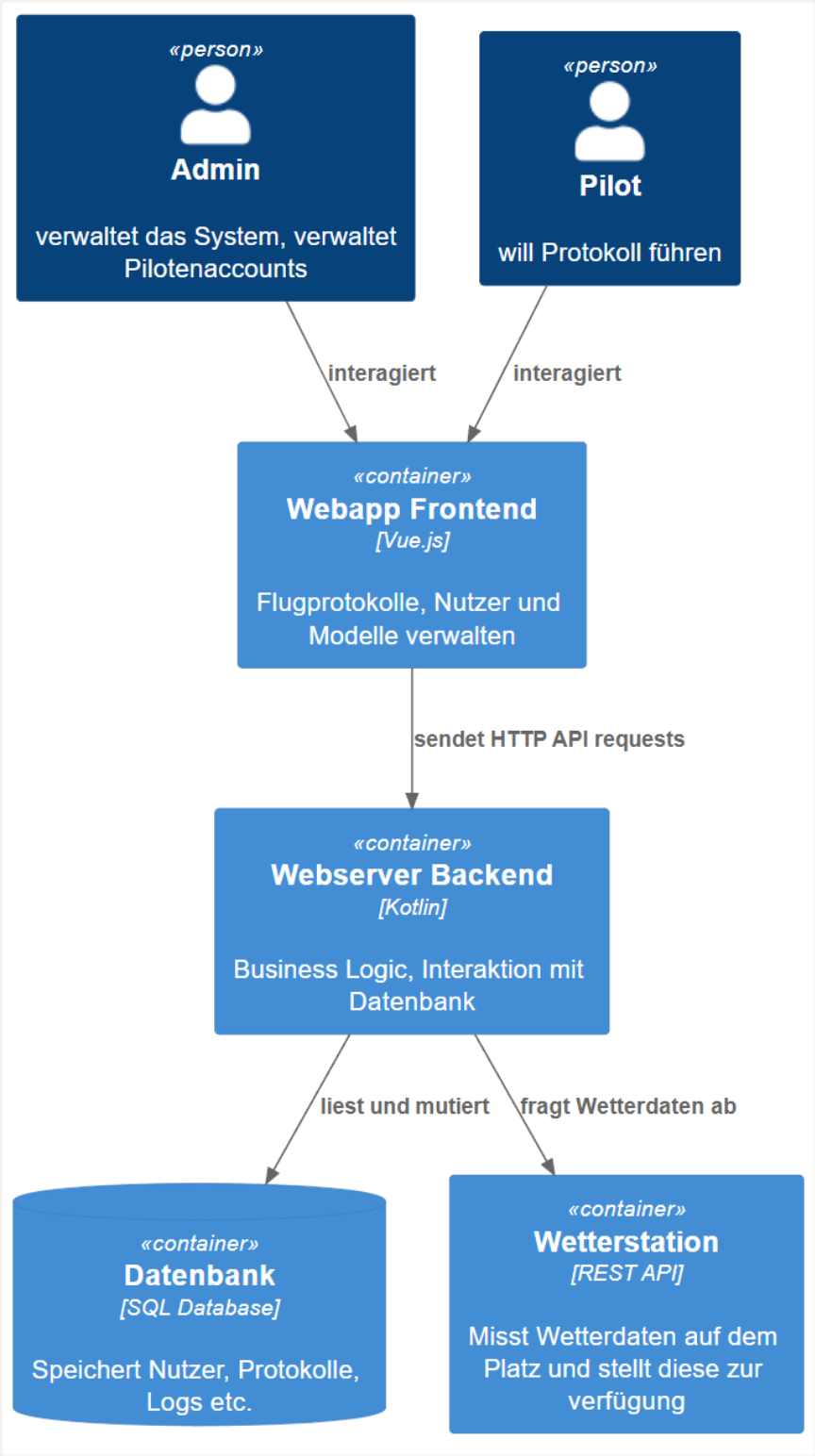


Abbildung 2. C4 Modell (Level 2)

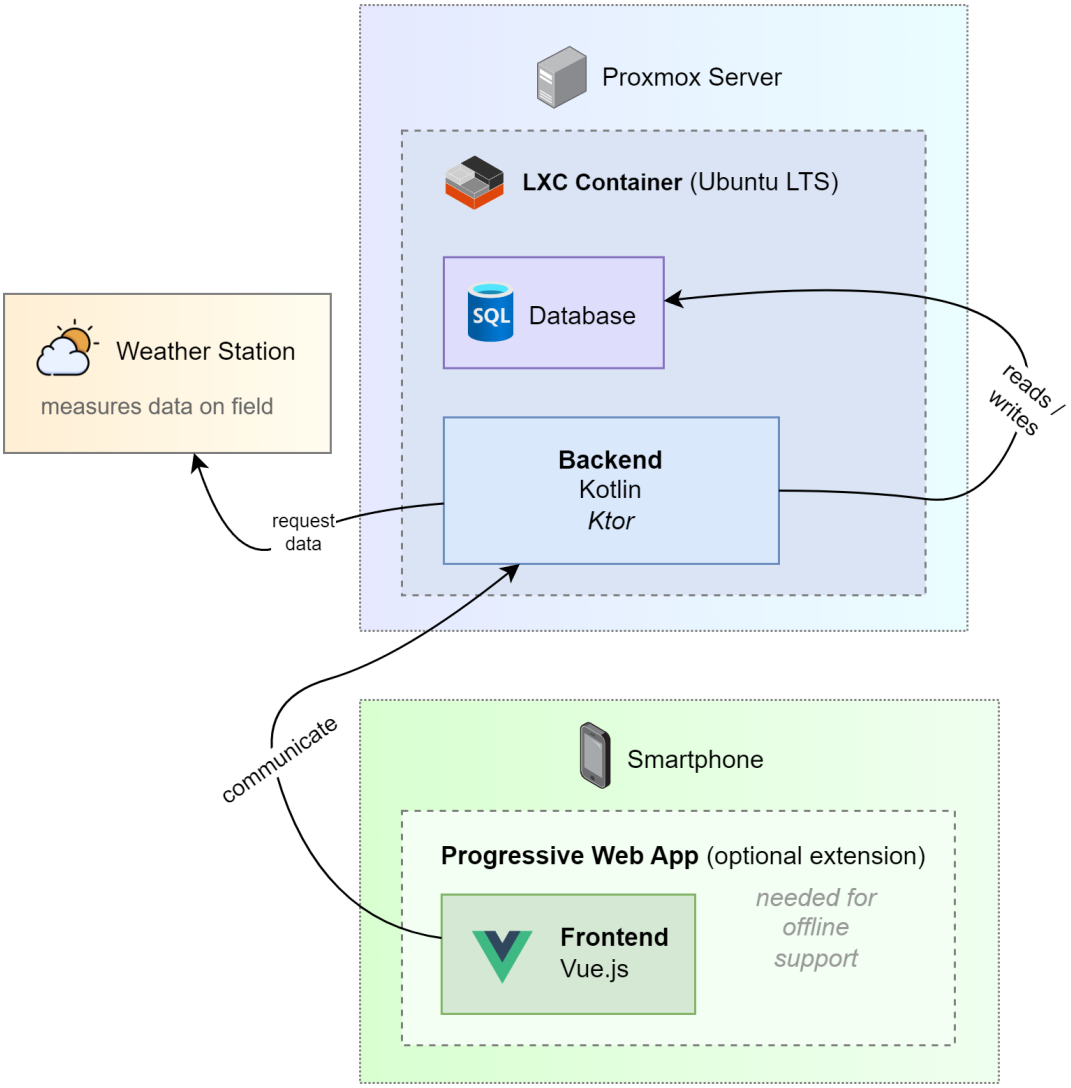


Abbildung 3. Informelles Architektur Diagramm (Aus Protokoll eines Auftraggebermeetings)

Use cases

Alle von uns identifizierten Use Cases haben eine Auswirkung auf die Architektur.

So unter anderem:

- **UC01**: Account registrieren, **UC07**: Accountdaten modifizieren, **UC08**: Account deaktivieren
  - Der Account bzw. die Änderungen werden in der Datenbank gespeichert
  - Der Admin, welcher den Account erstellt, muss authentifiziert sein
- **UC03**: Protokoll anlegen, **UC04**: Protokoll einsehen, **UC09**: Protokoll bearbeiten
  - Das Protokoll bzw. die Änderungen werden in der Datenbank gespeichert
- **UC05**: Flugleiter bestimmen
  - Es darf nur einen Flugleiter geben
  - Es muss über die Notwendigkeit eines Flugleiters informiert werden
    - Live-Update des Frontends
- **UC02**: Account Login, **UC06**: Account abmelden
  - Es muss eine Session Authentication stattfinden
  - Das Passwort solche sicher übertragen und gespeichert werden
- **UC10**: Tagesprotokoll erstellen
  - Es muss eine effiziente Aggregierung möglich sein

Somit ist für die Architektur das gesamte Use-Case-Diagramm relevant.

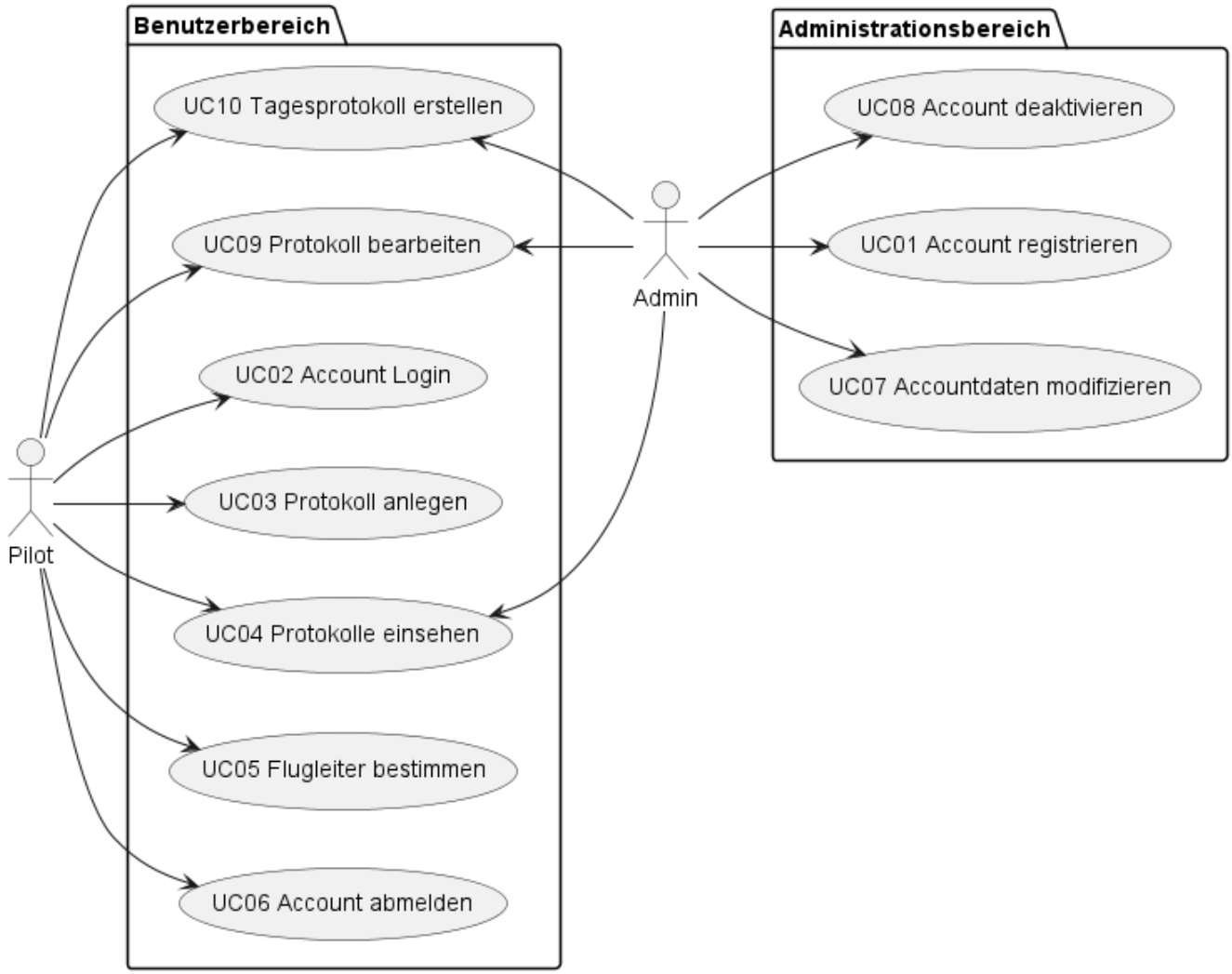


Abbildung 4. Use-Case-Diagramm

Komponentendiagramm

Für die abgebildeten Interfaces ergibt sich folgende Legende:

- durchgehende Linie → stellt Interface bereit
- gestrichelte Linie → benötigt Interface

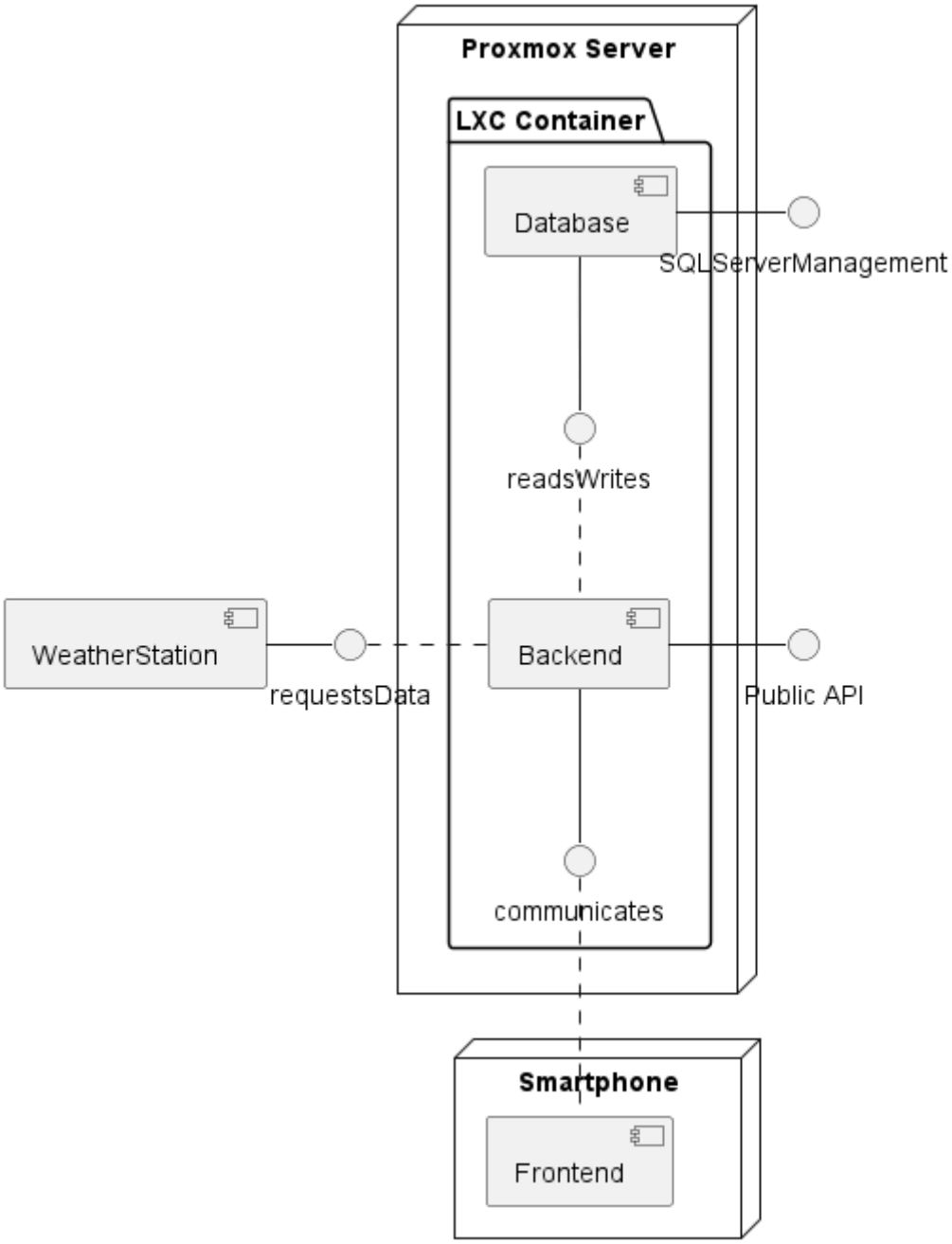


Abbildung 5. Component-Diagramm

1.2. Design

Dieses Dokument beschreibt den Feinentwurf von Komponenten, also Schnittstellen, Paketen, Klassen usw.

1.2.1. Diagramm

Das ER-Diagramm zeigt die Beziehungen zwischen den Entitäten des Systems. Jede Entität ist eine Tabelle in der Datenbank.

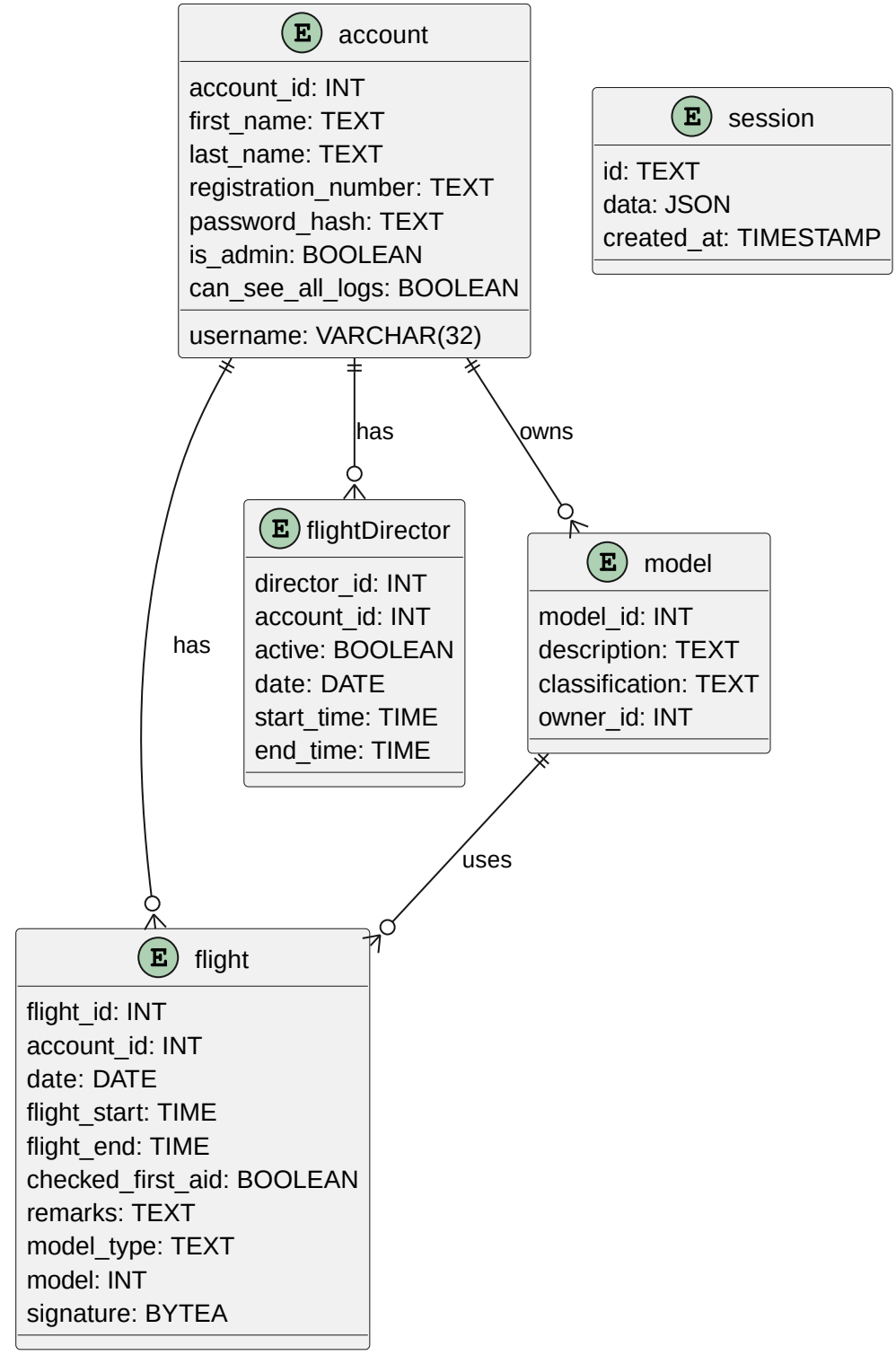


Abbildung 6. ER-Diagramm für das Datenbankschema

Hier noch eine alternative Ansicht aus IntelliJ:

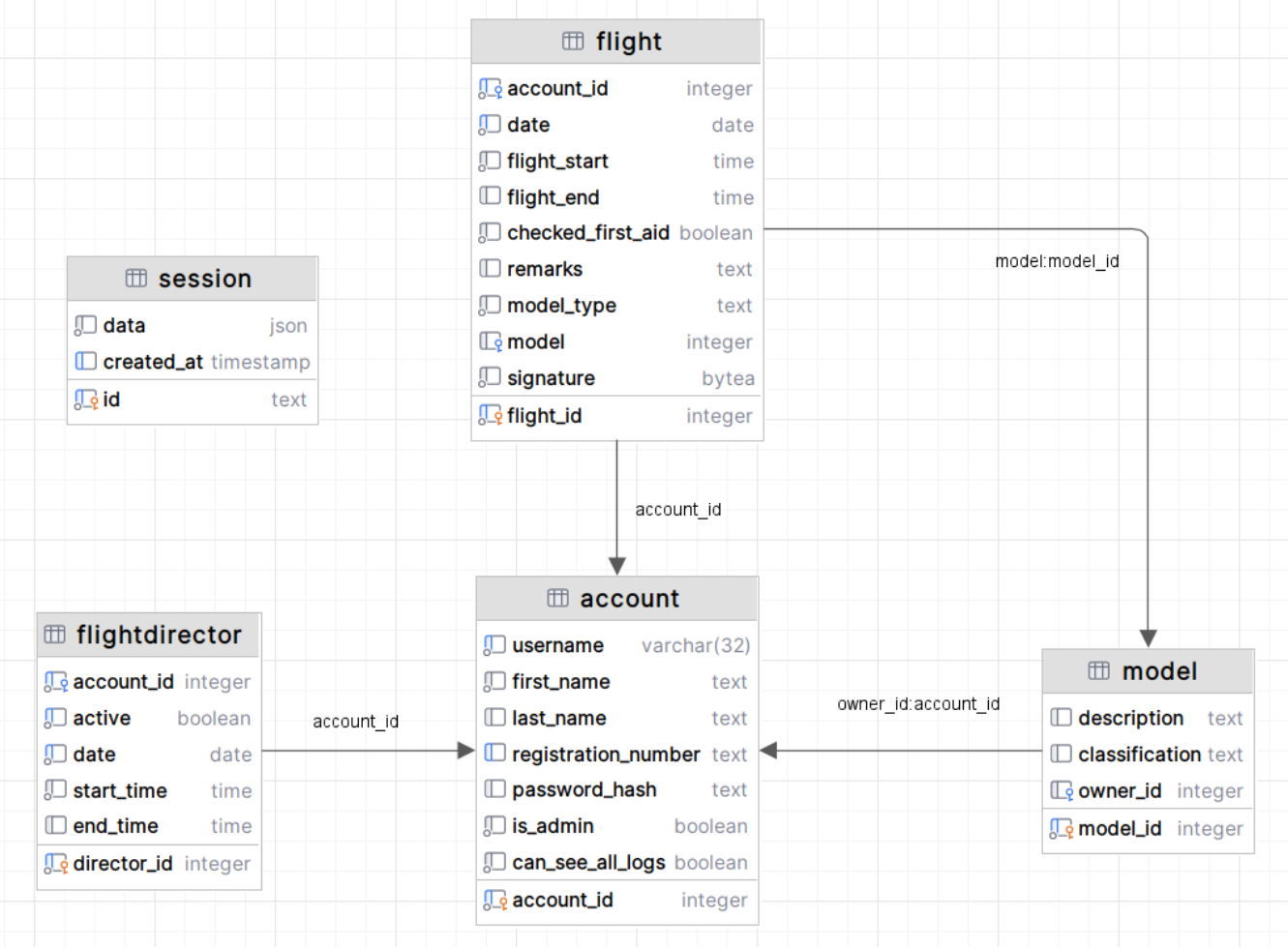


Abbildung 7. ER-Diagramm mit IntelliJ generiert (nachträglich)

1.2.2. Verwendete Technologien

Aus dem Entwurf der Architektur wurde sich nun final auf die folgenden Technologien festgelegt:

Backend

- Kotlin
  - kotlinx.serialization (JSON Serialization)
  - kotlinx.datetime (Date and Time)
  - Gradle (Build Tool)
- Ktor (Web Framework)
  - Websockets (Live App State Updates)
- SQLDelight (SQL Queries)
  - PostgreSQL Dialect
  - HikariCP (Connection Pooling)
- slf4j (Logging)

Frontend

- Vue.js
  - Vite (Build Tool)
  - Vue Router (Routing)
  - Pinia (State Management)
- TypeScript
  - pnpm (Package Manager)

1.2.3. Schnittstellen

Rest-API

Die Rest-API wird über Ktor bereitgestellt. Die Routen werden deklarativ definiert, ausgehend von der `routing` Funktion in `Application.kt`, von welcher sofort mittels Extension-Functions in einen separaten Scope in einer anderen Datei gewechselt wird.

Diese API haben wir mittels einer OpenAPI Spezifikation definiert. Dadurch ist der genaue Aufbau jederzeit allen Teammitgliedern klar, und die Schemata der Requests und Responses sind immer klar.

Swagger

Diese Spezifikation kann auch in Swagger UI geladen werden, um eine interaktive Dokumentation zu erhalten. Dies geht sowohl lokal auf dem eigenen Rechner, als auch durch das Backend selbst, welches die Spezifikation unter `/api` bereitstellt.

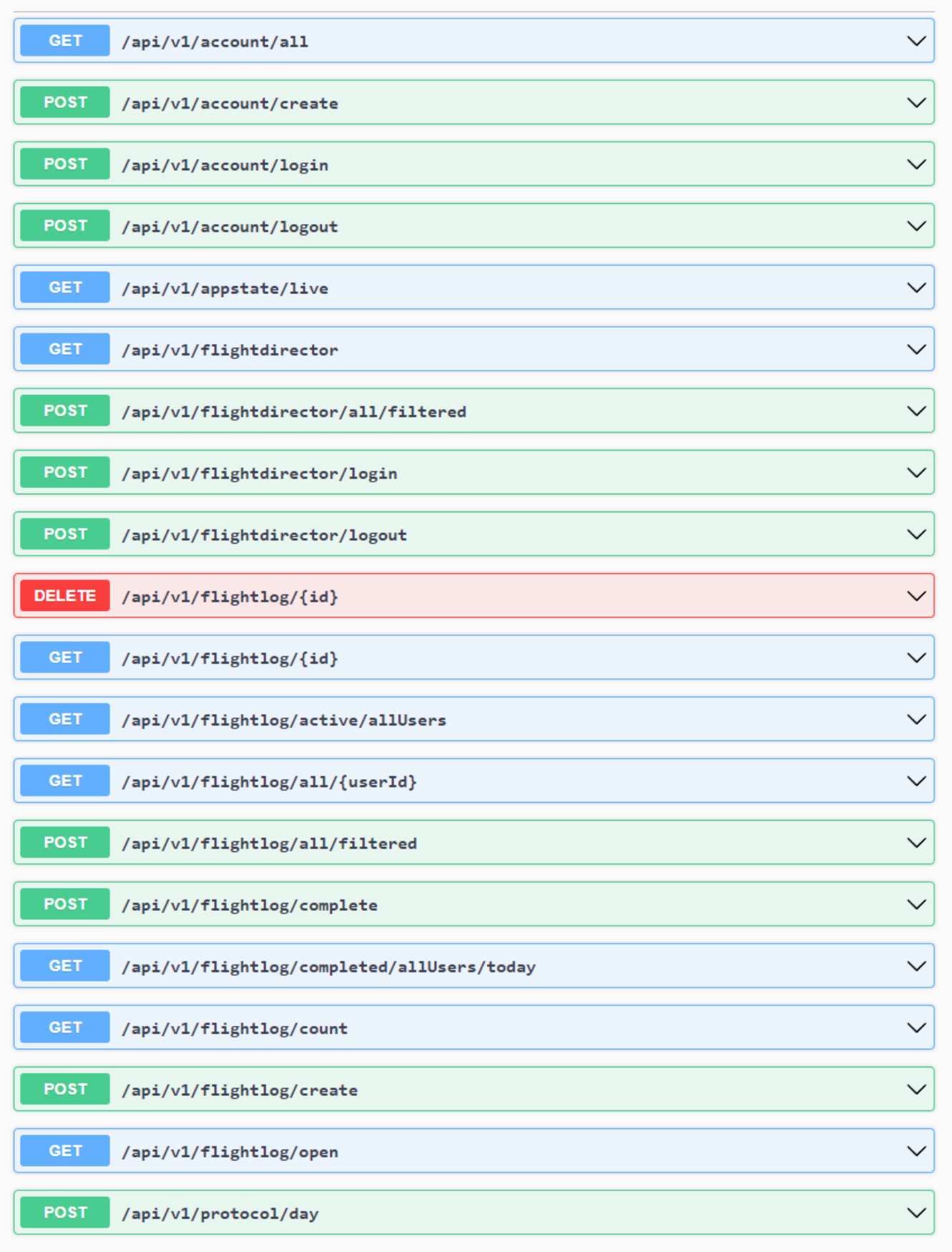


Abbildung 8. Übersicht aller Routen in Swagger

In Swagger können dann die spezifischen Routen aufgeklappt werden, um genauere Informationen zu erhalten:

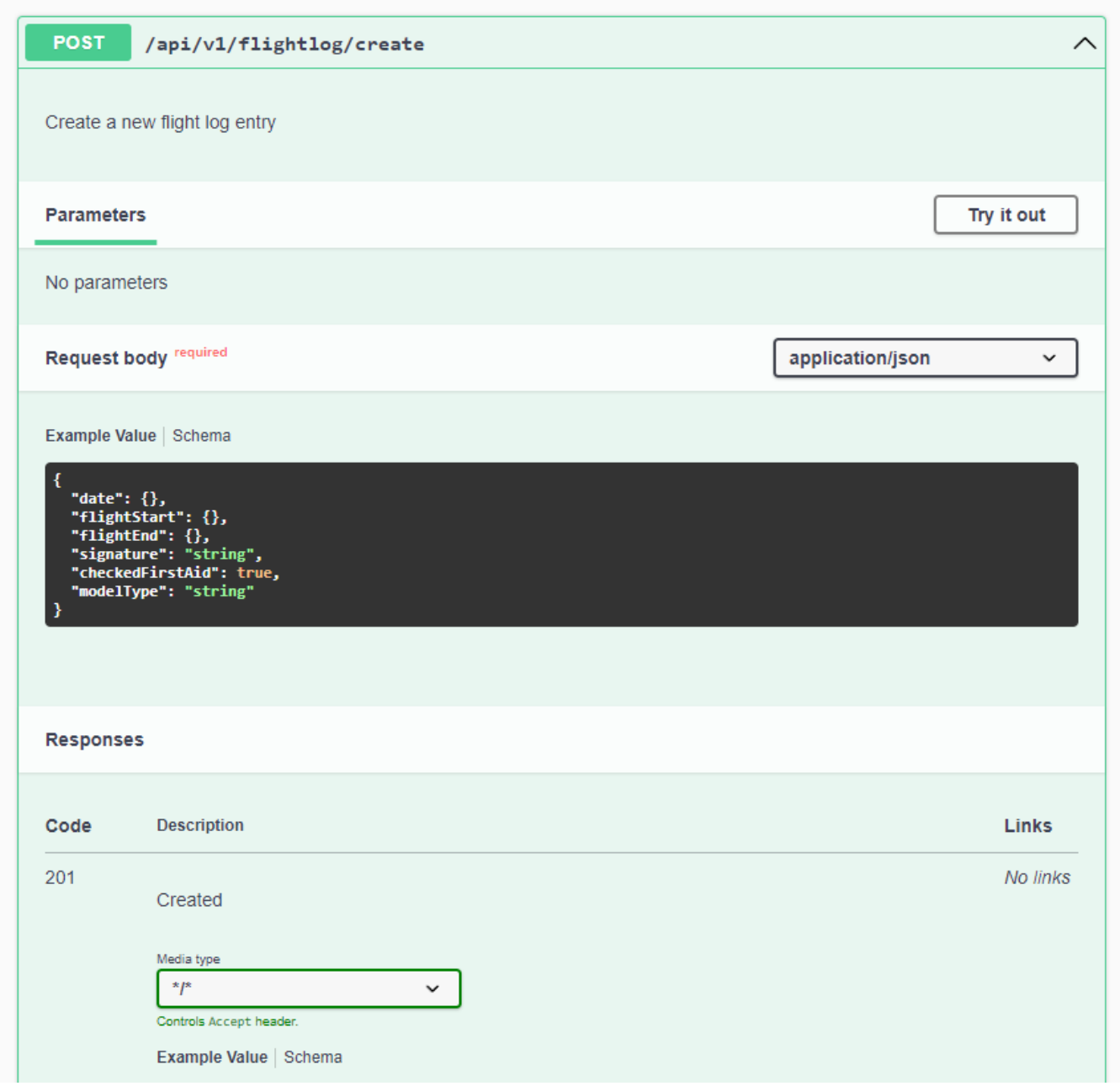


Abbildung 9. Übersicht aller Routen in Swagger

Genaue Spezifikation

Die genaue Spezifikation ist unter https://jakobkmar.github.io/E09-modellflug-logbuch/swagger zu finden, und wird zusammen mit der Softwaredokumentation auf GitHub Pages gehosted.

1.2.4. Komponenten und Struktur

Aufbau des Backends

Das Backend nutzt vor allem einen deklarativen und funktionalen Programmierstil, wodurch existierenden Klassen und Funktionen klein und spezialisiert sind.
Hauptsächlich wird bei Kotlin in Dateien strukturiert, welche jeweils in Paketen angeordnet sind.

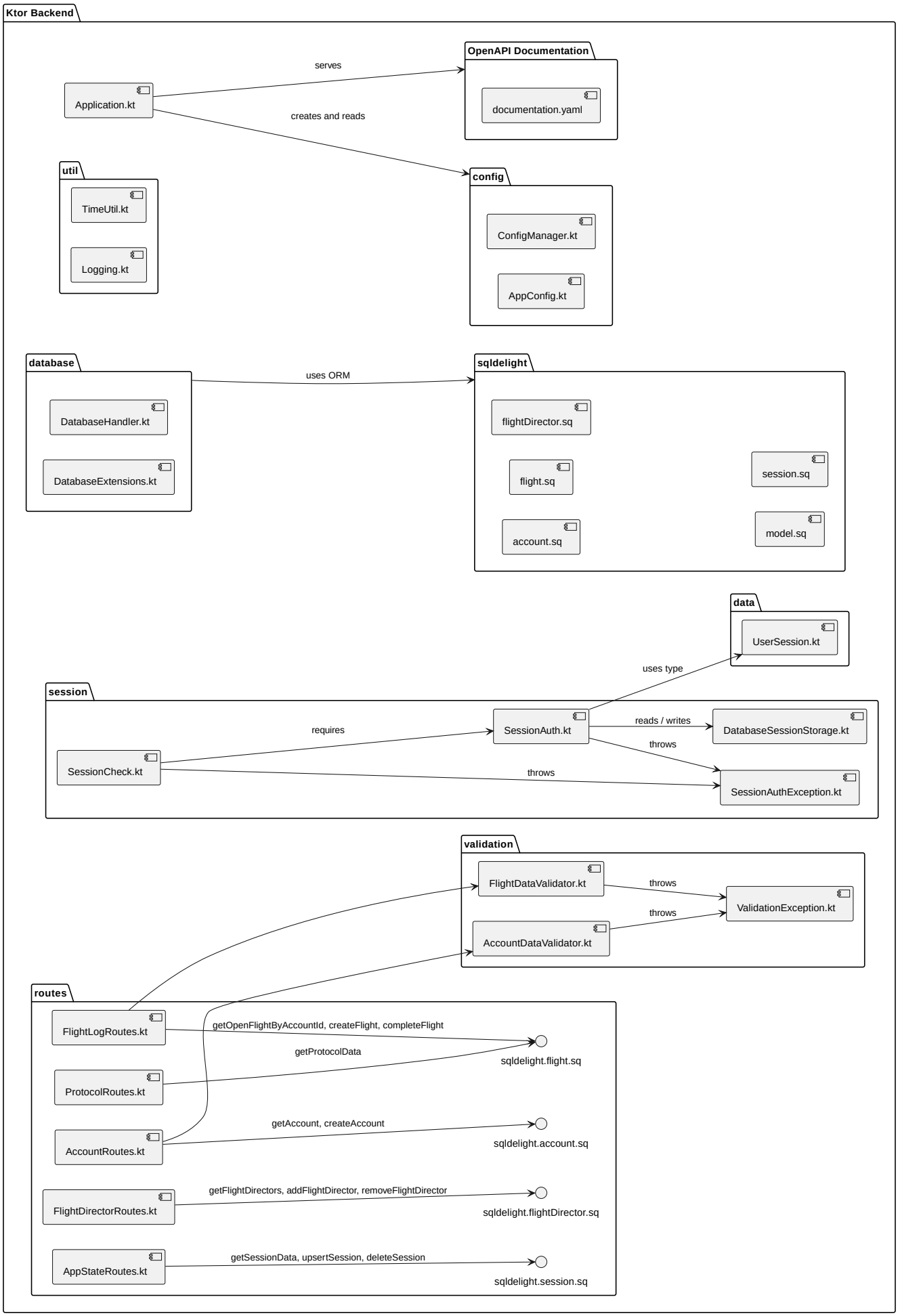


Abbildung 10. Paketdiagramm für das Backend

2. Softwaredokumentation

Die Softwaredokumentation ist in HTML Form mit GitHub Pages zur Verfügung gestellt:

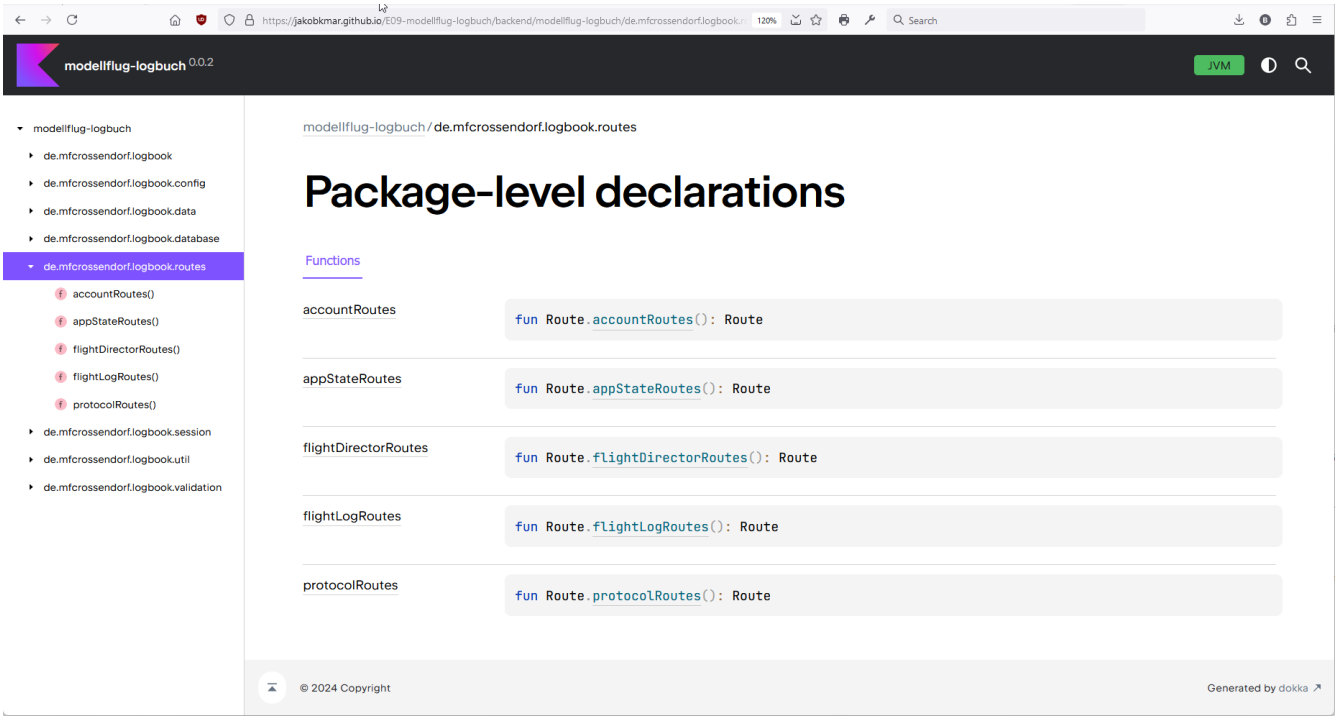
https://jakobkmar.github.io/E09-modellflug-logbuch/

Dabei werden insbesondere die folgenden zwei Module angeboten:

- **Common Data**
  - hier befinden sich alle geteilten Datenstrukturen (data classes etc.) welche sowohl durch das Frontend als auch das Backend verwendet werden
  - es handelt sich hautpsächlich um immutable und serialisierbare Klassen, welche auch zu `.d.ts` Typdefinitionen kompiliert werden können
- **Backend**
  - hier befindet sich der gesamte Backend-Code, welcher in Kotlin geschrieben ist
  - insbesondere die Routen sind hier deklarativ gegeben

Die Dokumentation wurde mit Dokka generiert, welches die offizielle API documentation engine für Kotlin ist.

Dies sieht dann wie folgt aus:



3. Am Projekt entwickeln

3.1. Entwicklungsumgebung

Nachfolgend sind die Tools aufgelistet, die für die Entwicklung der Anwendung benötigt werden. Teilweise sind Anweisungen für die Installation unter Windows enthalten. Auf Linux können die Pakete über den Paketmanager installiert werden.

3.1.1. Backend

- IntelliJ IDEA (Ultimate Edition wenn auch Frontend)
- Java 17+ `winget install EclipseAdoptium.Temurin.21.JDK`
- Gradle

3.1.2. Frontend

- IntelliJ IDEA (Ultimate Edition)
- **und/oder** Visual Studio Code
- Node.js `winget install OpenJS.NodeJS`
- pnpm `winget install pnpm.pnpm`

3.1.3. Datenbank

- PostgreSQL mit Docker https://docs.docker.com/get-docker/

Nachfolgend die `docker-compose.yml` Datei für die Datenbank:

```yaml
services:
  db:
    image: postgres
    # set shared memory limit when using docker-compose
    shm_size: 128mb
    environment:
      POSTGRES_USER: logbook
      POSTGRES_PASSWORD: logbook
      POSTGRES_DB: logbook
    ports:
      - 5432:5432
    volumes:
      - ./postgres-data:/var/lib/postgresql/data

  adminer:
    image: adminer
    ports:
      - 8081:8080
```

Diese an eine geeignete Stelle auf dem eigenen Rechner speichern und mit `docker-compose up -d` starten.

Nun ist die Datenbank unter `localhost:5432` und das Admin-Interface unter `localhost:8081` erreichbar.

3.2. Starten der Anwendung in der Entwicklungsumgebung

Im Terminal in den Projektordner wechseln. Dann in den `backend` Ordner wechseln und die Anwendung starten.

```
cd src/backend
```

```
# auf Linux
./gradlew runFullstackApp
# auf Windows
./gradlew.bat runFullstackApp
```

3.2.1. Konfiguration nach dem ersten Start

Mit dem ersten Start wird eine `config.toml` im `config` Ordner erstellt. Diese Datei enthält die Konfiguration für die Anwendung. Hier können die Einstellungen für die Datenbankverbindung und den Serverport angepasst werden, sowie der **Entwicklungmodus** aktiviert werden.

Ein Beispiel für die Konfiguration wäre:

```
# The time zone to use for the application.
# If not set, the system default is used.
# See https://kotlinlang.org/api/kotlinx-datetime/kotlinx-datetime/kotlinx.datetime/-
time-zone/-companion/of.html for more information.
# timeZone = null

[database]
  serverName = "localhost"
  # The PostgreSQL default port is 5432.
  port = 5432
  username = "logbook"
  password = "logbook"
  # The name of the database to use. (You chose this during database setup.)
  databaseName = "logbook"

[server]
  # The port the server should listen on.
  port = 8080
  # In development mode the server runs without many security features.
```

```
  # This is useful for testing and debugging in local environments.
  developmentMode = true
```