Institut für
Strukturmechanik
und Leichtbau

RWTH AACHEN
UNIVERSITY

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Effiziente Entwicklung von Verwaltungssoftware durch Low-Code-Ansätze und moderne Webtechnologien

Diese Seminararbeit wurde vorgelegt am

Fachbereich 9
Medizintechnik und Technomathematik
Fachhochschule Aachen, Campus Jülich

von

Markus Beckschulte
Matrikelnummer: 3593273

und wurde betreut von

1. Prüfer:
Prof. Dr. rer. nat. Alexander Voß
Fachbereich 9
FH Aachen University of Applied Science

2. Prüfer:
Prof. Dr.-Ing. Kai-Uwe Schröder
Institut für Strukturmechanik und Leichtbau
RWTH Aachen University

Aachen, Dezember 2024

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Effiziente Entwicklung von Verwaltungssoftware durch Low-Code-Ansätze und moderne Webtechnologien

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Aachen, den 20. Dezember 2024

M. Beckschulte

Beckschulte, Markus

Kurzfassung

Die Entwicklung moderner Softwareanwendungen erfordert eine strukturierte Herangehensweise, die komplexe Systeme wie Client-Server-Architekturen einbezieht. Diese Architekturen bilden die Grundlage vieler Anwendungen, bei denen ein Client und ein Server über standardisierte Schnittstellen wie REST-APIs kommunizieren. Besonders hervorzuheben sind Single-Page-Applications (SPA), die durch ihre interaktiven Benutzeroberflächen und nahtlose Nutzererfahrungen zunehmend an Bedeutung gewinnen.

Die Arbeit widmet sich den theoretischen Grundlagen und der praktischen Umsetzung einer solchen Architektur, wobei React als Framework für die Client-Seite und Python-basierte Technologien wie FastAPI und Tortoise-ORM auf der Server-Seite eingesetzt werden. Darüber hinaus wird ein durchgängiges Modell verwendet, um Entitäten und deren Relationen darzustellen. Themen wie Object-Relational Mapping (ORM), die Implementierung von C.R.U.D.-Operationen, JSON-Web-Tokens (JWT) für Authentifizierungsprozesse und die Integration dieser Technologien in ein kohärentes System werden ausführlich erläutert.

Ziel ist es, ein tiefgreifendes Verständnis für die Synergien zwischen diesen Technologien zu schaffen und die Grundlage für eine effiziente Entwicklung von Verwaltungssoftware mit Low-Code-Ansätzen zu legen. Die Arbeit behandelt sowohl die theoretischen als auch praktischen Aspekte dieser Architektur und bietet einen umfassenden Überblick über die eingesetzten Konzepte und Methoden.

Schlagwörter: Low-Code, Object-Relational Mapping, Python, React, REST-API

Inhaltsverzeichnis

Abkürzungsverzeichnis

ORM Objektrelationale Abbildung (Object-Relational Mapping)

SPA Single-Page-Application

REST-API Representational State Transfer API

API Application Programming Interface

JSON JavaScript Object Notation

JWT JSON Web Token

ER Entity-Relationship

URI Uniform Resource Identifier

C.R.U.D. Create, Read, Update, Delete

JTI JWT Identifier

DTO Data Transfer Object

CSS Cascading Style Sheet

PEP8 Python Enhancement Proposal

BPMN Business Process Modelling Notation

1 Einleitung

Moderne Softwarelösungen und ihre Entwicklung stehen zunehmend vor der Herausforderung, flexibel, effizient und benutzerfreundlich zu sein. Zudem steigen die Anforderungen bezüglich Sicherheit, Skalierbarkeit und Automatisierung. Klassische Entwicklungsansätze stoßen hierbei oft an ihre Grenzen, da sie mit einem hohen Zeit- und Ressourcenaufwand verbunden sind. Im Bereich der Verwaltungssoftware zeigt sich dies: Die Modellierung komplexer Datenstrukturen in allen Ebenen der Software führt häufig zu redundanten und fehleranfälligen Entwicklungsprozessen.

Ein Lösungsansatz, der dies adressiert, ist der Einsatz moderner Technologien und offener Standards in Kombination mit einem Low-Code-Ansatz. Durch die einmalige Definition der Datenstruktur und der automatischen Generierung der daraus resultierenden Komponenten im Front- und Backend kann den Entwicklungsaufwand sowie die Fehleranfälligkeit erheblich reduzieren.

1.1 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, eine Verwaltungssoftware zu entwickeln, die durch den Einsatz moderner Technologien und offener Standards sowohl eine effiziente Entwicklung als auch eine hohe Benutzerfreundlichkeit gewährleistet. Dabei wird untersucht, wie ein Low-Code-Ansatz konkret zur Reduktion von Redundanz und zur Beschleunigung der Entwicklungsprozesse beitragen kann. Außerdem soll eine skalierbare und sichere Architektur geschaffen werden, die sich leicht erweitern lässt.

1.2 Vorgehen

Um dieses Ziel zu erreichen, wird zunächst auf die theoretischen Grundlagen eingegangen, insbesondere auf die Prinzipien von Client-Server-Architekturen, REST-APIs und Datenmodellierung. Aufbauend darauf werden die eingesetzten Technologien (React, FastAPI, Tortoise-ORM) und deren Integration erläutert. Anschließend wird die konkrete Umsetzung der Verwaltungssoftware beschrieben, einschließlich der Abbildung von Datenstrukturen und Beziehungen, der Authentifizierung über JSON-Web-Tokens (JWT) und der Gestaltung der Benutzeroberfläche.

Anschließend erfolgt eine Evaluierung der erzielten Ergebnisse sowie eine Diskussion der Herausforderungen und Potenziale für zukünftige Erweiterungen. Abschließend wird ein Fazit gezogen.

2 Theoretische Grundlagen

Im Folgenden wird auf die theoretischen Grundlagen der Client-Server-Architektur eingegangen, wobei auf der Seite des Clients im Speziellen eine React-Webseite und auf der Seite des Servers eine REST-API verwendet wird. Client-Server-Architekturen sind ein zentrales Paradigma moderner Softwareentwicklung und in zahlreichen Werken zur Softwarearchitektur beschrieben [1]. Bei einer React-Webseite handelt es sich um eine Single-Page-Application (SPA), die mittels des JavaScript-Frameworks React entwickelt wird [2, 3]. Eine REST-API stellt eine einheitliche Schnittstelle bereit, um Objekte, die auf einem Server liegen, mittels HTTP-Operationen zu verwalten.

In den Folgenden Abschnitten wird darauf eingegangen, welche Relationen zwischen verschiedenen Entitäten bestehen können und wie Entitäten und ihre Relationen mittels einer REST-API verwaltet werden können. Anschließend werden JSON-Web-Tokens und ihre Verwendung in einem Login- bzw. Logout-Prozess behandelt. Zum Ende des Kapitel wird React thematisiert.

Zur Erläuterung soll ein beispielhaftes Modell dienen, das alle notwendigen Aspekte abdeckt: Es gibt die Entitätsklassen Gruppen, Benutzer, Teams, Artikel, Kommentare sowie Arbeitsplätze. Während ein Benutzer mehreren Gruppen angehören kann, kann er nur Mitglied eines Teams sein. Weiter kann ein Benutzer Artikel verfassen und andere Artikel kommentieren. Der Benutzer ist einem eigenen Arbeitsplatz zugeordnet. Ein Werkzeug zur Modellierung von Daten ist das Entity-Relationship-Modell (ER-Modell), das seit der Einführung durch Chen ein grundlegendes Mittel zur Datenmodellierung ist [4]. Modelliert man das Beispiel als ER-Diagramm, so erhält man das in Abbildung 2.1 gezeigte Diagramm.

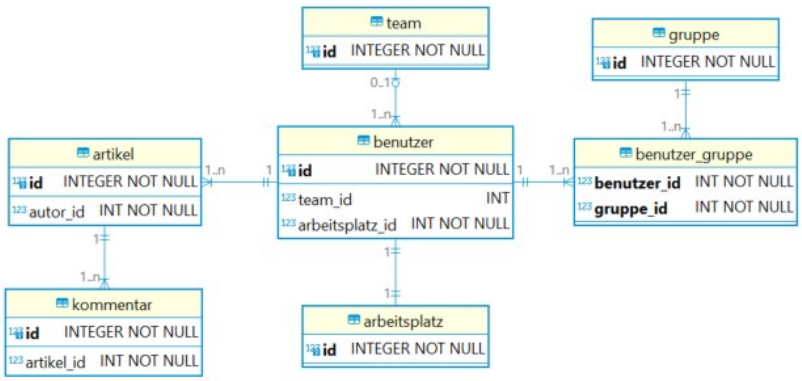


Abb. 2.1: ER-Diagramm des Beispielmodells

2.1 Relationen zwischen Entitätsklassen

Anhand des Beispiels in Abbildung 2.1 ergeben sich verschiedene Arten von Beziehungen zwischen Entitätsklassen, welche in der Datenbankmodellierung und -theorie seit langem etabliert sind [4]:

Many-to-Many (viele zu vielen): Ein Benutzer kann vielen Gruppen angehören und eine Gruppe hat viele Benutzer.

One-to-Many (einer zu vielen): Zu einem Artikel gehören viele Kommentare, während ein Kommentar eine Zuordnung zu einem Artikel benötigt. Zu einem Benutzer gehören viele Artikel, wobei ein Artikel auch immer einem Benutzer zugeordnet sein muss. Zudem gehören zu einem Team mehrere Benutzer. Ein Benutzer muss jedoch nicht einem Team angehören (nullable).

One-to-One (einer zu einem): Ein Benutzer hat genau einen Arbeitsplatz. Ein Arbeitsplatz kann auch einem Benutzer zugeordnet sein, jedoch besteht der Arbeitsplatz auch ohne Benutzer weiterhin.

Diese Beziehungstypen müssen sowohl von einer REST-API als auch von der verwendeten Software zur Objektrelationalen Abbildung (Object-Relational Mapping, ORM) abgedeckt werden. ORM-Frameworks erleichtern die Abbildung von Objektmodellen auf relationale Datenbanken und wurden in der Literatur umfassend beschrieben [5]. In dieser Arbeit kommt Tortoise-ORM zum Einsatz, das ein Python-basiertes ORM-Framework ist [6].

2.2 Object Relational Mapping (ORM) mit Tortoise-ORM

Um die Persistenzschicht zu vereinfachen, wird ein ORM-Framework eingesetzt. Allgemeine Patterns zur Objektrelationalen Abbildung wurden von Fowler beschrieben [5]. Tortoise-ORM als konkretes Framework bietet Python-Entwicklern eine effiziente Möglichkeit, mit relationalen Datenbanken umzugehen [6]. Hierbei unterstützt es eine Vielzahl von Datenbankmanagementsystemen: PostgreSQL, SQLite, MySQL, MariaDB, Microsoft SQL Server sowie Oracle SQL Server.

Tortoise-ORM ist ein vergleichsweise junges Projekt von Andrey Bondar, dessen Anfänge im Jahr 2018 liegen. Es ist ein asynchron arbeitendes, von Django inspiertes ORM. Dadurch bietet Tortoise-ORM eine leicht zugängliche API und die Vorzüge der Asynchronität. Weiter ist es mit dem Gedanken entwickelt worden, effizient Web-Services entwickeln zu können mit den in Abschnitt 2.1 beschriebenen Relationen.[6]

Im Zuge des Augenmerks auf die Webentwicklung wurde 2020 die Serialisierung mit Pydantic direkt in das Projekt intregiert [7]. Pydantic ist eine Python-Bibliothek, die Datenvalidierung und -serialisierung auf Basis von Python-Typannotationen unterstützt. Hierdurch können Entitäten aus der Datenbank direkt als JSON-Objekt serialisiert sowie bestehende JSON-Objekte validiert werden, ohne explizit mit Pydantic Datenklassen zu definieren. Desweiteren bietet es eine direkte Integration in das Framework FastAPI, auf das im nächsten Abschnitt eingegangen wird.

2.3 Representational-State-Transfer-API (REST-API)

Eine REST-API dient dazu, Objekte, die auf einem Server liegen, zu lesen und zu verwalten. REST steht für Representational State Transfer, ein Architekturstil, der erstmals im Jahr 2000 durch Roy Fielding eingeführt wurde [8]. Die Prinzipien von REST wurden in den letzten Jahren in zahlreichen Werken weiter konkretisiert [9]. Es ist ein Architekturstil, der auf vier zentralen Vorgaben basiert:

1. Identifizierung von Resourcen
2. Manipulation von Ressourcen durch Repräsentationen
3. Selbstbeschreibende Nachrichten
4. Hypermedia als Engine des Anwendungszustands

Zur eindeutigen Referenzierung von Resssourcen dienen hierbei Uniform Resource Identifier (URI), deren generische Syntax in RFC 3986 beschrieben ist [10].

Angewandt auf das HTTP-Protokoll ergeben sich hieraus verschiedene Paradigmen, auf die im Folgenden eingegangen wird.

Persistente Speicher werden durch die vier grundlegenden Operationen „Create“, „Read“, „Update“ und „Delete“ realisiert, die zusammen das Akronym „C.R.U.D“ bilden [11, 12, 13]. Diese orientieren sich an grundlegenden Datenbank-Operationen im relationalen Modell [12].

Um diese einzelnen Operationen anbieten zu können, werden sie wie folgt auf HTTP-Operationen abgebildet:

- **Create:** POST
- **Read:** GET
- **Update:** PUT
- **Delete:** DELETE

Bei der Erstellung eines Objekts und bei der Auflistung aller Objekte werden jeweils der gleiche Pfad verwendet, beispielsweise `/api/entity`. Bei den Operationen Update, Delete und Read eines bestimmten Objekts wird eine ähnliche Route verwendet, `/api/entity/{id}`, wobei `{id}` für einen eindeutigen Identifizierer steht. Hierbei steht `entity` jeweils für eine bestimmte Entitätsklasse, beispielsweise bei «Benutzer» `/api/benutzer` bzw. `/api/benutzer/24`, wenn es sich um den Benutzer mit dem Identifizierer 24 handelt.

2.3.1 Ablauf der einzelnen Operationen

Im folgenden werden die elementaren C.R.U.D-Operationen und anschließend die relationalen Operationen erläutert wie sie sich nach aktuellem Stand der Technik etabliert haben.

C.R.U.D-Operationen

Create:

1. Es wird eine POST-Anfrage an den Server geschickt mit dem Pfad `/api/entity`. Hierbei ist im Body der Anfrage ein JSON-Objekt enthalten, das die zu erstellende Entität beschreibt.
2. Es wird überprüft, ob der anfragende Benutzer die Entität erstellen darf.
3. Die Eingaben werden auf Gültigkeit überprüft.
4. Es wird versucht, die Entität zu erstellen. Falls sie schon existiert, wird eine entsprechende Fehlermeldung an den Benutzer geschickt.
5. Die erstellte Entität wird erneut als JSON-Objekt serialisiert und an den Benutzer als Antwort geschickt.

Read (alle Entitäten):

1. Es wird eine GET-Anfrage an den Pfad `/api/entity` gesendet.
2. Es wird überprüft, ob der anfragende Benutzer die Berechtigung zum Lesen der Entitäten besitzt.
3. Die Entitäten werden von der Datenbank abgefragt.
4. Die abgefragten Entitäten werden anschließend als JSON-Liste serialisiert.
5. Die serialisierten Objekte werden an den Benutzer als Antwort gesendet.

Read (einzelne Entität):

1. Der Benutzer sendet eine GET-Anfrage and den Pfad `/api/entity/{id}`, um die Entität mit der entsprechenden ID abzufragen.
2. Es wird überprüft, ob der Benutzer die Berechtigung hierzu besitzt.

3. Die Entität wird von der Dantenbank abgefragt.
4. Falls sie nicht vorhanden ist, wird eine entsprechende Fehlermeldung zurück gesendet.
5. Die abgefragte Entität wird als JSON-Objekt serialisiert und als Antwort gesendet.

Update:

1. Der Benutzer sendet eine PUT-Anfrage and den Pfad `/api/entity/{id}`, um die Entität mit der entsprechenden ID zu aktualisieren. Hierbei wird im Body ein JSON-Objekt mitgesendet, das die zu aktualisierende Entität beschreibt.
2. Es wird überprüft, ob der Benutzer hierzu die Berechtigung aufweist.
3. Die angefragte Entität wird von der Datenbank abgerufen. Falls sie nicht existiert, wird eine entsprechende Fehlermeldung zurückgegeben.
4. Anschließend werden die Benutzereingaben auf Gültigkeit überprüft.
5. Die Entität wird in der Datenbank aktualisiert.
6. Die aktualisierte Entität wird als JSON-Objekt serialisiert und an den Benutzer als Antwort geschickt.

Delete:

1. Der Benutzer sendet eine DELETE-Anfrage and den Pfad `/api/entity/{id}`, um die Entität mit der entsprechenden ID zu löschen.
2. Es wird überprüft, ob der Benutzer hierzu die Berechtigung aufweist.
3. Die zu löschende Entität wird von der Datenbank abgerufen. Falls sie nicht vorhanden ist, wird eine entsprechende Fehlermeldung zurück gesendet.
4. Die Entität wird von der Datenbank gelöscht.
5. Eine Antwort mit dem Status `204 - No Content` wird an den Benutzer gesendet, um die erfolgreiche Löschung zu signalisieren.

Relationale Operationen: Neben der Elementareigenschaften einer Entität kann eine Entität auch andere Entitäten als Eigenschaft besitzen. Diese Beziehungstypen, die auch in Abschnitt 2.1 aufgeführt sind, müssen von einer REST-API abgedeckt werden. Im Folgenden wird darauf eingegangen, welche Operationen hierfür notwendig sind und wie sie umgesetzt werden.

Many-to-Many-Beziehung: Bei einer Many-to-Many-Beziehung sind die erforderlichen Operationen (1) die Subentität abzurufen sowie (2) eine Subentität der Relation hinzuzufügen und (3) entfernen zu können. In dem Beispiel aus Abbildung 2.1 Gruppe-Benutzer muss es also möglich sein, (1) die Benutzer einer Gruppe abzurufen, (2) einen Benutzer einer Gruppe hinzuzufügen oder (3) einen Benutzer von einer Gruppe entfernen zu können.

(1) Abrufen einer Subentität hat folgendes Muster:
`GET /api/entity/{id}/subentities`
Möchte man die Benutzer der Gruppe mit der ID 42 abrufen, so ruft man `GET /groups/42/users` auf.

(2) Hinzufügen einer Subentität hat folgendes Muster:
`PUT /api/entity/{id}/subentities/{subentity-id}`
Möchte man den Benutzer mit der ID 24 der Gruppe mit der ID 42 hinzufügen, ruft man `PUT /groups/42/users/24` auf.

(3) Entfernen einer Subentität hat folgendes Muster:
`DELETE /api/entity/{id}/subentities/{subentity-id}`
Möchte man den Benutzer mit der ID 24 der Gruppe mit der ID 42 entfernen, ruft man `DELETE /groups/42/users/24` auf.

One-to-Many-Beziehung: Hierbei gibt es zwei Szenarien, die getrennt voneinander betrachtet werden müssen:

1. Beide Seiten der Relation können ohne die jeweils andere existieren.

2. Eine Seite der Relation kann nicht ohne die andere Seite existieren.

Können beide Seiten der Relation unabhänig voneinander existieren, wie bei der Relation Team-Benutzer aus dem Beispiel in Abbildung 2.1, sind folgende Operationen notwendig:

- Abrufen einer Subentität von der One-Seite hat folgendes Muster:
`GET /api/entity/{id}/subentities`
Möchte man die Benutzer des Teams mit der ID 42 abrufen, so ruft man `GET /api/teams/42/users` auf.

- Abrufen einer Subentität von der Many-Seite hat folgendes Muster:
`GET /api/entity/{id}/subentity`
Möchte man das Team des Benutzers mit der ID 42 abrufen, so ruft man `GET /api/users/42/team` auf.

- Hinzufügen einer Subentität hat folgendes Muster:
  `PUT /api/entity/{id}/subentities/{subentity-id}`
  Möchte man den Benutzer mit der ID 24 dem Team mit der ID 42 hinzufügen, ruft man `PUT /api/teams/42/users/24` oder `PUT /api/users/24/team/42` auf.

- Entfernen einer Subentität hat folgendes Muster:
  `DELETE /api/entity/{id}/subentities/{subentity-id}`
  Möchte man den Benutzer mit der ID 24 dem Team mit der ID 42 entfernen, ruft man `DELETE /api/teams/42/users/24` oder `DELETE /api/users/24/team` auf.

Ist die Many-Seite von der Zuordung abhängig, wie bei der Relation Kommentar-Artikel aus dem Beispiel in Abbildung 2.1, sind folgende Operationen notwendig:

- Abrufen einer Subentität von der One-Seite hat folgendes Muster:
  `GET /api/entity/{id}/subentities`
  Möchte man die Kommentare des Artikels mit der ID 42 abrufen, so ruft man `GET /api/artikel/42/kommentare` auf.

- Abrufen einer Subentität von der Many-Seite hat folgendes Muster:
  `GET /api/entity/{id}/subentity`
  Möchte man den Artikel des Kommentars mit der ID 42 abrufen, so ruft man `GET /api/kommentare/42/artikel` auf.

- Erstellen einer Subentität hat folgendes Muster:
  `POST /api/entity/{id}/subentities`
  Möchte man einen Kommentar dem Artikel mit der ID 42 hinzufügen, ruft man `POST /api/artikel/42/kommentare` auf. Hierbei enthält der Body die weiteren für den Kommentar benötigten Attribute.

- Löschen einer Subentität hat folgendes Muster:
  `DELETE /api/entity/{id}/subentities/{subentity-id}`
  Möchte man den Kommentar mit der ID 24 des Artikels mit der ID 42 entfernen, ruft man `DELETE /api/artikel/42/kommentare/24` auf. Alternativ kann man auch
  `DELETE /api/kommentare/24` aufrufen.

One-to-One-Beziehung: Zur Herstellung einer One-to-One-Beziehung muss die eine Seite stehts vor der anderen existieren können, es können nicht beide gleichzeitig angelegt werden. Daher muss eine Seite immer unabhängig von der anderen existieren können. Daher ist hier die Fragestellung, ob die andere Seite auch unabhängig existieren kann. Dies ist das gleiche Problem wie auch schon bei One-to-Many-Beziehung, jedoch erhält man beim Abrufen der Subentität keine Liste als Antwort sondern ein einzelnes Objekt. Bezogen auf die Relation Benutzer-Arbeitsplatz aus dem Beispiel in Abbildung 2.1 kann eine Beziehung erst erstellt werden, wenn der Arbeitsplatz existiert.

2.4 Webframework mit FastAPI

FastAPI ist ein Python-basiertes Webframework, mit dem sich unter anderem REST-APIs effizient erstellen lassen. Es nutzt Python-Type-Hints umfassend, um dynamische Routen zu generieren sowie die zulässigen Cookie-, Header- und Body-Daten eindeutig zu deklarieren. Diese werden durch Pydantic definiert, verifiziert und (de-)serialisiert. Desweiteren hat es eine direkte Integration von OpenAPI, früher Swagger. OpenAPI ist eine offene, standardisierte Spezifikation, mit der sich REST-APIs eindeutig und maschinenlesbar beschreiben lassen, wodurch die Dokumentation der API deutlich erleichtert wird [14]. Es stellt zudem eine direkte Weboberfläche via Swagger UI oder ReDoc bereit, in der die generierte Dokumentation grafisch aufgewertet präsentiert wird.

Wie Tortoise-ORM arbeitet es auch asynchron, da es auf ASGI (Asynchronous Server Gateway Interface) basiert. Durch die Verwendung von Pydantic sowohl bei Tortoise-ORM als auch bei FastAPI ist eine Integration beider Bibliotheken direkt gegeben: Die Datenmodelle, die in Tortoise-ORM definiert wurden, werden anschließend in ein Pydantic-Modell überführt. Diese werden dann als zu erwartender Input oder zu sendender Output für eine bestimmte Route von FastAPI verwendet und auch direkt zur Validierung und (De-)Serialisierung sowohl auf der Client- als auch auf der Serverseite verwendet. Clientseitig kann das beispielsweise geschehen, indem das passende JSON-Schema verwendet wird.

2.5 (De-)Serialisierung und Validierung mit Pydantic

Pydantic ist eine Python-Bibliothek zur Validierung von Daten. Hierbei werden Pydantic-Modelle zur Beschreibung einer Datenstruktur verwendet. Sie dienen dazu, eingehende und ausgehende Datenvalidierungen vorzunehmen sowie eine klare, schemabasierte Schnittstelle mittels dem Standard JSON-Schema für externe Systeme oder Frontends bereitzustellen. Auf diese Weise lassen sich Datenübertragungen zwischen Anwendungskomponenten und externen Schnittstellen standardisiert, typisiert und validiert gestalten, ohne redundanten Code für die Modellierung und Validierung von Datenstrukturen schreiben zu müssen.

2.6 JSON Web Tokens (JWT)

Neben der Darstellung von Entitäten durch eine REST-API muss diese durch gewisse Authentifizierungsmechanismen abgesichert werden. Hierfür werden klassischerweise JSON-Web-Tokens (JWTs) verwendet, die in RFC 7519 spezifiziert sind [15]. Sie sind insbesondere für zustandslose Architekturen (wie REST) geeignet, da keine serverseitige Sessionhaltung notwendig ist.

Neben der Verwendung bei Protokollen wie OpenID Connect oder OAuth 2.0 [16, 17] zum Informationsaustausch mit einem Identity Provider kann ein JWT auch zur eindeutigen Verifizierung eines Clients verwendet werden.

Im folgenden wird auf den Aufbau von Refresh- und Access-Token und wie man sie validiert eingegangen. Im Anschluss wird deren Verwendung bei Login-, Logout- und Refresh-Vorgängen thematisiert.

2.6.1 Aufbau eines Refresh- und Access-Tokens

Ein JWT besteht aus Header, Payload und Signatur [15]. Im *Header* wird festgehalten, welcher Algorithmus verwendet wurde und dass es sich um einen JWT handelt. Im Payload stehen Informationen wie Ablaufzeitpunkt, Nutzer-ID und Token-Typ. Diese sind beim Access- und Refresh-Token folgende:

exp: Ablaufzeitpunkt des Tokens

sub: ID des Nutzers

type: Art des Tokens, entweder Refresh oder Access

jti: zufallsgenerierte ID des Tokens

Anschließend werden Header und Payload signiert. Hierzu werden sie mit base64 codiert und mit einem Punkt zusammengefügt. Anschließend wird diese Zeichenkette mit dem im Header festgehaltenen Algorithmus und einem nur dem Server bekannten Secret gehasht. Ein Secret ist hierbei eine zufällige Zeichenkette. Dieser Hash wird anschließend an die zuvor gebildete Zeichenkette mit einem Punkt dazwischen angehängt. Es entsteht folgender schematischer Aufbau:

```
base64UrlEncode(header) + "." + base64UrlEncode(payload) + "."
    ↪ + base64UrlEncode(hash)
```

2.6.2 Validierung eines Tokens

Bei der Validierung wird der Token mittels des Secrets geprüft. Zudem wird sichergestellt, dass das Ablaufdatum noch nicht überschritten ist und der Token nicht auf einer Blacklist steht. Auch diese Vorgehen sind in den entsprechenden Spezifikationen und Best Practices festgehalten [15].

Zunächst wird der Token in Header, Payload und Signatur aufgeteilt. Anschließend werden Header und Payload gegen die Signatur geprüft wobei das serverseitig gespeicherten Secret verwendet wird. Hierdurch wird sichergestellt, dass der Token vom Server erstellt wurde, da nur er das Secret kennt. Dann wird geprüft, ob der im Payload mitgelieferte Ablaufzeitpunkt in der Vergangenheit liegt. Falls ja, ist er abgelaufen und nicht mehr valide. Nun kann überprüft werden, ob die im Payload mitgelieferte ID («jti»), die den Token eindeutig identifiziert, in der Tokenblacklist vorhanden ist. Falls dies der Fall ist, ist der Token nicht valide. Hat der Token die vorherigen Überprüfungen überstanden, ist der Token valide und die im Payload mitgelieferte ID des Users («sub») entspricht der des angemeldeten Benutzers.

2.6.3 Tokenblacklist

Die Tokenblacklist wird mittels Tortoise-ORM in der Datenbank persistiert. Hierbei werden sowohl die ID des Token («jti») als auch sein Ablaufdatum («exp») festgehalten. Da ein Token auch anhand seines Ablaufdatums als nicht valide betrachtet werden kann und diese Prüfung vor der Tokenblacklist stattfindet, kann man Tokens, deren Ablaufdatum in der Vergangenheit liegen, aus der Tokenblacklist regelmäßig entfernen. Diese regelmäßige Löschung muss als regelmäßiger Hintergrundprozess implementiert werden.

2.6.4 Login-, Logout- und Refresh-Vorgang

Im Folgenden wird auf den Login-, Logout- und Refresh-Vorgang unter Verwendung von JSON-Web-Tokens eingegangen.

Ablauf des Login-Vorgangs

1. Client sendet Nutzername und Passwort JSON-codiert im Body als POST-Anfrage an den Login-Endpoint, beispielsweise `/api/login`.
2. Der Benutzer wird in der Datenbank abgefragt
3. Das mitgelieferte Passwort wird gegen den abgespeicherten Hash validiert
4. Es wird eine Antwort vorbereitet, in dem ein Access-Token und ein Refresh-Token, jeweils als Cookie, mitgeliefert werden.

5. Die Antwort wird an den Client gesendet.

Abauf des Logout-Vorgangs

1. Es wird überprüft, ob man eingeloggt ist, indem der mitgelieferte Access-Token validiert wird.
2. Es wird eine Antwort vorbereitet, mit dem der Access-Token und der Refresh-Token gelöscht werden.
3. Die Antwort als HTTP Response wird an den Client gesendet.

Ablauf des Refresh-Vorgangs

Ist der Access-Token abgelaufen, so kann der Client ihn mittels des Refresh-Tokens erneuern.

1. Es wird überprüft, ob der mitgelieferte Refresh-Token valide ist.
2. Es wird eine Antwort vorbereitet, in dem ein neuer Access-Token und ein neuer Refresh-Token, jeweils als Cookie, mitgeliefert werden.
3. Die alten Token werden einer Tokenblacklist hinzugefügt.
4. Die Antwort wird als HTTP Response an den Client gesendet.

2.7 Überblick über React

React wurde von Facebook (jetzt Meta) initiiert und ist heute ein etabliertes Framework für moderne Webanwendungen auf Basis von JavaScript, das die Erstellung von SPAs erleichtert [2, 3]. Eine SPA zeichnet sich dadurch aus, dass das Routing innerhalb des Browsers stattfindet, was als clientseitiges Routing bezeichnet wird [18]. Während bei klassischen Webseiten bei einer Änderung des Pfades stets die Webseite neu geladen und vom Server neu ausgeliefert werden muss, wird bei einer SPA der Pfad nur manipuliert und die Webseite kann darauf reagieren. Dies geschieht mit der History-API von HTML5 [19]. So wird erreicht, dass nur der sich ändernde Inhalt vom Server geladen werden muss, was eine deutlich nahtlosere Benutzererfahrung ermöglicht und die Serverlast veringert [20]. Als Client in einer Client-Server-Architektur stellt React eine ideale Grundlage dar, um mittels einer REST-API Daten zu konsumieren, darzustellen und zu manipulieren.

3 Entwicklung der Verwaltungssoftware

Die Verwaltungssoftware soll als Client-Server-Architektur verwirklicht werden, wobei der Client als Webseite zur Verfügung gestellt werden soll. Im Folgenden wird behandelt, wie sich diese Architektur grundsätzlich zusammensetzt, wie die einzelnen Komponenten aufgebaut sind und wie sie miteinander zusammenarbeiten.

3.1 Architektur der Gesamtlösung

Der Grundgedanke ist, dass aus der Struktur der Daten automatisch eine Server-API generiert wird, aus der anschließend durch den Client ein Frontend generiert wird. Somit handelt es sich um einen Low-Code-Ansatz, was bedeutet, dass zur Entwicklung eines vollständigen Produkts nur verhältnismäßig wenig Code geschrieben werden muss [21]. Hierbei wird die Datenstruktur im Backend definiert, die so aufgearbeitet werden muss, dass der Client anschließend ein entsprechendes Frontend aufbauen kann, das auch mit der dahinterliegenden Funktionalität zusammen arbeitet.

Zum Austausch von Daten in der Welt der Webseitenentwicklung wird hauptsächlich das JSON-Format verwendet, da es im Vergleich zu zum Beispiel XML im Bezug auf die versendeten Rohdaten äußerst datensparsam ist [22]. Da sowohl Client als auch Server wissen müssen, wie die zu behandelnden Daten aussehen müssen (also eine Repräsentation der Datenstruktur) bietet es sich an, dies mittels des JSON-Schema-Formats, wie es in RFC 8927 beschrieben ist [23], auszutauschen. Weiter muss dem Client auch mitgeteilt werden, welche Operationen auf dem Server ausgeführt werden können. Im Folgenden wird darauf eingegangen, wie dieser Austausch von statten geht und wie die einzelnen Komponenten, Client und Server, aufgebaut sind.

3.2 Konzept des REST-Backends

Das Backend ist vollständig in Python geschrieben. Hierbei wurde bei der Auswahl der verwendeten Pakete darauf geachtet, dass sie eine Schnittstelle zur asyncio-API besitzen, um eine asynchrone Verarbeitung zu garantieren.

Zur Darstellung und Verarbeitung der Daten wird Tortoise-ORM und als Webframework wird FastAPI verwendet. Während durch Tortoise-ORM die Daten abgerufen und manipuliert werden, stellt FastAPI die Serverlogik bereit, um mittels eines ORMs verschiedene Daten über HTTP zur Verfügung stellen zu können. Im Folgenden wird darauf eingegangen, wie sie genau funktionieren und wie sie miteinander integriert werden.

3.2.1 Tortoise-ORM

Tortoise-ORM ermöglicht die Definition von Datenbankmodellen als Python-Klassen, wobei Relationen direkt über Klassenattribute und spezifische Feldtypen abgebildet werden. Hierfür kommen Feldklassen wie `ForeignKeyField` oder `ManyToManyField` zum Einsatz, die mithilfe entsprechender Parameter (etwa das Zielattribut des referenzierten Modells) die Beziehung zwischen Entitäten eindeutig definieren. Die daraus resultierenden Modelle erlauben eine konsistente und typsichere Abbildung der Datenbankstruktur im Anwendungscode.

Ein weiterer zentraler Bestandteil von Tortoise-ORM ist die Integration mit Pydantic. Durch den Einsatz des `pydantic_model_creator` können automatisch Pydantic-Modelle aus Tortoise-ORM-Modellen generiert werden.

Leider hat sich im Laufe der Entwicklung gezeigt, dass die Methode zur Generierung von Pydantic-Modellen `pydantic_model_creator` sehr unzuverlässig und instabil funktioniert, weshalb er komplett refaktoriert wurde. Die so entstandenen Änderungen sind auch in das offizielle Projekt geflossen nach dem öffentlichen Pull-Request-Prozess von GitHub [24].

3.2.2 Pydantic

Neben der mittels Tortoise-ORM erstellten Pydantic-Modelle werden auch Datenstrukturen für den Login-Prozess und für die Struktur der Cookies definiert. Desweiteren ist es möglich, innerhalb der Tortoise-Modelle zu definieren, welche Felder in das Pydantic-Modell mit aufgenommen werden sollen. So können sensible Daten wie Passwortinformationen zentral herausgefiltert werden. Zudem können sogenannte `computed_fields` definiert werden. Hierbei handelt es sich um Felder, hinter denen eine eigene Funktion steht, die bei der Serialisierung berechnet und in das Ausgabeformat eingefügt werden. Somit bietet der Einsatz von Tortoise-ORM in Kombination mit Pydantic eine robuste, gut wartbare Basis für den Umgang mit persistenten Daten in einer modernen, asynchronen Webanwendung.

3.2.3 FastAPI

In enger Verzahnung mit den zuvor beschriebenen Komponenten bietet das Webframework FastAPI die optimale Ergänzung für den Aufbau moderner, asynchroner Webanwendungen. FastAPI nutzt Pydantic-Modelle, wie sie mithilfe von `pydantic_model_creator` aus Tortoise-ORM-Modellen generiert werden können, um sowohl bei eingehenden Anfragen als auch bei ausgehenden Antworten eine einheitliche und valide Datenstruktur zu gewährleisten. Anfragen an definierte Endpunkte (Routen) durchlaufen eine automatische Validierung auf Basis dieser Pydantic-Modelle, sodass ungültige oder fehlende Daten frühzeitig erkannt und mit aussagekräftigen Fehlermeldungen abgefangen werden können.

Die Integration von FastAPI mit Tortoise-ORM vereinfacht zudem den gesamten Entwicklungsprozess von Backend-Systemen. Durch die nahtlose Verbindung zwischen Datenbankmodellen, validierten Datenstrukturen und Endpunkten entfällt viel manueller Aufwand, wie etwa die Erstellung separater DTOs (Data Transfer Objects) oder eigener Datenvalidierungslogik. Stattdessen liefern die Tortoise-ORM-Modelle und ihre abgeleiteten Pydantic-Modelle bereits alle notwendigen Informationen, um Endpunkte effizient zu implementieren.

Aufbauend auf dieser Infrastruktur wurde ein weiterer Schritt unternommen, um den Entwicklungsaufwand zusätzlich zu minimieren und die Konsistenz der Anwendung zu gewährleisten. Hierzu wurden Methoden implementiert, die aus den Tortoise-ORM-Modellen automatisch entsprechende FastAPI-Routen generieren. Durch das Auslesen der Modellstrukturen und ihrer definierten Beziehungen können Endpunkte für C.R.U.D.-Operationen sowie für den Zugriff auf verknüpfte Entitäten vollautomatisch erstellt werden.

Diese Automatisierung bezieht nicht nur die primären Datenbanktabellen mit ein, sondern berücksichtigt auch die jeweiligen Relationen. Somit entstehen für jede Entität und deren assoziierte Datensätze eigenständige Routen, ohne dass der Entwickler diese manuell definieren muss. Dies führt zu einer erheblichen Zeitersparnis und sorgt für eine einheitliche Struktur in der gesamten API. Daten können dadurch in standardisierter Form abgerufen, manipuliert und mit anderen Entitäten in Beziehung gesetzt werden – all das mit minimalem manuellem Aufwand.

Das Zusammenspiel aus Tortoise-ORM, Pydantic und FastAPI wird durch diese automatisierte Routenerzeugung nochmals aufgewertet, da nun sowohl die Datenvalidierung, das Persistenzmanagement als auch die API-Endpunkt-Generierung aus einer Hand und mit geringer Redundanz erfolgen. Dadurch entsteht ein hochgradig dynamisches, skalierbares und leicht wartbares Backend, das sich ohne großen Mehraufwand an wachsende Anforderungen anpassen lässt.

Als letzte Routen werden zwei Routen definiert, die für die Auslieferung des Frontends zuständig sind. Die erste dieser Routen liefert die statischen Dateien, die durch `npm` generiert wurden, aus. Die zweite ist eine Catch-All-Route, die stets die `index.html`-Datei der SPA ausliefert. Eine Catch-All-Route ist eine Route, die auf alle unbekannten Routen zutrifft.

3.3 Single-Page-Application (SPA) mit React

JavaScript weist in seiner ursprünglichen Form teils erhebliche Unzulänglichkeiten und Inkonsistenzen auf, die es für Entwicklerinnen und Entwickler herausfordernd machen, wartbaren und robusten Code zu verfassen. Durch den Einsatz des React-Frameworks können diese Schwierigkeiten signifikant reduziert werden. React stellt klare Strukturen, wiederverwendbare Komponenten und etablierte Best Practices bereit, die die Softwareentwicklung mit JavaScript insgesamt übersichtlicher und effizienter gestalten.

3.3.1 Gestaltung mit Bootstrap

Bootstrap ist ein Frontend-CSS-Framework, das von Twitter im Jahr 2011 veröffentlicht wurde [25]. Es bietet einen Mobile-First-Ansatz. Das bedeutet, dass die Webseite primär für die Darstellung auf kleinen Displays wie bei Smartphones oder Tablets entwickelt wird. Dieser Ansatz wird verfolgt, da sich in mehreren Studien gezeigt hat, dass primär Smartphones zum Abrufen von Webseiten genutzt werden [26].

Das Layout der Webseite ist aufgeteilt in drei Bereiche: die Kopfzeile, die Seitenleiste und der tatsächliche Seiteninhalt. Während sie auf einem großen Bildschirm, wie auf einem Laptop, komplett dargestellt werden können, siehe Abbildung 3.1, wird bei mobilen Endgeräten die Kopfleiste als ausklappbares Element gestaltet und die Seitenleiste lässt sich durch einen Button am unteren linken Rand öffnen. Dies wird in Abbildung 3.2 dargestellt. Dadurch steht selbst bei kleinen Endgeräten genügend Platz zur Verfügung, um den tatsächlichen Seiteninhalt anzeigen zu können.

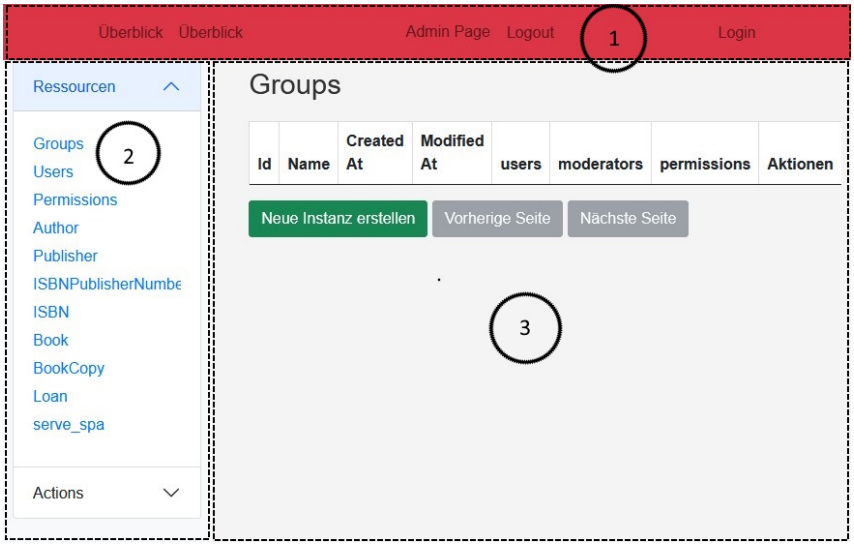
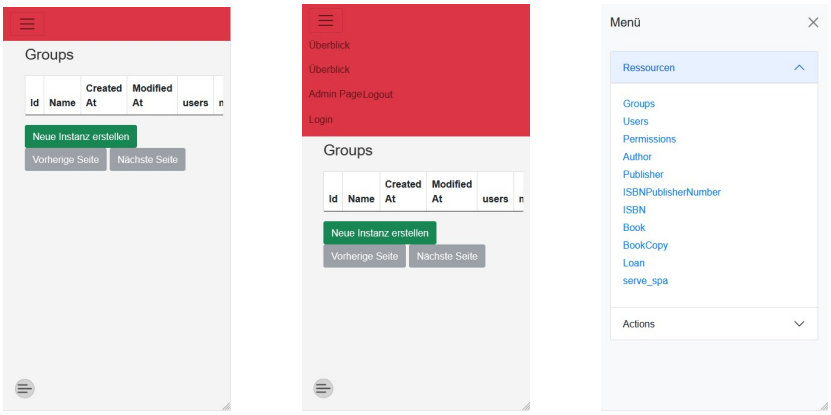


Abb. 3.1: Frontend auf einem Laptop-Bildschirm: (1) Kopfzeile, (2) Seitenleiste, (3) Seiteninhalt



(a) Seiteninhalt im Fokus (b) ausgeklappte Kopfzeile (c) ausgefahrene Seitenleiste

Abb. 3.2: Frontend auf einem Smartphone-Bildschirm

3.3.2 Client-seitige Logik

Da der Client sehr generisch arbeiten und gut auf Veränderungen der zu behandelden Datentypen reagieren können soll, muss darauf geachtet werden, dass die einzelnen Operationen und auch Datentypen stehts zur Verfügung stehen und verarbeitet werden können. An diesem Punkt zeigen sich die Vorteile der zuvor geleisteten Vorarbeit, nämlich dass hierfür durch den Server eine OpenAPI-Spezifikation zur Verfügung gestellt wird. Diese kann von der JavaScript-Bibliothek Swagger-Client konsumiert und ausgewertet werden, wodurch Client-seitig eine direkte Schnittstelle zur REST-API besteht.

Swagger-Client

Im Folgenden wird die JavaScript-Bibliothek Swagger-Client vorgestellt, die in direktem Zusammenhang mit dem OpenAPI-Standard steht. Swagger-Client ermöglicht es, anhand einer vorhandenen OpenAPI-Definition automatisiert auf eine REST-API zuzugreifen, ohne dass einzelne Endpunkte, Parameter oder Datenstrukturen manuell implementiert werden müssen.

Hierbei wird zunächst die entsprechende OpenAPI-Spezifikation (beispielsweise als JSON- oder YAML-Datei) eingelesen. Anschließend generiert Swagger-Client auf dieser Basis eine Client-API, die es erlaubt, sämtliche im OpenAPI-Dokument definierten Routen, Ressourcen und Operationen programmgesteuert und typisiert anzusprechen. Auf diese Weise lassen sich sämtliche HTTP-Methoden und Pfadparameter, ebenso wie Authentifizierungsmechanismen (etwa OAuth 2.0 oder API-Keys), automatisiert nutzen, ohne selbst den kompletten Nachrichtenaufbau oder die Parameterverarbeitung implementieren zu müssen.

Die Verwendung von Swagger-Client erleichtert somit den Integrationsaufwand erheblich. Änderungen an der API müssen in vielen Fällen lediglich in der OpenAPI-Spezifikation angepasst werden, woraufhin Swagger-Client die aktualisierten Strukturen automatisch berücksichtigt. Dadurch erhöht sich nicht nur die Effizienz beim Anbinden und Verwenden komplexer REST-APIs, sondern auch die Wartbarkeit und Zukunftssicherheit des Anwendungscodes.

Formulare mit react-jsonschema-form

Die durch den Swagger-Client extrahierten JSON-Schemata beschreiben den Aufbau eines JSON-Objekts. Es enthält Informationen darüber, welche Attribute ein Objekt haben soll bzw. darf sowie welche Werte diese annehmen dürfen.

Ein Formular, wie es auf einer Webseite zu finden ist, besitzt neben der Zuordnung von eingegebenen Werten zu Attributen auch Beschriftungen und Beschreibungen. Diese sind allerdings auch im JSON-Schema-Format definierbar.

Somit bietet ein JSON-Schema alle notwendigen Informationen, um ein Formular zu erstellen. In Listing 3.1 wird ein JSON-Schema gezeigt, das Name, Nachname und Alter enthält. Das daraus generierte Formular wird in Abbildung 3.3 gezeigt. Wird auf den Button «Submit» geklickt, wird das in Listing 3.2 gezeigte JSON-Objekt versendet.

```
1  {
2          "title": "A registration form",
3          "description": "A simple form example.",
4          "type": "object",
5          "required": [
6                  "firstName",
7                  "lastName"
8          ],
9          "properties": {
10                 "firstName": {
11                         "type": "string",
12                         "title": "First name",
13                         "default": "Chuck",
14                         "description": "Your first name"
15                 },
16                 "lastName": {
17                         "type": "string",
18                         "title": "Last name",
19                         "description": "Your last name"
20                 },
21                 "age": {
22                         "type": "integer",
```

```
                "title": "Age",
                "minimum": 0,
                "maximum": 200
            }
        }
}
```

Listing 3.1: beispielhaftes JSON-Schema

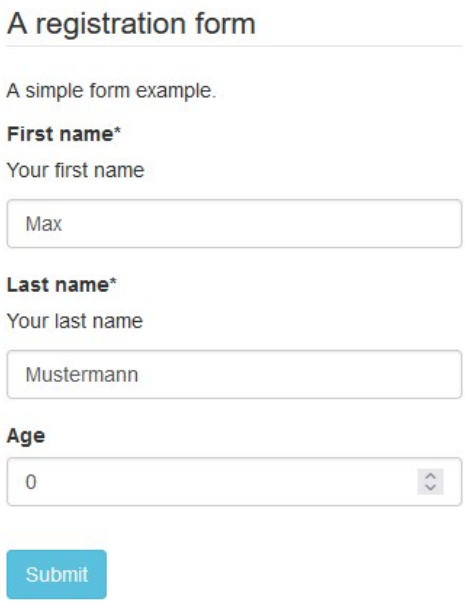


Abb. 3.3: aus dem JSON-Schema generiertes Formular

```
{
    "firstName": "Max",
    "lastName": "Mustermann",
    "age": 0
}
```

Listing 3.2: zu sendendes JSON-Objekt

Unterschiedliche Interpretationen des JSON-Schema-Standards

Die Interoperabilität zwischen Pydantic und der Bibliothek react-jsonschema-form über die durch Pydantic erzeugten JSON-Schemata ist in einigen Fällen gestört. So werden Attribute, die `null` bzw. `None` annehmen dürfen, deutlich anders interpretiert. Während bei einem JSON-Schema, das durch Pydantic erzeugt wird, das Attribut entweder einen Datentyp vom Typ `null`

oder den tatsächlichen Datentyp (wie zum Beispiel `int` oder `str`) annehmen darf, erwartet react-jsonschema-form einfach ein Attribut, das den Zusatz `nullable = true` im Schema besitzt. Daher musste eine Funktion für das React-Frontend entwickelt werden, das zwischen diesen beiden Interpretationen das JSON-Schema konvertiert.

3.3.3 Umgang mit verschiedenen Beziehungstypen

Im Folgenden wird erläutert, wie separate Routen in React genutzt werden, um die unterschiedlichen Entitäten und deren Relationen übersichtlich abzubilden und eine klare Navigationsstruktur in der Benutzeroberfläche sicherzustellen.

Hierfür werden für jede Entität eigene Routen definiert, wie etwa `/resources/Users/new` zum Anlegen neuer Benutzer oder `/resources/Teams/new` zum Erstellen von Teams. Zusätzlich stehen spezielle Routen zur Verfügung, um Relationen anzuzeigen und zu bearbeiten. So kann beispielsweise ein Benutzer um neue, mit ihm verknüpfte Einträge erweitert oder bestehende Verknüpfungen können entfernt werden. Die einzelnen Komponenten – etwa Formulare zum Erstellen oder Bearbeiten von Entitäten – werden dabei dynamisch auf Basis der OpenAPI-Spezifikation generiert. Auf diese Weise leitet das System die erforderlichen Datenmodelle unmittelbar aus der API ab und stellt automatisch passende Eingabemasken sowie Dropdown-Listen für verknüpfte Datensätze bereit.

Durch diese Vorgehensweise lassen sich nicht nur grundlegende C.R.U.D.-Operationen für einzelne Ressourcen umsetzen, sondern auch komplexe Beziehungsstrukturen wie Many-to-Many- und One-to-Many-Beziehungen verwalten.

Bei Änderungen, wie dem Erstellen neuer Objekte in abhängigen Ressourcen, wird durch ein erneutes Laden der relevanten Schemata oder dem erneuten Mounten der entsprechenden Komponenten sichergestellt, dass aktuell hinzugefügte Datensätze (etwa ein neu erstelltes Team) unmittelbar in den Auswahlfeldern anderer Entitäten erscheinen. Dadurch wird gewährleistet, dass die Benutzeroberfläche stets auf dem neuesten Stand ist und mit den zuletzt angelegten oder bearbeiteten Entitäten gearbeitet werden kann.

Insgesamt trägt diese konsequente Nutzung von Routen in Kombination mit dynamischer Schema-Generierung und regelmäßigen Aktualisierungen der UI zu einer flexiblen und erweiterbaren Architektur bei. Sie sorgt für eine klare Trennung zwischen einzelnen Ressourcen und deren Beziehungen und erleichtert damit die Wartung und Skalierbarkeit der Anwendung – sowohl bei wechselnden Modellstrukturen als auch bei sich ändernden Anforderungen.

3.4 Einsatz von JWT

In FastAPI wurde eine Middleware hinzugefügt, die die Validierung des als Cookie mitgelieferten Access-Tokens übernimmt und aus diesem die User-ID extrahiert. Anschließend kann die User-ID von den einzelnen Routen verwendet werden.

Für den Logout-, Login- und Refresh-Vorgang wurden mit FastAPI entsprechende Routen definiert.

4 Implementierung und Testen

Im Folgenden wird darauf eingegangen, welche Software während der Entwicklung konkret eingesetzt wurde. Anschließend werden Tests und Qualitätssicherung thematisiert.

4.1 Verwendete Software

Tabelle 4.1 zeigt die verwendete Software sowie ihre Version.

Tabelle 4.1: verwendete Software

Software/Bibliothek	**Version**	**Hinweis**
Python	3.12.3	als virtual Environment
Tortoise-ORM	0.22.3	mittels `pip` installiert
FastAPI	0.115.6	mittels `pip` installiert
React	11.13.5	mittels `npm` installiert
react-jsonschema-form	5.23.1	mittels `npm` installiert
PyCharm	2023.2.5	Entwicklungsumgebung für Python
Visual Studio Code	1.95.3	Entwicklungsumgebung für JavaScript

4.2 Teststrategien und Qualitätssicherung

Im Folgenden wird auf verwendete Teststrategien sowie Maßnahmen zur Qualitätssicherung, die im Zuge dieser Seminararbeit verwendet wurden, eingegangen. Hierbei wird auf die Strategien und Maßnahmen dieses Projekts sowie der Bibliothek Tortoise-ORM eingegangen.

4.2.1 Teststrategien

Zur Entwicklung des Projekts Tortoise-ORM wird das in Python mitgelieferte Framework Unittest verwendet. Hierbei sind mehr als 1000 Testfälle definiert. Für den in Abschnitt 3.2.1 erwähnten Pull-Request mussten sämtliche Tests bestanden sowie neue Tests hinzugefügt werden.

Weiter wurden für die Software, die im Zuge dieser Arbeit entwickelt wurden, Testfälle geschrieben, die sich mit der Tokenvalidierung befassen. Hierfür wurden Testmethoden geschrieben, die anschließend mit pytest ausgeführt werden. pytest ist eine Bibliothek für Python, die die Definition von Testmethoden erheblich vereinfacht. Im Zuge der Entwicklung der Testmethoden mussten auch verschiedene Klassen durch ein Mockup ersetzt werden, wie zum Beispiel die `Request`-Klasse von FastAPI.

4.2.2 Qualitätssicherung

Während der Entwicklung der Software werden zur Qualitätssicherung verschiedene Linter eingesetzt. Linter werden zur statischen Code-Analyse verwendet [27]. Der Code wird also nicht ausgeführt sondern nur analysiert. Hierbei werden Flake8, Mypy und Pylint eingesetzt. Während Flake8 und Pylint auf den Stil des Codes achten, führt Mypy eine statische Typprüfung der Typ-Annotationen durch. Typ-Annotationen wurden mit der Python-Version 3.5 eingeführt.

Für Python existieren mehrere Konventionen und Best-Practices. Die bekannteste Sammlung von Konventionen für Python ist Python Enhancement Proposal (PEP8). Aus dieser Bezeichnung entstand auch die Bezeichnung für Flake8, es prüft nämlich unter anderem gegen die dort festgehaltenen Konventionen.

Während Python eine dynamisch typisierte Programmiersprache ist, besitzt es den Mechanismus der Typ-Annotation. Typ-Annotationen werden bei der Laufzeit nicht beachtet, jedoch bieten sie einem Entwickler die Möglichkeit, auf eine konsistente Nutzung von übergegebenen Argumenten, die Behandlung von Ausnahmen und ungewöhnlichen Rückgabewerten zu achten. Zudem ist es einer IDE stets möglich nachzuvollziehen, um welchen Datentyp es sich bei einer Variable handelt, was die Entwicklung auch drastisch vereinfacht.

Das Projekt Tortoise-ORM setzt auf den Service von Codacy. Es überprüft die Qualität des Codes von eingereichten Änderungen und ist fest in den Pull-Request-Prozess integriert.

4.3 Beispiele für Testfälle und Ergebnisse

`pydantic_model_creator` - alphabeitsche Sortierung

Für den `pydantic_model_creator` ist es möglich, die Attribute alphabetisch zu ordnen. Hierzu besitzt er das Argument `sort_alphabetically`. Der Code, um diese Funktion zu überprüfen, findet sich in Listing 4.1.

```
1     def test_event_sorted(self):
2         Event_Named = pydantic_model_creator(
3             Event,
4             sort_alphabetically=True
5         )
```

```
6           schema = Event_Named.model_json_schema()
7           self.assertEqual(
8               list(schema["properties"].keys()),
9               [
10                  "address",
11                  "alias",
12                  "event_id",
13                  "modified",
14                  "name",
15                  "participants",
16                  "reporter",
17                  "token",
18                  "tournament",
19              ],
20          )
```

Listing 4.1: Testfall `pydantic_model_creator`, alphabetische Sortierung

Hierbei wird zunächst ein Pydantic-Modell aus dem Tortoise-Modell `Event` generiert, mit dem Argument `sort_alphabetically`. Anschließend wird das hieraus resultierende JSON-Schema abgerufen. In diesem Schema werden im Schlüssel `properties` die einzelnen Attribute aufgelistet als Key-Value-Paare, wobei der Key die Bezeichnung und Value die Beschreibung des Attributs sind. Da hier die Bezeichnungen der Attribute sortiert werden sollen, werden diese Keys als Ist-Wert angesehen. Neben dem Ist-Wert wird ein Soll-Wert benötigt. Der Soll-Wert wird manuell erstellt. Anschließend wird der Ist-Wert mit dem Soll-Wert verglichen mit der durch Unittest zur Verfügung gestellten Methode `assertEqual`. Wenn sie nicht gleich sind, also die Reihenfolge nicht gleich ist oder auch der Inhalt, so ist der Test nicht bestanden und es wird eine entsprechende Fehlermeldung generiert.

`auth_middleware` - erfolgreiche Tokenvalidierung

Die Tokenvalidierung wird als Middleware gelöst, was eine spezielle Form des Testens erfordert. Es muss für das `Request`-Objekt ein Mockup erstellt werden. Wie dies geschieht, ist Listing 4.2 zu entnehmen.

```
1 @pytest.mark.anyio
2 async def test_auth_middleware_valid_token(self, client:
    ↪ AsyncClient):
3     sub = str(uuid.uuid4())
4
5     async def check_if_user_uuid_is_set(request: Request) ->
        ↪ Response:
6         assert request.state.user_uuid == uuid.UUID(sub)
7         return Response()
```

```
8
9       expiration = datetime.now(timezone.utc) + final_config.
            ↪ jwt_access_expires
10      jti = str(uuid.uuid4())
11      to_encode = {"exp": expiration, "sub": sub, "type": "access
            ↪ ", "jti": jti}
12      encoded_jwt = encode_token(to_encode)
13      cookies = {"access_token": encoded_jwt, "cookie2": "value2"
            ↪ }
14      req = AsyncMock(spec=Request)
15      req.cookies = cookies
16
17      await auth_middleware(req, check_if_user_uuid_is_set)
```

Listing 4.2: Testfall `auth_middleware`, erfolgreiche Tokenvalidierung

Hier wird ein JWT aufgebaut, so wie er auch im tatsächlichen Programm erstellt wird, jedoch ist «sub» ein zufällig generierter Wert und nicht der eines tatsächlichen Benutzers. Anschließend wird er encodiert. Es wird ein Dictionary angelegt, das diesen Token als `access_token` enthält, was einen Satz von Cookies darstellen soll. Da es nicht ohne weiteres möglich ist, ein `Request`-Objekt zum Testen zu erstellen, wird es als Mockup abstrahiert. Hierbei wird die von Unittest mitgelieferte Klasse `AsyncMock` verwendet. Die Cookies werden dem Mockup-Objekt zugewiesen. Abschließend wird die Middleware direkt aufgerufen, mit dem `Request`-Objekt und einer Funktion, die dem Handler der Route entspricht, als Argumente. Innerhalb dieses Handlers wird überprüft, ob die im Request gesetzte ID des Benutzers mit der des unter «sub» abgespeicherten ID übereinstimmt. Ist dies der Fall, so konnte ID des Benutzers erfolgreich aus dem im Request mitgelieferten Cookie extrahiert werden.

5 Diskussion und Ausblick

Die Entwicklung moderner Verwaltungssoftware stellt vielfältige Herausforderungen dar, insbesondere in Bezug auf die effiziente Abbildung komplexer Datenstrukturen und die Minimierung des Entwicklungsaufwands. Ziel dieser Arbeit war es, durch den Einsatz eines Low-Code-Ansatzes eine Software zu entwickeln, die diesen Anforderungen gerecht wird. Dabei sollte die Automatisierung wesentlicher Entwicklungsprozesse den Aufwand reduzieren und gleichzeitig eine skalierbare sowie sichere Systemarchitektur gewährleisten.

Zur Umsetzung dieses Ziels wurde ein Ansatz gewählt, der auf der Kombination moderner Technologien wie React, FastAPI und Tortoise-ORM basiert. Während React als Framework für die Entwicklung der Benutzeroberfläche fungierte, bot FastAPI eine effiziente Möglichkeit, REST-APIs zu implementieren. Tortoise-ORM diente als Grundlage für die Modellierung der Datenstruktur und die Verwaltung der Datenbank. Als Schnittstelle dieser Technologien diente Pydantic als Validierungswerkzeug. Die Integration dieser Technologien ermöglichte eine nahtlose Verbindung zwischen Backend und Frontend und trug zur Reduktion redundanter Implementierungen bei.

Die Ergebnisse der Arbeit zeigen, dass durch den gewählten Ansatz eine funktionale und skalierbare Verwaltungssoftware entwickelt werden konnte. Insbesondere die automatische Generierung von C.R.U.D.-Operationen und die Abbildung komplexer Relationen wie Many-to-Many-Beziehungen erwiesen sich als effizient. Auch die Verwendung von JSON-Schema und OpenAPI zur Automatisierung der Frontend-Generierung zeigte deutliche Vorteile. Dennoch wurde im Entwicklungsprozess deutlich, dass die Interoperabilität zwischen den eingesetzten Technologien, etwa Pydantic und react-jsonschema-form, in einigen Bereichen eingeschränkt ist. Diese Herausforderungen konnten jedoch durch zusätzliche Anpassungen adressiert werden.

Ein weiterer Punkt, der das Potenzial des gewählten Ansatzes unterstreicht, ist die Sicherheit. Durch die Verwendung von JSON-Web-Tokens (JWT) wurden moderne Authentifizierungsmechanismen implementiert, die insbesondere in zustandslosen Architekturen von Vorteil sind. Dennoch wäre eine differenzierte Rechteverwaltung ein sinnvoller nächster Schritt, um die Software für den Einsatz in realen Szenarien weiter zu optimieren. Hierbei sind zwei Herangehensweisen denkbar: (1) es werden pro Route eigene Rechte definiert oder (2) jede Ressource hat einen eigenen Rechtevektor, ähnlich wie man ihn bei Unix-Dateisystemen (z. B. ext4) finden kann. Zu (2) muss hierzu festgehalten werden, welcher Benutzer und welche Gruppe eine Ressource besitzt sowie welche Rechte Benutzer, Gruppe oder Jeder bezüglich dieser Ressource besitzt: Lesen, Schreiben und Ausführen. Es handelt sich damit um drei Bits, die zusammen genommen den Rechtevektor bilden.

Für zukünftige Arbeiten bieten sich verschiedene Erweiterungsmöglichkeiten. Eine Implementierung von Workflows, etwa zur Abbildung spezifischer Benutzerinteraktionen, könnte die Software noch besser an die Bedürfnisse der Anwender anpassen. Schließlich könnte die Integration weiterer Low-Code-Werkzeuge wie zum Beispiel ein BPMN-Editor zur Administration von Workflows das System weiter verbessern.

Insgesamt zeigt die Arbeit, dass die gewählte Kombination aus Technologien und Ansätzen eine leistungsfähige Grundlage für die Entwicklung moderner Verwaltungssoftware bietet. Die erzielten Ergebnisse bilden eine solide Basis, auf der zukünftige Verbesserungen und Erweiterungen aufbauen können. Mit einem klaren Fokus auf Skalierbarkeit, Benutzerfreundlichkeit und Sicherheit erfüllt die entwickelte Software die zentralen Anforderungen und bietet zugleich Raum für innovative Weiterentwicklungen.

Literatur

[1] Frank BUSCHMANN u. a. Pattern-Oriented Software Architecture, Volume 1, A System of Patterns. 1., Auflage. Wiley Series in Software Design Patterns. New York, NY: John Wiley & Sons, 2013. ISBN: 1118725263.

[2] Alex BANKS. Learning React: Functional web development with React and Redux. First edition. Sebastopol, CA: O'Reilly Media, 2017. URL: https://learning.oreilly.com/library/view/-/9781491954614/?ar.

[3] React - A JavaScript library for building user interfaces. https://reactjs.org. Zugriff am 10. Dezember 2024.

[4] Peter Pin-Shan CHEN. The entity-relationship model—toward a unified view of data. In: ACM Trans. Database Syst. 1.1 (März 1976), S. 9–36. ISSN: 0362-5915. DOI: 10.1145/320434.320440. URL: https://doi.org/10.1145/320434.320440.

[5] Martin FOWLER. Patterns of Enterprise Application Architecture. Pearson Deutschland, 2002, S. 560. ISBN: 9780321127426. URL: https://elibrary.pearson.de/book/99.150005/9780133065206.

[6] Tortoise ORM v0.22.2 Documentation. Zugriff am 10. Dezember 2024. URL: https://tortoise.github.io/.

[7] Changelog - Tortoise ORM v0.22.2 Documentation. Zugriff am 10. Dezember 2024. URL: https://tortoise.github.io/CHANGELOG.html.

[8] Roy Thomas FIELDING und Richard N. TAYLOR. Architectural styles and the design of network-based software architectures. AAI9980887. Diss. 2000. ISBN: 0599871180. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[9] Leonard RICHARDSON. RESTful Web APIs / Leonard Richardson and Mike Amundsen; foreword by Sam Ruby. eng. North Sebastopol, California: O'Reilly, 2013. ISBN: 9781449359744.

[10] Tim BERNERS-LEE, Roy T. FIELDING und Larry M MASINTER. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986. Jan. 2005. DOI: 10.17487/RFC3986. URL: https://www.rfc-editor.org/info/rfc3986.

[11] Chris J. DATE. An introduction to database systems / C. J. Date. eng. 6. ed., reprint. with corr. «The» systems programming series. Reading, Mass. [u.a: Addison-Wesley, 1995. ISBN: 020154329X.

[12] E. F. CODD. A relational model of data for large shared data banks. In: Commun. ACM 13.6 (Juni 1970), S. 377–387. ISSN: 0001-0782. DOI: 10.1145/362384.362685. URL: https://doi.org/10.1145/362384.362685.

[13] James MARTIN. Managing the Data Base Environment. 1st. USA: Prentice Hall PTR, 1983. ISBN: 0135505828.

[14] Darrel MILLER u. a., Hrsg. OpenAPI Specification (Version 3.1.0). Online. Zugriff am 16.12.2024. Feb. 2021. URL: https://spec.openapis.org/oas/v3.1.0.

[15] Michael B. JONES, John BRADLEY und Nat SAKIMURA. JSON Web Token (JWT). RFC 7519. Mai 2015. DOI: 10.17487/RFC7519. URL: https://www.rfc-editor.org/info/rfc7519.

[16] Dick HARDT. The OAuth 2.0 Authorization Framework. RFC 6749. Okt. 2012. DOI: 10.17487/RFC6749. URL: https://www.rfc-editor.org/info/rfc6749.

[17] Nat SAKIMURA u. a., Hrsg. OpenID Connect Core 1.0. Online. Zugriff am 16.12.2024. Nov. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html.

[18] Mozilla CORPORATION. SPA (Single-page application). Zugriff am 13.12.2024. URL: https://developer.mozilla.org/en-US/docs/Glossary/SPA.

[19] WHATWG. HTML Living Standard - 7.4 Navigation and session history. Living Standard, Online. Zugriff am 13.12.2024. URL: https://html.spec.whatwg.org/multipage/browsing-the-web.html.

[20] Michael S. MIKOWSKI und Josh C. POWELL. Single page web applications : JavaScript end-to-end / Michael S. Mikowski ; Josh C. Powell. eng. Shelter Island: Manning, 2014. ISBN: 9781617290756.

[21] John RYMER und Clay RICHARDSON. The Forrester Wave™: Low-Code Development Platforms, Q2 2021. Techn. Ber. Zugriff am 13.12.2024. Forrester Research, Inc., 2021.

[22] Anca-Raluca BREJE u. a. Comparative Study of Data Sending Methods for XML and JSON Models. In: International Journal of Computer Science and Information Security (IJCSIS) 9.12 (2018), S. 199–204.

[23] Ulysse CARION. JSON Type Definition. RFC 8927. Nov. 2020. DOI: 10.17487/RFC8927. URL: https://www.rfc-editor.org/info/rfc8927.

[24] Markus BECKSCHULTE. Pull Request #1745: pydantic_model_creator refactorization. URL: https://github.com/tortoise/tortoise-orm/pull/1745 (besucht am 13. 12. 2024).

[25] Bootstrap from Twitter. Zugriff am 13.12.2024. 2011. URL: https://blog.x.com/developer/en_us/a/2011/bootstrap-twitter.

[26] We Are SOCIAL und MELTWATER. Digital 2023: Global Overview Report. Zugriff am 13.12.2024. 2023. URL: https://datareportal.com/reports/digital-2023-global-overview-report.

[27] MATTHEW WILKES. Advanced Python Development: Using Powerful Language Features in Real-World Applications. 1st edition. Berkeley, CA: Apress, 2020. ISBN: 9781484257937. DOI: 10.1007/978-1-4842-5793-7.

[28] Congxiao BAO u. a. IP/ICMP Translation Algorithm. RFC 7915. Juni 2016. DOI: 10.17487/RFC7915. URL: https://www.rfc-editor.org/info/rfc7915.