Hochschule Aalen

Projektbericht
Studiengang : Informatik

Projektarbeit AR-Schnitzeljagd

von

Lukas Steckbauer, Rosario Aranzulla

85836, 85816

Betreuender Mitarbeiter : Dr. Marc Hermann

Einreichungsdatum : 13.08.2024

Eidesstattliche Erklärung

Hiermit erkläre ich, **Lukas Steckbauer, Rosario Aranzulla**, dass ich die vorliegenden Angaben in dieser Arbeit wahrheitsgetreu und selbständig verfasst habe.

Weiterhin versichere ich, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, dass alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ort, Datum Unterschrift (Student)

Kurzfassung

Die vorliegende Projektarbeit dokumentiert eingehend das bearbeitete Projekt *AR-Schnitzeljagd*. Das Projekt umfasst den Entwurf, die Implementierung und Inbetriebnahme eines Schnitzeljagd-Editors, welcher zur Erstellung und zum Bearbeiten bestehender Schnitzeljagdten Verwendung findet. Zudem wurde eine Smartphone-Anwendung für die Durchführung einer Schnitzeljagd mit Augmented-Reality-Unterstützung entwickelt.

Im Folgenden wird ein umfassender Einblick in die technischen Details der Projektrealisierung dargelegt, angefangen mit einer Einführung in projektrelevante Frameworks, Technologien und Libraries, der Konzeption und Entwurf, bis hin zur konkreten Implementierung und im Projektdurchlauf entstandene Hindernisse.

Im Rahmen der Projektarbeit wurden vielfältige Aspekte der Software-Entwicklung berührt, wobei der Schwerpunkt auf der Entwicklung mit Micro-Services, Interprozesskommunikation über REST-API und dem Umgang mit Augmented-Reality-Bibliotheken lag.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

1. Einleitung

1.1. Motivation

Die ersten Tage und Wochen an der Hochschule können für neue Studierende eine herausfordernde und überwältigende Zeit sein. Sie müssen sich in einer neuen Umgebung zurechtfinden, viele neue Menschen kennenlernen und gleichzeitig den akademischen Anforderungen gerecht werden. Eine effektive Methode, um den Einstieg zu erleichtern, ist eine Einführungstour, die jedoch oft nur begrenzt interaktiv und ansprechend ist.

Traditionelle Einführungstouren sind meist sehr statisch und linear. Studierende folgen einer festgelegten Route und hören passiv zu, während Informationen vermittelt werden. Dies führt oft dazu, dass wichtige Details nicht in Erinnerung bleiben, da die Interaktion und das eigenständige Entdecken fehlt.

Zudem bieten solche Touren selten die Möglichkeit zur aktiven Teilnahme und Mitgestaltung. Die Studierenden sind Zuschauer statt Akteure, was die Aufnahmefähigkeit und das Engagement reduziert. Ohne die Möglichkeit, selbst Entscheidungen zu treffen oder Aufgaben zu lösen, bleibt der Lerneffekt gering und die Tour wird schnell langweilig.

Schließlich ist die Interaktion zwischen den neuen Studierenden bei herkömmlichen Einführungstouren meist eingeschränkt. Da der Fokus auf der Vermittlung von Informationen liegt, bleibt wenig Raum für soziale Interaktionen und Teamarbeit. Somit fällt es den Studierenden schwer, frühzeitig Kontakte zu knüpfen, um eine Lerngruppe zu finden und ein Gemeinschaftsgefühl entwickeln zu können.

Diese Herausforderungen verdeutlichen die Notwendigkeit für innovativere und interaktivere Ansätze, wie sie durch eine Schnitzeljagd geboten werden können.

1.2. Zielsetzung

Das Ziel dieses Projekts ist die Entwicklung eines Systems zur Erstellung, Anmeldung und Durchführung von Schnitzeljagden an der Hochschule Aalen. Durch eine Schnitzeljagd werden die Studierenden nicht nur spielerisch mit den verschiedenen

Gebäuden, Räumen und Einrichtungen vertraut gemacht, sondern auch zur aktiven Teilnahme und Zusammenarbeit angeregt. Dies fördert nicht nur das Verständnis der Campusstruktur, sondern auch das Gemeinschaftsgefühl und den Zusammenhalt unter den neuen Studierenden.

1.3. Vorgehen

Um das beschriebene Projekt zu realisieren, wurden folgende Projekt-Phasen durchlaufen:

Technologische Forschungsphase

Die Durchführung einer Schnitzeljagd sollte über das Smartphone erfolgen. Im Verlauf der Jahre haben sich einige Möglichkeiten, Software für mobile Geräte zu entwickeln, durchgesetzt, die jeweils ihre Vor- und Nachteile besitzen. In dieser Projektphase war es wichtig, die vielen unterschiedlichen Technologien zu erforschen und eine für das Projekt passende Plattform zu wählen, in welcher die Durchführung der Schnitzeljagden erfolgen kann.

Entwurfsphase

Nachdem das grundlegende, projektrelevante technologische Wissen verfeinert worden war, wurde ein erster Entwurf der fundamentalen Architektur vorgestellt. Nachdem dieser ausreichend ausgereift war, konnten erste Workflows der Benutzer erstellt werden. Sobald die Baseline für die Implementierung festgelegt worden war, konnten erste Aufgaben verteilt werden.

Implementierungsphase

Die einzelnen Funktionalitäten wurden daraufhin implementiert und getestet. Hierbei war es entscheidend, die geplanten Features schrittweise zu realisieren und kontinuierlich mit den Anforderungen zu überprüfen.

2. Grundlagen

2.1. Software Design

2.1.1. Einführung

Software-Design ist ein zentraler Aspekt der Softwareentwicklung, der maßgeblich den Erfolg und die Wartbarkeit eines Projekts beeinflusst. Verschiedene architektonische Patterns bieten Lösungen für wiederkehrende Probleme und helfen Entwicklern, robuste und skalierbare Anwendungen zu erstellen. Robert C. Martin, ein Pionier in der Software-Architektur, betont in seinem Buch *Clean Architecture* die Bedeutung von guten Architekturen:

> The goal of software architecture is to minimize the human resources required to build and maintain the required system. [1]

Dieses Zitat unterstreicht, dass eine durchdachte Architektur nicht nur die Entwicklungszeit verkürzt, sondern auch die langfristige Wartung vereinfacht. In diesem Kontext werden im Folgenden zwei bedeutende architektonische Patterns vorgestellt, die im Rahmen der Projektdurchführung relevant sind: Domain-driven Design (DDD) und Microservices.

2.1.2. Architektonische Patterns und Ansätze

Microservices

Unter dem Begriff *Microservices* versteht sich der Ansatz, ein Software-System als Orchestration mehrerer Dienste zu unterteilen. Jeder Dienst ist für eine Aufgabe des gesamten System-Kontexts verantwortlich und kann unabhängig von anderen Diensten als ein eigener Prozess laufen. [2]

Domain Driven Design

Das *DDD* ist eine Methodologie, die hauptsächlich bei komplexen Softwareprojekten zum Einsatz kommt. Es gehört weder zur Kategorie der Frameworks noch zu den architektonischen Patterns. Vielmehr ist es eng mit der Denkweise der Microservices, wie in Kapitel 2.1.2 beschrieben, verbunden. Domain-driven Design konzentriert sich darauf, Aktivitäten und Prozesse aus dem realen Leben in die Softwareentwicklung zu abstrahieren. [2]

Onion-Architecture

Die *Onion-Architecture*, auch bekannt als *Clean Architecture*, wurde von Jeffrey Palermo entwickelt und wird insbesondere für C#-Projekte von Microsoft empfohlen. Sie zielt darauf ab, die typischen Probleme monolithischer Anwendungen, wie hohe Kopplung und geringe Wartbarkeit, zu vermeiden. Die Architektur visualisiert die Software als konzentrische Schichten, vergleichbar mit den Schichten einer Zwiebel. Folgende Abbildung stellt diesen Aufbau nochmals dar:

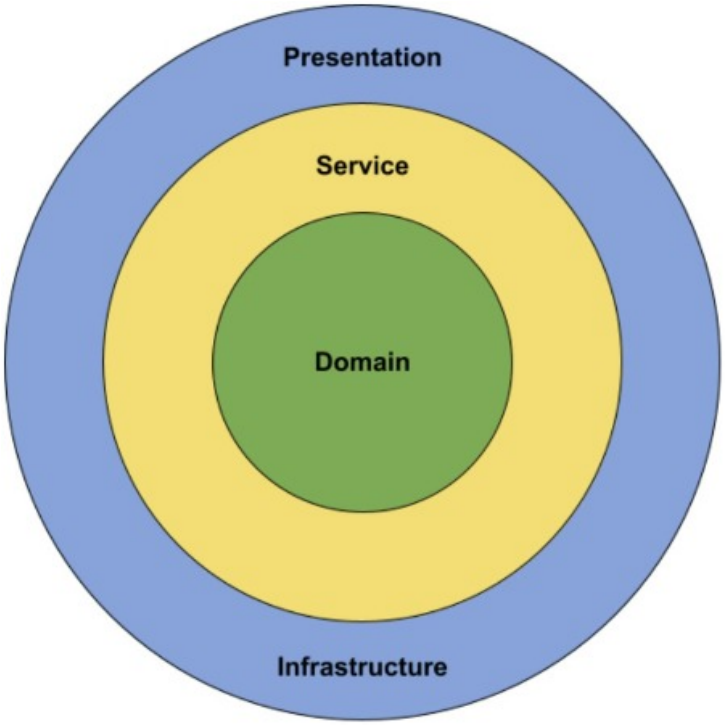

Abbildung 2.1.: Bild Onion Architecture, nach [3]

Die Onion-Architecture gliedert sich in vier Hauptschichten:

- **Domain Layer:** Im Zentrum steht die Domäne, die Geschäftslogik und Geschäftsregeln beinhaltet und keinerlei Abhängigkeiten zu äußeren Schichten hat.
- **Service Layer:** Diese Schicht enthält die Anwendungsspezifische Logik und nutzt die in der Domain Layer definierten Schnittstellen.
- **Infrastructure Layer:** Hier befinden sich Implementierungen für Datenzugriff, Netzwerkkommunikation und andere externe Dienste.
- **Presentation Layer:** Diese Schicht ist für die Benutzeroberfläche und die Präsentation der Daten verantwortlich.

Konzepte der Onion-Architecture fördern die Abhängigkeit von Abstraktionen (Interfaces) anstatt von konkreten Implementierungen. Diese Abhängigkeitsinversion erlaubt es, Implementierungen zur Laufzeit auszutauschen, was die Flexibilität und Erweiterbarkeit der Software erhöht.

Die Vorteile der Onion-Architecture sind:

- **Hohe Testbarkeit:** Da alle Schichten über definierte Schnittstellen kommunizieren, können einzelne Komponenten leicht isoliert und getestet werden.
- **Klare Abhängigkeiten:** Abhängigkeiten fließen strikt in Richtung der zentralen Domänenschicht, wodurch höhere Schichten die Implementierungen der unteren Schichten verwenden können, aber nicht umgekehrt.
- **Trennung von Geschäftslogik und Implementierungsdetails:** Die Geschäftslogik kann unabhängig von technischen Implementierungsdetails entwickelt werden. Notwendige Schnittstellen zu externen Systemen werden definiert, aber deren konkrete Implementierung wird in den äußeren Schichten gekapselt.

Diese Struktur ermöglicht es, komplexe Anwendungen modular zu entwickeln und erleichtert die Wartung und Weiterentwicklung der Software [3].

2.1.3. Kollaboration mehrerer Dienste

Eventbasierte Architektur: Message Broker und Message Bus

Eine eventbasierte Architektur ermöglicht es, Nachrichten schnell und flexibel zwischen mehreren Diensten auszutauschen. Ein verbreiteter Ansatz hierfür ist

die Nutzung eines *Message-Brokers*. Im Folgenden werden die vier grundlegenden Aspekte dieser Architektur beschrieben (vgl. [4]):

1. **Initiierendes Event**: Ein *Ereignis*, das den Event-Fluss anstößt und an einen Event-Kanal des Event-Brokers gesendet wird.
2. **Event-Broker**: Ein *Orchestrator*, der mehrere Kanäle verwaltet, auf denen Event-Prozessoren lauschen können.
3. **Event-Prozessor**: Eine Einheit, die auf einem Kanal lauscht und ein eingehendes Event verarbeitet. Dabei kann ein neues (verarbeitetes) Event erzeugt und an einen anderen Kanal gesendet werden.
4. **Das zu verarbeitende Event**: Ein Ereignis, das durch die beschriebenen Mechanismen verarbeitet oder erzeugt wird.

Vorteile

1. **Architektonische Erweiterbarkeit**: Events können vorläufig erzeugt und an den Message-Broker gesendet werden, ohne dass der Event-Prozessor bereits implementiert sein muss. Dies ermöglicht die spätere Implementierung des Prozessors.
2. **Abkapselung**: Jedes Modul ist für seine eigene Funktionalität verantwortlich, während alles außerhalb Bestandteil anderer Module ist.
3. **Skalierbarkeit**: Events können von mehreren (gleichen) Event-Prozessoren verarbeitet werden. Der Kanal fungiert als FIFO-Warteschlange und ermöglicht eine notwendige Synchronisation.
4. **Asynchronität**: Die Kommunikation erfolgt asynchron, was die Entkopplung der Dienste und die Verbesserung der Systemleistung ermöglicht.

Nachteile

1. **(De-)Marshalling**: Nachrichten müssen in ein anderes Format (z.B. JSON, String, Byte-Array) konvertiert und wieder zurückkonvertiert werden. Dies kann die Performance beeinträchtigen.
2. **Policies**: In einer traditionellen Message-Broking-Architektur kann theoretisch jeder auf einen Kanal *subscriben*. Es gibt keine klaren Schichten zur Trennung der Zugriffsrechte.
3. **Komplexität**: Das Management mehrerer Module und die Nachverfolgung von *Event-Publishern* und *Event-Subscriber* können schwierig sein.

RESTful APIs

Für die Kommunikation und den Datenaustausch zwischen mehreren Diensten stehen verschiedene Kommunikations-Protokolle zur Verfügung. HTTP ist ein standardisiertes Kommunikationsprotokoll und ist in Bereichen der Web-Anwendungen weit verbreitet.

Representational State Transfer (REST) definiert kein neues Kommunikationsprotokoll. Es ist ledigliche eine Sammlung von Architekturbeschränkungen und definiert Regeln, die der Entwickler befolgen muss, um eine Zustandslosigkeit in der Kommunikation zu erfüllen [5].

2.2. Technologien

Für die Implementierung wurden zahlreiche Technologien der Software-Entwicklung in Betracht gezogen. In diesem Abschnitt werden die für das Projekt relevanten Frameworks und Plattformen beschrieben.

2.2.1. ASP.NET

ASP.NET ist ein Open-Source-Web-Framework, das von Microsoft entwickelt wurde und die Erstellung moderner, skalierbarer und leistungsfähiger Webanwendungen ermöglicht. Zudem werden in ASP.NET moderne Webtechnologien und -standards unterstützt, wodurch sich die Entwicklung von interaktiven und responsiven Webanwendungen erleichtert. Dies schließt auch APIs und Echtzeit-Kommunikation ein, die für die Anwendung relevant sein könnten. Diese Merkmale ermöglichen es auch, die Anwendung modular zu ergänzen, indem externe APIs wie OpenStreetMaps oder andere Dienste einfach angebunden werden können.

Entity Framework Core

Entity Framework Core (EF Core) ist ein leichtgewichtiges, erweiterbares und Open-Source-Object-Relational Mapping (ORM) Framework für .NET, das entwickelt wurde, um den Datenzugriff und die Datenmanipulation in .NET-Anwendungen zu vereinfachen. EF Core bietet Datenbankunabhängigkeit, da es verschiedene Datenbanksysteme wie SQL Server, MySQL, PostgreSQL und SQLite unterstützt.

Ein weiterer Vorteil von EF Core ist Möglichkeit, das Datenbankschema aus konkreten Models über Source-Code zu generieren, ohne *Create-Table* SQL-Statements schreiben zu müssen. EF Core bietet hierbei eine einfach zu verstehende *Fluent-API*

an. Zudem ermöglicht EF Core die einfache Verwaltung von Datenbankmigrationen, was es erlaubt, Änderungen am Datenmodell nachzuverfolgen und auf die Datenbank anzuwenden. Dies erleichtert die Wartung und Weiterentwicklung erheblich.

Ein zusätzliches Merkmal von EF Core ist die Integration von Language Integrated Query (LINQ), die es ermöglicht, komplexe Abfragen auf eine intuitive und typsichere Weise zu schreiben. Dies verbessert die Lesbarkeit und Wartbarkeit des Codes erheblich. Schließlich bietet EF Core verschiedene Mechanismen zur Performanceoptimierung, wie zum Beispiel asynchrone Abfragen und Caching-Strategien.

2.2.2. Unity und Mobile-Apps

Unity ist eine Plattform zur Entwicklung und Darstellung interaktiver 3D-Inhalte, die in vielen Bereichen wie Spieleentwicklung, Filmproduktion, Architekturvisualisierung und Virtual Reality (VR) eingesetzt wird. Sie ist eine der am weitesten verbreiteten Spiel-Engines weltweit und bietet eine Vielzahl an Werkzeugen und Funktionen, für das implementieren immersiver und realistischer Umgebungen.

Unity basiert auf einer Engine, die eine umfassende Sammlung von Software-Bibliotheken bereitstellt. Diese Bibliotheken ermöglichen die Verarbeitung von Grafiken, Physik, Sound und Benutzereingaben. Der Kern der Engine ist für die Rendern von 2D- und 3D-Grafiken verantwortlich, die in Echtzeit berechnet und auf dem Bildschirm angezeigt werden. Unity unterstützt die Programmiersprachen C# und UnityScript (eine Art JavaScript, welche allerdings nicht mehr weiterentwickelt wird), wobei C# die primäre Sprache für die Entwicklung in Unity ist.

AR-Foundation

Für das Erstellen von Augmented-Reality Anwendungen bietet Unity unter der *AR Foundation* grundlegende Bausteine an. Über vorgefertigte Projekt-Templates lässt sich eine funktionierende und ausführbare Demo-Anwendung für Android-Geräte generieren. Das AR Foundation Framework ermöglicht das Einbinden von Features wie beispielsweise Oberflächen-Erkennung, Objekt-, Bild- und Gesicht-Verfolgung sowie Sitzungs- und Geräte-Management [6].

Nützliche Bibliotheken

Newtonsoft.Json.NET ist eine in der Praxis verwendete Bibliothek in C# bzw. .NET Anwendungen für das Serialisieren und Deserialisieren von Objekten im JSON-Format. Für die Verwendung in Unity-Anwendungen existiert ein GitHub-Fork

(*Json.Net.Unity3D*) und ermöglicht das Einbinden über eine `.unitypackage` Datei. [7]

AR+GPS Location ist eine Utility aus dem Unity-Asset-Store und ermöglicht eine Übersetzung von GPS-Koordinaten im Unity-Raum. Hierdurch können Objekte über die Angabe eines realen GPS-Standorts im Unity-Raum plaziert werden. Wenn die Anwendung an der realen Stelle des angegebenen GPS-Standorts geöffnet wird, wird das in Unity angelegte Objekt am richtigen Ort angezeigt. [8]

2.2.3. Svelte und Web-Apps

Svelte ist ein modernes JavaScript-Framework zur Erstellung von Benutzeroberflächen. Im Gegensatz zu herkömmlichen Frameworks wie React oder Vue, die den Großteil ihrer Arbeit im Browser ausführen, verschiebt Svelte diese Arbeit in die Kompilierungsphase. Dies bedeutet, dass der Code während des Build-Prozesses in effizientes, optimiertes JavaScript umgewandelt wird, das direkt im Browser ausgeführt wird. Dadurch wird die Laufzeitbelastung erheblich reduziert, was zu einer besseren Performance und geringeren Ladezeiten führt [9].

Ein wesentlicher Vorteil von Svelte ist seine einfache Syntax, die es Entwicklern ermöglicht, reaktiven Code zu schreiben, ohne auf komplexe State-Management-Lösungen zurückzugreifen. Die Komponenten in Svelte bestehen aus HTML, CSS und JavaScript, was die Lernkurve für neue Benutzer verringert und die Entwicklungserfahrung vereinfacht [9].

SvelteKit ist ein Framework für den Aufbau von Svelte-Anwendungen. Es erweitert die Fähigkeiten von Svelte, indem es zusätzliche Werkzeuge und Funktionen bereitstellt, die speziell für die Entwicklung komplexer, leistungsstarker Webanwendungen notwendig sind. SvelteKit vereinfacht die Einrichtung und Strukturierung von Projekten und bietet Funktionen wie Routing, Server-Side Rendering (SSR), statische Seitengenerierung und eine integrierte Entwicklungsumgebung [9].

Vorteile

Svelte integriert CSS-Styles direkt in die Komponenten mittels <*style*>-*Tags*. Dies eliminiert die Notwendigkeit zusätzlicher Setups.

Alternativ kann auf bereits existierende CSS-Bibliotheken wie TailwindCSS zurückgegriffen werden. Standardmäßig werden Styles scoped angelegt, sodass keine Konflikte zwischen den Komponenten entstehen. Dies sorgt für eine sichere und wartbare Codebase einzelner Komponenten aufgrund der reduzierten Abhängigkeiten.

Svelte-Pages sind standardmäßig mit einem einfachen Prerendering ausgestattet, das sich über das Setzen eines Attributes steuern lässt. SvelteKit bietet zudem einfache Einstellungen, um verschiedene Startpfade für das Prerendering zu definieren, was die Flexibilität bei der Konfiguration erhöht.

Ein weiteres zentrales Feature von Svelte ist die Reaktivität. Durch das Definieren von sogenannten *Reactive Statements* kann ein Ablauf beschrieben werden, der ausgeführt wird, wenn eine Abhängigkeit innerhalb dieses Blocks aktualisiert wird. Dies erleichtert das Management von Zustandsänderungen und sorgt für reaktive und dynamische Benutzeroberflächen.

Svelte besitzt eingebaute Prüfungen für Barrierefreiheit (*Accessibility-Linting*), die auf offensichtliche und auch auf komplexere Probleme hinweisen. Dies unterstützt Entwickler dabei, von Anfang an barrierefreie Anwendungen zu erstellen.

Letztendlich reduziert sich die Menge des notwendigen Codes erheblich durch die deklarative Natur von Svelte. Diese führt zu einer klareren und übersichtlicheren Codebasis, was die Entwicklung und Wartung von Anwendungen vereinfacht [10].

Nachteile

Die Community-Größe von Svelte ist im Vergleich zu bekannteren JavaScript-Frameworks wie React oder Angular aufgrund der Neuheit des Frameworks noch relativ klein. Dies kann die Problemlösung erschweren, da einige Fehler möglicherweise nicht so gut dokumentiert oder gelöst sind und somit weniger Ressourcen auf Foren wie Stack Overflow zur Verfügung stehen.

Ein weiterer Nachteil ist das fehlende Entwickler-Tooling oder eine Command Line Interface (CLI). Bei Angular können Komponenten oder Dienste beispielsweise über die Angular CLI schnell und effizient erstellt werden. Diese Art von Tooling fehlt bei Svelte, was den Entwicklungsprozess etwas weniger komfortabel machen kann.

Zudem ist die Verfügbarkeit und Nutzbarkeit von bereits vorhandenen Libraries eingeschränkt. Obwohl diese Libraries nahtlos mit einem Paketmanager (z.B. npm von Node.js) installiert werden können, erfordert es oft zusätzlichen Aufwand, um sie in Svelte-Komponenten zu integrieren [11].

Pages & Routes

Eine Svelte-Anwendung ist verzeichnisbasiert aufgebaut. Jedes Verzeichnis entspricht einer Route und kann ein oder mehrere Unterverzeichnisse haben, die die Route erweitern. Jede Route kann eine Seite haben, die immer den Namen

+*page.svelte* trägt. Svelte verwendet diese Seiten als Hauptanzeige, wenn zur zugehörigen Route navigiert wird.

UI Components

Für die Darstellung, Wiederverwendbarkeit und Konsistenz über verschiedene Seiten hinweg werden Svelte-Komponenten verwendet. Diese bilden die Bausteine zur Anzeige unterschiedlicher Daten. Zusätzlich können UI Component Libraries wie beispielsweise *flowbite-svelte* eingebunden werden, um standardisierte und ansprechende Benutzeroberflächenelemente wie Buttons, Inputs oder Tabellen zu implementieren. Das Benutzen einer vorgefertigten Bibliothek erleichtert zudem die Entwicklung und trägt dazu bei, eine einheitliche und intuitive Benutzererfahrung sicherzustellen.

Jede Svelte-Komponente besteht aus einem Skript-Teil und einem Design-/Style-Teil. Eine Komponente kann auch weitere Komponenten einbinden. In diesem Fall fungiert die übergeordnete Komponente als Elternteil (Parent), während die eingebundenen Komponenten als Kinder (Children) bezeichnet werden.

Stores

Um während des Erstellungs- und Bearbeitungsprozesses Daten zu speichern, wird ein Svelte Store verwendet. Dies ermöglicht es verschiedenen Komponenten, auf diese Daten zuzugreifen.

Node & Node Modules

Node.js ist eine JavaScript-Laufzeitumgebung, die serverseitige Anwendungen ermöglicht. Sie stellt über den Paketmanager "*npm*"Werkzeuge zur Verwaltung und Bereitstellung von externen Paket-Abhängigkeiten sowie Entwickler-Tools bereit. Neben dem Betrieb eines Webservers ermöglicht *Node.js* auch die serverseitige Verarbeitung von API-Anfragen und die Anbindung an Datenbanken.

2.2.4. Docker

3. Problemanalyse

3.1. Anforderungen und Problemabgrenzung

Die Schnitzeljagden sollen in erster Linie den Studienanfängern (Ersties) dienen, um ihnen auf interaktive Weise den Campus nahe zu bringen und sie mit den wichtigsten Orten und Einrichtungen vertraut zu machen.

Im Rahmen des Projekts wurden folgende Ziele in einer Vier-Felder-Matrix aufgeteilt.

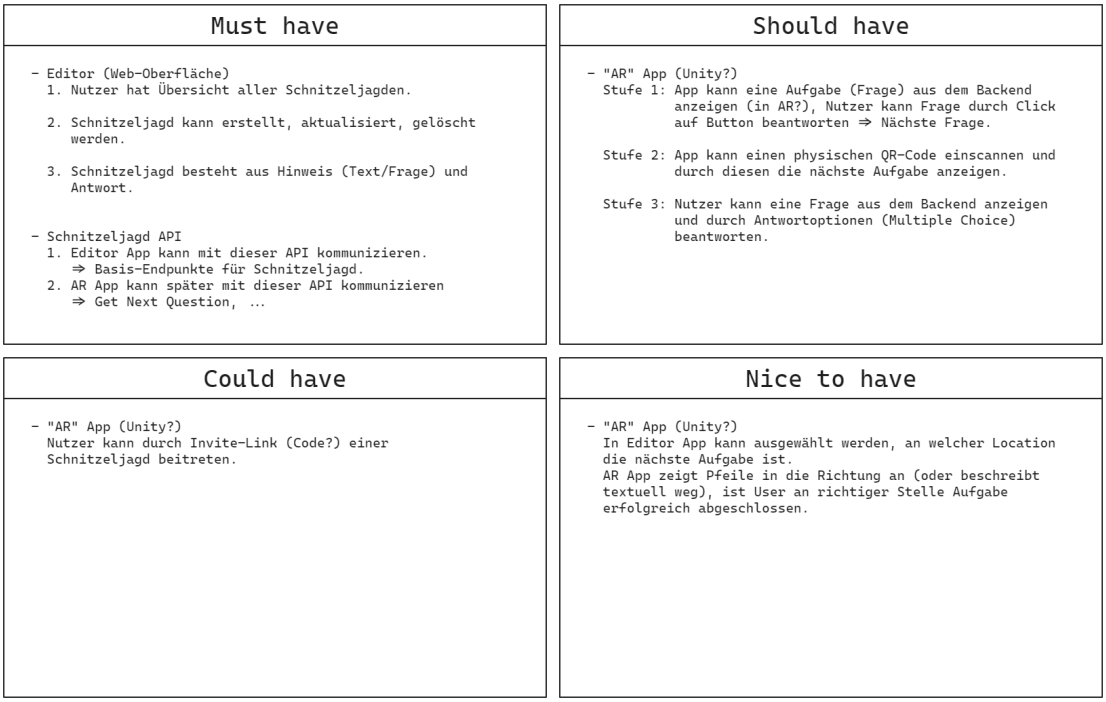

Abbildung 3.1.: Darstellung der Projekt-Ziele aufgeteilt in Vier Felder

Für die erfolgreiche Projektumsetzung sind folgende Eigenschaften zu berücksichtigen.

Flexibilität

Ein wichtiger Aspekt des Projekts ist, dass eine Schnitzeljagd so flexibel wie möglich durchgeführt werden soll. Eine Aufgabenstellung und die dazu gehörende Lösung sollten hierbei entkoppelt und dynamisch erweiterbar sein, ohne einen größeren Aufwand in der Implementierung zu benötigen.

Skalierbarkeit

Das System muss in der Lage sein, mehrere Schnitzeljagden gleichzeitig durchzuführen, ohne dass es zu Problemen kommt. Es sollte möglich sein, die Ressourcen des Systems dynamisch entsprechend der Anzahl der aktuell aktiven Benutzer zu skalieren, ohne dass dabei Performanceeinbußen oder ähnliche Beeinträchtigungen auftreten.

Benutzerfreundlichkeit

Der Anmelde- und Durchführungsprozess sollte unabhängig voneinander klar strukturiert sein, um den Benutzern eine reibungslose und intuitive Erfahrung zu bieten. Während der Durchführung der Schnitzeljagd sollten keine Schwierigkeiten auftreten. Die Schnitzeljagd soll den Benutzern eine einzigartige Erfahrung bieten, ohne durch unnötige oder ablenkende Elemente zu stören. Idealerweise fungiert die Anwendung als Schnittstelle zur Durchführung der Schnitzeljagd, wobei das Lösen der Aufgaben direkte Interaktionen im realen Leben erfordert.

Die digitale Plattform ermöglicht eine einfache Anmeldung und Durchführung der Schnitzeljagd, wodurch der organisatorische Aufwand minimiert und die Zugänglichkeit maximiert wird.

Sicherheit

Das Thema Sicherheit ist im Bezug auf sensible Benutzerdaten wie Passwörter sehr wichtig. Durch die architektonischen Überlegungen die in Kapitel 4 beschrieben werden, wäre es möglich im Anschluss die Benutzerdaten durch kryptographisch sichere Hashfunktionen zu speichern. Um den Projektumfang auf das Wesentliche zu reduzieren, wurde dies im Projekt nicht berücksichtigt.

3.2. Architektonische Überlegungen

Bei der Wahl einer angemessenen Architektur für das System sind viele Aspekte zu berücksichtigen, unter anderem die in Kapitel 3.1 genannten Eigenschaften.

In Kapitel 4 wird daher der service-orientierte Ansatz für den Entwurf der Software allumfassend beschrieben.

4. Software-Entwurf

4.1. Blockansicht

Im folgenden ist die Struktur (*High-Level-Ansicht*) des Software-Systems in Form eines Blackbox-Diagrams dargestellt.

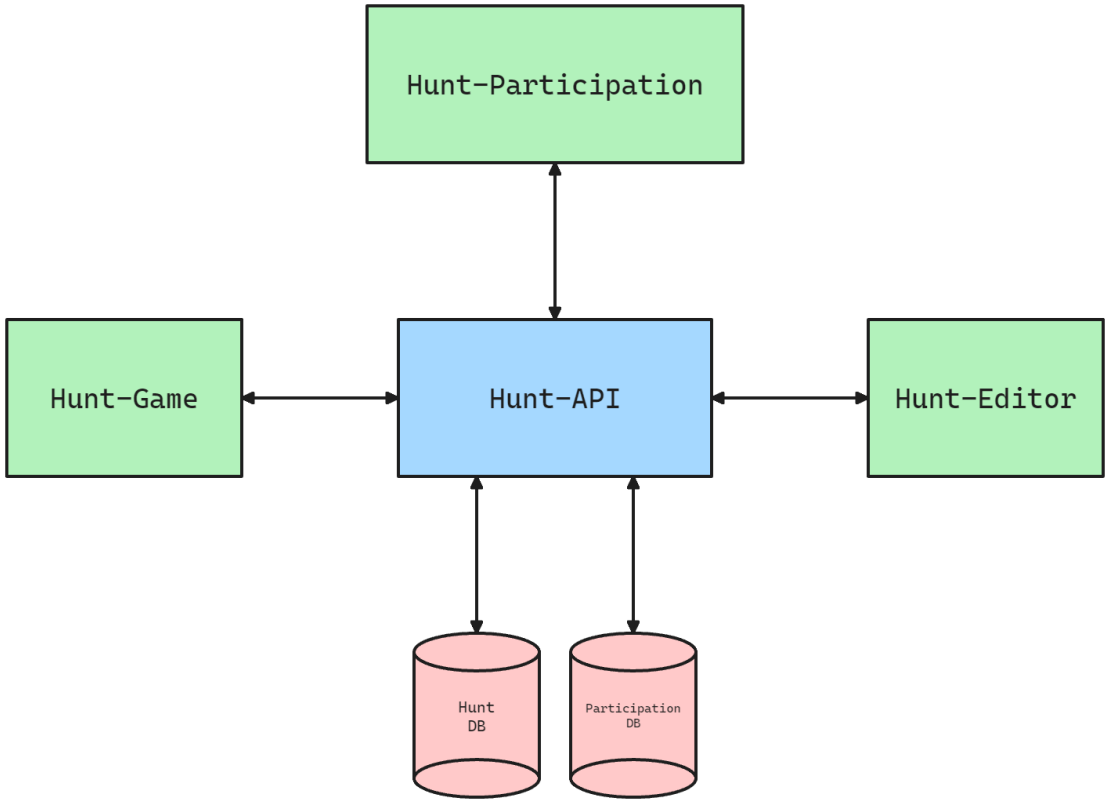

Abbildung 4.1.: Bild Systemkontext als Blackbox-Diagram

Das in Abbildung 4.1 dargestellte Blackbox-Diagramm beschreibt den Zusammenhang der unterschiedlichen Subsysteme zueinander. Die verschiedenen Aufgaben, die das System zu leisten hat, wurden als einzelne Anwendungen vorhergesehen. Diese sind in Abbildung 4.1 grün markiert. Dies ermöglicht eine klare Trennung

der verschiedenen Domänen (Erstellung, Anmeldung, Durchführung). Das Backend steht als zentrale Schnittstelle für die Bereitstellung Systemspezifischer Funktionalitäten zur Verfügung. Dieses sind in Abbildung 4.1 blau markiert. Für die Datenpersistierung besitzt das Backend zwei Datenbank-Verbindungen. Dieses sind in Abbildung 4.1 rot markiert.

4.2. Hunt-Api Backend

4.2.1. Übersicht

Anhand einer zentral zur Verfügung stehenden Schnittstelle *Hunt-Api* können die verschiedenen Anwendungen zur Gesamt-Funktionalität des Systems beitragen. Die Schnittstelle bietet verschiedene Endpunkte für die Verwaltung von Schnitzeljagden, Teilnahmen und die bewältigten Aufgaben der Teilnehmer. Eine Darstellung der Hunt-API Architektur ist in Abbildung 4.2 näher beschrieben.

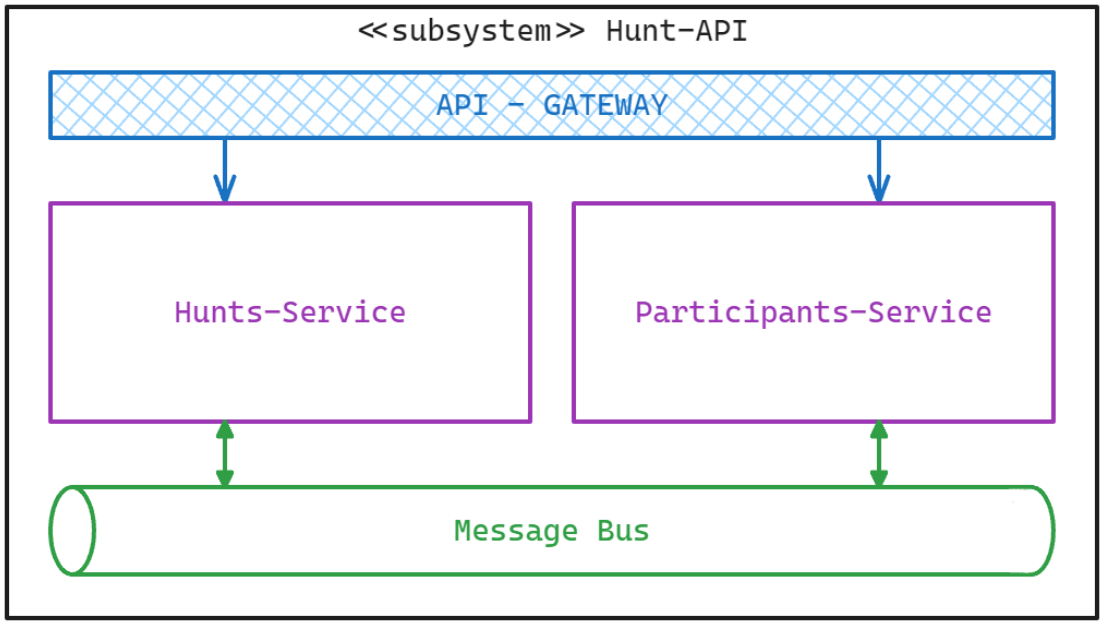

Abbildung 4.2.: Bild Hunt-Api Subsystem

Als Architekturmuster wurde ein domänenorientierter Ansatz gewählt, der in Kapitel 2.1.2 näher erläutert wird. Jede Domäne repräsentiert dabei einen spezifischen Teilaspekt des Gesamtsystems. Die ausgewählten Domänen, *Hunts* und *Participants*, sind in Abbildung 4.2 lila hervorgehoben unr repräsentieren den jeweiligen Dienst als Microservice (vgl. Kapitel 2.1.2).

Für die Kommunikation unterschiedlicher Dienste ist ein Message-Bus vorhergesehen (vgl. Kapitel2.1.3), worüber eventgesteuert Nachrichten ausgestauscht werden. Dieser ist in Abbildung 4.2 grün hervorgehoben.

Um eine einheitliche Schnittstelle bereitzustellen, die anwendungsübergreifend genutzt werden kann, wurde ein Api-Gateway vorhergesehen. Hierrüber werden Anfragen an an den entsprechenden Dienst weitergeleitet. Das API-Gateway ist in Abbildung 4.2 blau gekennzeichnet.

4.2.2. Hunts-Service

Übersicht

Für die Verwaltung der erstellten Schnitzeljagden eines Organisators, ist der Hunts-Service vorhergesehen. Abbildung 4.3 zeigt die Struktur des Dienstes.

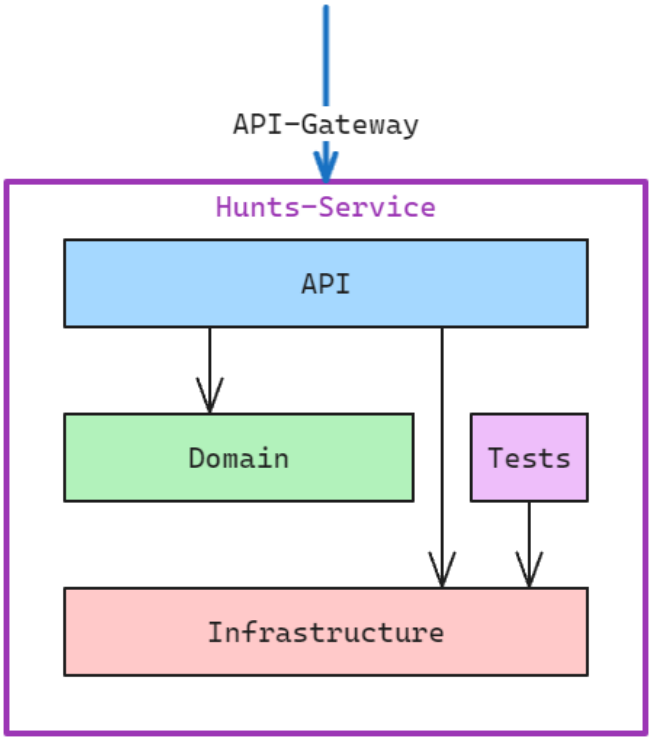

Abbildung 4.3.: Bild Hunts Microservice

Innerhalb des Hunts-Service wurde ein schichtenorientierter Ansatz gewählt, welche Ähnlichkeiten mit der in Kapitel 2.1.2 beschriebenen *Onion-Architecture* teilt. Jede

Schicht entspricht einer horizontalen Teilung der unterschiedlichen Anwendungs-Aspekte. Für die Persistierung der Schnitzeljagden (Hunts) wurde das Repository-Pattern vorhergesehen.

Die grundlegende Funktionalität der Domäne (*Hunts*, *Assignments*, etc.) wird in der *Domain-Schicht* zur Verfügung gestellt, welche in Abbildung 4.3 grün gekennzeichnet wird. Hier sind Modelle und Entities, sowie Datentypen, Enumerations und Repository-Interfaces definiert, die im System durchweg Verwendung finden. Die Domain-Schicht ist abgekapselt von Anwendungs- und Infrastrukturlogik und besitzt daher keine Abhängigkeiten zu externen Modulen.

Eine Implementierung der Repository-Interfaces wird in der *Infrastruktur-Schicht* (Infrastructure) zur Verfügung gestellt, die in Abbildung 4.3 rot hervorgehoben wird. Diese kann zudem unabhängig von der Domänen-Logik isoliert getestet werden, wie in in Abbildung 4.3 lila dargestellt wird.

Die *Anwendungs-Schicht* ist in Abbildung 4.3 blau hervorgehoben und bündelt die Funktionalität der Domain-Schicht und Infrastruktur-Schicht gemeinsam. Über eine einheitliche Schnittstelle (*api*/*Hunt*) können Schnitzeljagden erstellt, bearbeitet, gelöscht und aufgelistet werden.

Schnittstellendefinition

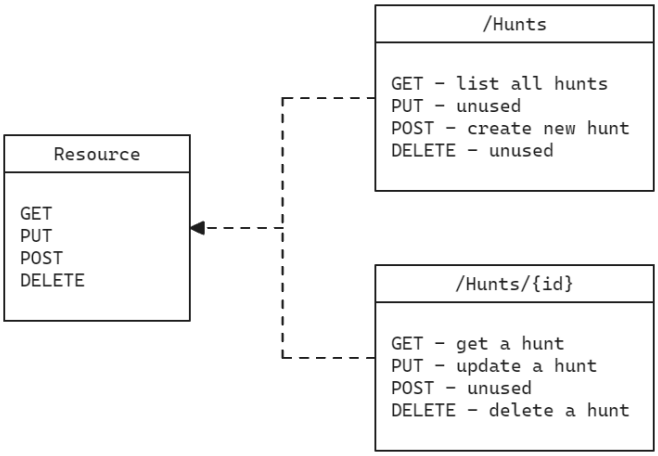

Abbildung 4.4.: Skizze der Hunts Ressourcenansicht

In Abbildung 4.4 sind die unterschiedlichen Operationen auf Schnitzeljagden (*Hunts*) anhand des ressourcen-orientierten Ansatz aus Kapitel 2.1.3 dargestellt. Diese entspricht der tatsächlichen Schnittstelle, die für den Hunts-Service vorhergesehen wurde.

Datenbank-Modell

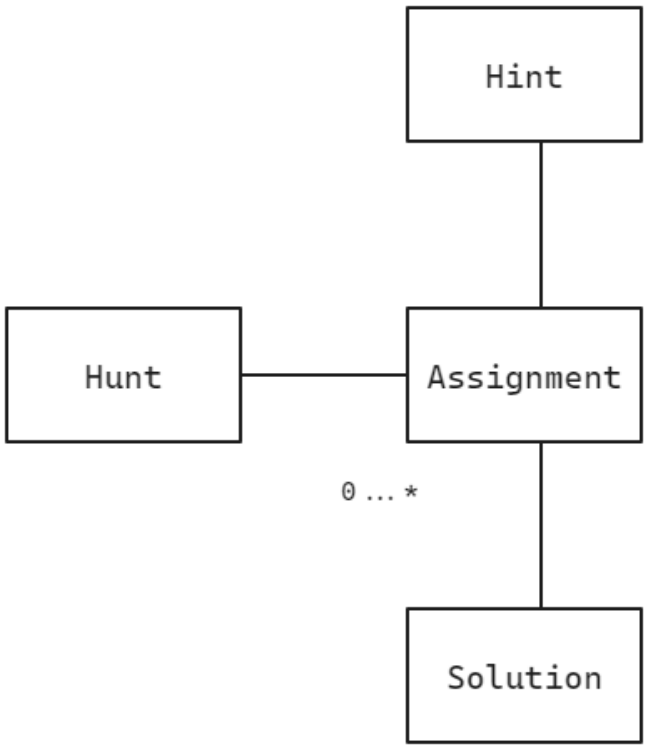

Abbildung 4.5.: Skizze des Hunts ER-Modells

Abbildung 4.5 beschreibt das gewählte Entity-Relationship Diagram für das Datenbank-Modell des Hunts-Service.

- **Hunt**: Die Schnitzeljagd. Enthält einen Titel, eine Beschreibung und eine Liste von 0 bis n Aufgaben (*Assignments*).
- **Assignment**: Eine konkrete Aufgabe einer Schnitzeljagd. Besteht aus einem Hinweis (*Hint*) und eine Lösung (*Solution*).
- **Hint**: Ein Hinweis. Besteht aus einem Hinweis-Typ (als Enumeration) und den Hinweis-Daten als String.
- **Solution**: Eine Lösung zu einem gegebenen Hinweis. Besteht aus einem Lösungs-Typ (als Enumeration) und den Lösungs-Daten als String.

Wieso auf die Verwendung eines abtrakten Datenmodells verzichtet wurde, wird im Anhang A.1.3 näher beschrieben.

4.2.3. Participants-Service

Übersicht

Damit ein Teilnehmer an einer Schnitzeljagd teilnehmen und diese auch durchführen kann, soll ihm die Anmelde- und Spielfunktionalität zur Verfügung gestellt werden. Dies wird über den Participants-Service ermöglicht. Abbildung 4.6 zeigt die Struktur des Dienstes.

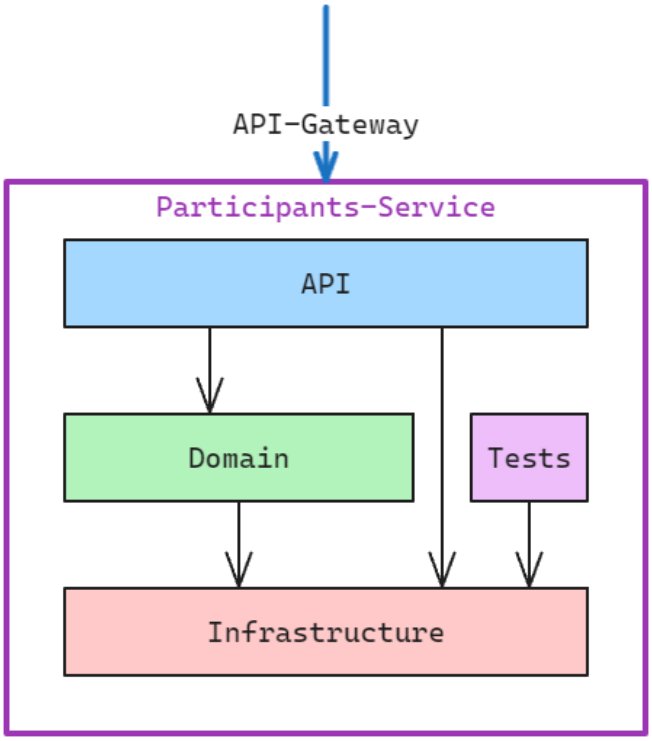

Abbildung 4.6.: Bild Participants Microservice

Der Participants-Service besitzt eine ähnliche Struktur wie der Hunts-Service aus Kapitel 4.2.2 und erfüllt die Aufgabe der Teilnahmen-Verwaltung.

Schnittstellendefinition

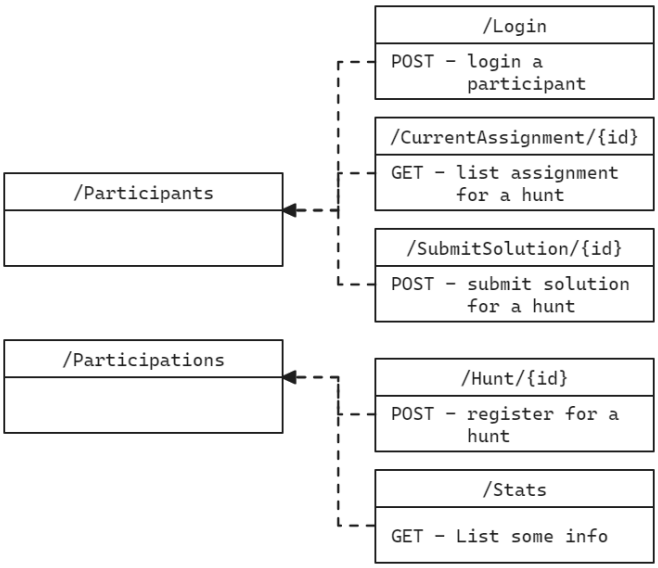

Abbildung 4.7.: Skizze der Participants Schnittstelle

Abbildung 4.7 definiert die vorhergesehene Schnittstelle für den Participants-Service.

Unter den Endpunkten *Participants* werden dem Teilnehmer die Funktionalität zum *Login* angeboten. Hierrüber erhält er zusätzlich Informationen über aktuell angemeldete Schnitzeljagden. Wenn er eine ensprechende Schnitzeljagd starten oder weitermachen will, kann er hierfür den aktuellen Hinweis anfragen mittels *Current-Assignment*. Wenn der Teilnehmer den Hinweis befolgt und die Lösung gefunden hat, kann er diese mittels *Submit-Solution* dem System mitteilen. Das System ermittelt hierdurch, ob die Lösung korrekt ist.

Der genaue Ablauf und das Zusammenspiel der einzelnen Endpunkte wird in Kapitel 4.5.2 erläutert.

Damit sich der Teilnehmer für eine Schnitzeljagd registrieren kann, ist unter dem Endpunkt *Participations* mit *Hunt* und entsprechender *Hunt-Id* eine Registrierungsmöglichkeit vorhergesehen. Zusätzlich können allgemeine Informationen wie beispielsweise Teilnehmeranzahl oder Anzahl aller Teilnahmen an Schnitzeljagden über den Endpunkt *Stats* aufgelistet werden.

Datenbank-Modell

4.3. Hunt-Editor Web-App

4.3.1. Übersicht

Über den Hunt-Editor kann der Organisator Schnitzeljagden verwalten. Hierzu startet er die Anwendung und landet auf die Dashboard-Ansicht. Hier erhält er eine Übersicht von erstellten Schnitzeljagden. Ihm wird die Funktionalität zur Erstellung (*Create*) angeboten. Er kann Titel und Beschreibung sowie die jeweiligen Aufgaben mit Hinweis und Lösung anlegen und speichern. Die Reihenfolge der Reihenfolge der Aufgaben kann bearbeitet werden.

4.3.2. Wireframing

Um den Editor zu entwickeln, der das Anlegen und Verwalten von Schnitzeljagden ermöglicht, wurde zunächst der Ansatz des Wireframings verwendet. Wireframing ist eine wesentliche Phase im Designprozess, die es ermöglicht, die Struktur und Funktionalität einer Anwendung visuell darzustellen, bevor detaillierte Design- und Entwicklungsarbeiten beginnen. In dieser frühen Planungsphase wird ein einfaches, oft schematisches Layout der Benutzeroberfläche erstellt, das die Anordnung der verschiedenen Elemente wie Buttons, Menüs, und interaktiven Komponenten zeigt.

Im Folgenden werden die verschiedenen Wireframes aufgezeigt und erläutert, wieso sie eine solide Grundlage für die weitere Entwicklung des Editors boten, welche Stärken sie in Bezug auf Benutzerfreundlichkeit und Funktionalität aufwiesen, und welche Schwächen oder Herausforderungen während der Umsetzung erkannt wurden, die in späteren Phasen berücksichtigt werden mussten.

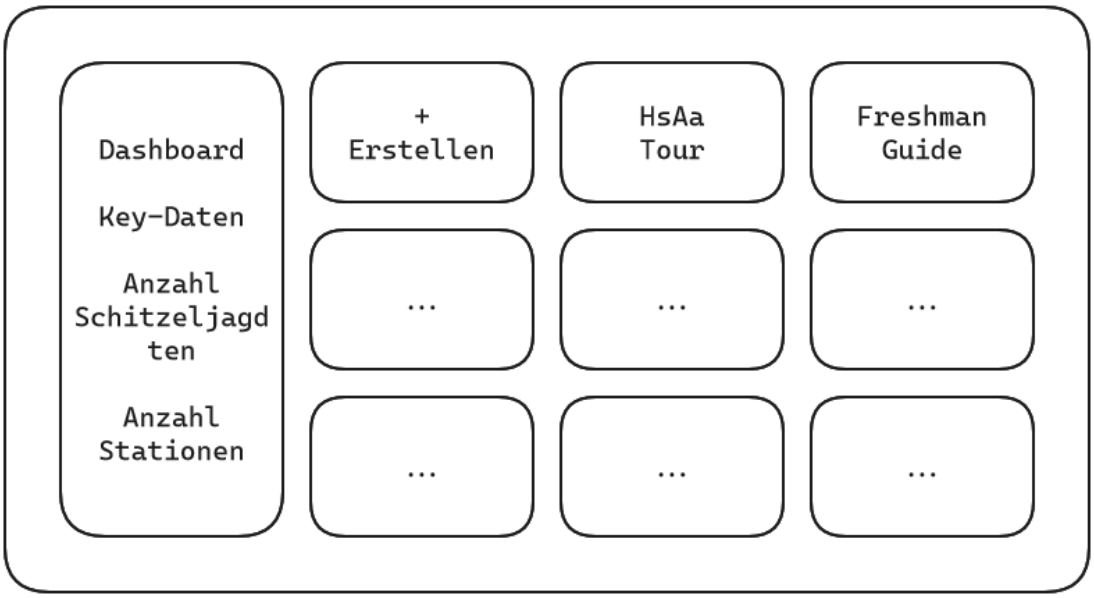

Abbildung 4.8.: Skizze Dashboard des Hunt-Editors

Dies war ein Wireframe, was relativ am Anfang der Entwicklungsphase entstanden ist. Hier wurde auf ein Grid-Layout gesetzt, um die Schnitzeljagden anzuzeigen. An der Seite befindet sich eine Sidebar, in der verschiedene Key-Daten abgelesen werden können. Da es aber durch Architekturänderungen keine Stationen mehr gibt, fällt diese Information weg. Das Grid-Layout sollte aber bestehen bleiben.

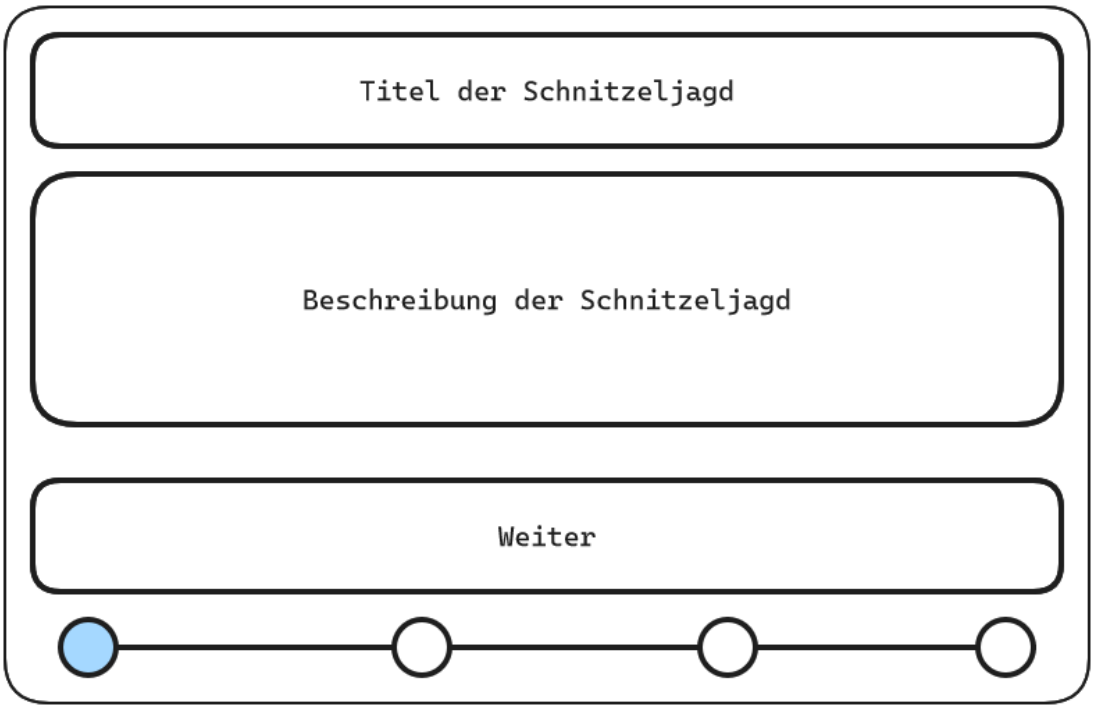

Abbildung 4.9.: Skizze für Eingabe von Basisdaten im Hunt-Editor

Hier wird nun der Prozess des Erstellen einer Schnitzeljagd beschrieben, gefallen hat uns hier das Verwenden einer Progressbar, um den aktuellen Fortschritt beim Erstellen anzuzeigen. Auch die Eingabe von Titel und Beschreibung sollte so später übernommen werden.

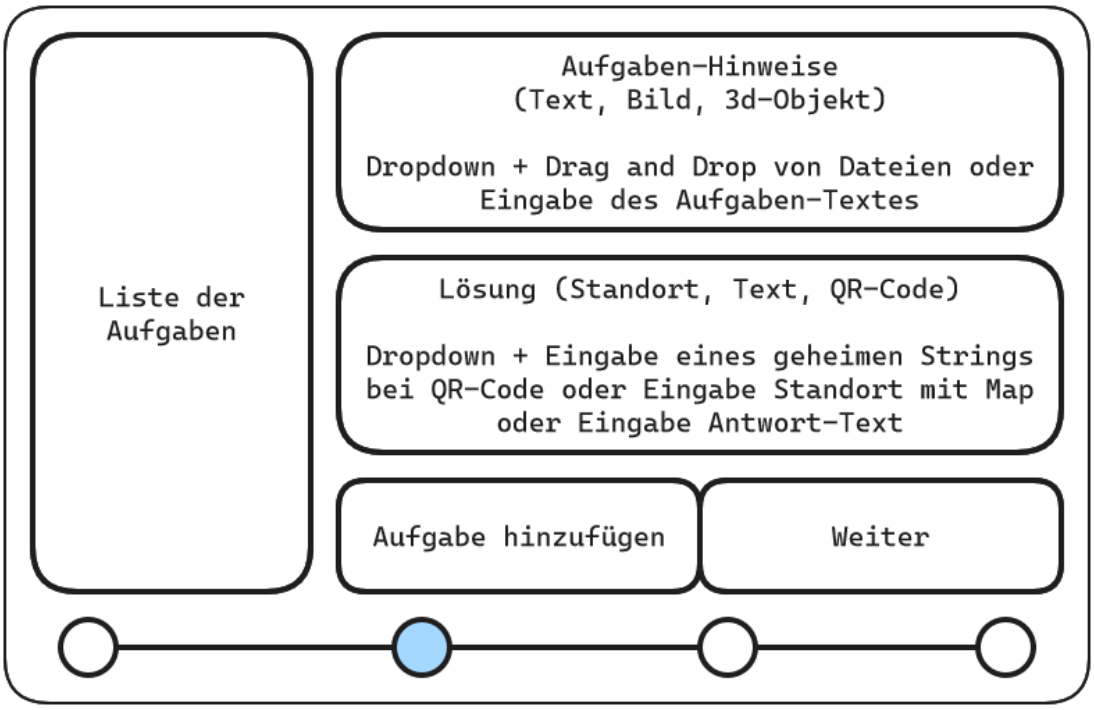

Abbildung 4.10.: Skizze für Anlegen von Aufgaben im Hunt-Editor

Nun zum schwersten Teil, das Anlegen und Bearbeiten von Aufgaben in einer Schnitzeljagd. Hier war zunächst der Ansatz, auf der linken Seite eine Tabelle der Aufgaben zu führen. Durch Klick auf die entsprechende Zeile werden dann auf der rechten Seite die Aufgabe und Lösung angezeigt. Hier ist auch noch der Aufgabentyp "3d-Objekt"vorhanden, welcher am Schluss nicht übernommen wurde.

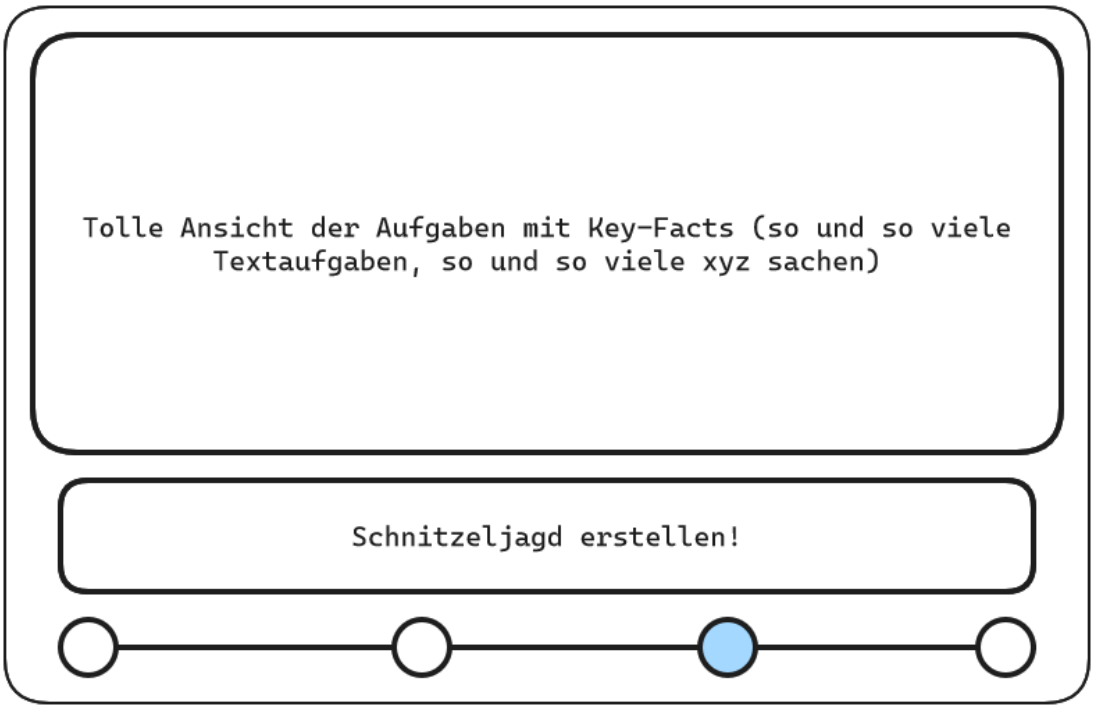

Abbildung 4.11.: Skizze zur Übersicht einer Schnitzeljagd im Hunt-Editor

Dieses Wireframe zeigt zum Schluss nochmal eine Übersicht aller wichtigen Infos der Schnitzeljagd an. Dieses Konzept hat uns gut gefallen.

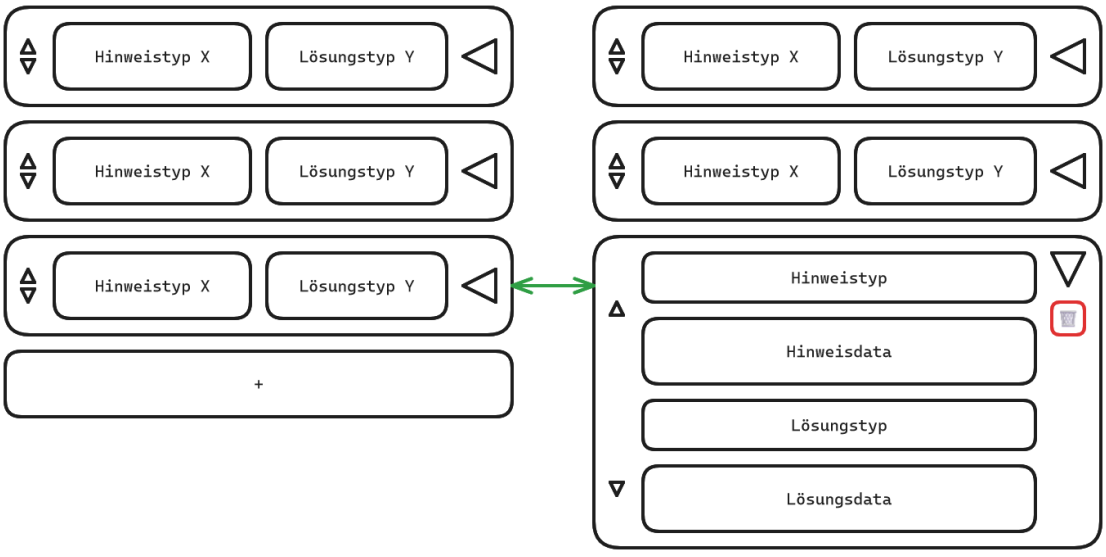

Abbildung 4.12.: Skizze zur Auflistung von Aufgaben im Hunt-Editor

Da die seitliche Liste aus Abbildung 4.10 zum Anzeigen der Aufgaben für mobile Geräte ungeeignet ist, haben wir uns für den Ansatz aus Abbildung 4.12 entschieden. Hier werden die Aufgaben nacheinander angezeigt und deren Reihenfolgen kann über das Bedienen der Pfeiltasten angepasst werden. Um die Aufgaben zu bearbeiten, können diese über den Button auf der rechten Seite aufgeklappt und wieder zugeklappt werden. Im aufgeklappten Zustand befindet sich jeweils ein Dropdown für Aufgabe und Lösung als auch Eingabefelder / FileUpload oder eine Map Integration für Standort als Lösung.

4.4. Participant Web-App

4.4.1. Übersicht

Über die Participant Web-App kann sich ein Teilnehmer an einer Schnitzeljagd anmelden. Hierzu startet er die Anwendung und landet auf einer Begrüßungsseite. Hier kann er sehen, wie viele Teilnehmer und wie viele Teilnahmen es insgesamt gibt. Um nun einer Schnitzeljagd beizutreten, kann der Teilnehmer einem Organisator eine Mail schreiben. Erhält er daraufhin einen Einladungslink für eine Schnitzeljagd, so muss er dort nur einen Nutzernamen und ein Passwort auswählen. Mit diesem Nutzernamen und Passwort meldet er sich im in Kapitel 4.5 beschriebenen Hunt-Game an.

4.4.2. Wireframing

4.5. Hunt-Game Web-App

4.5.1. Übersicht

Beim Start der Anwendung sieht der Teilnehmer eine Sammlung an Bildern der Hochschule Aalen, darunter befindet sich eine kleine Timeline, die den Entwicklungsprozess der Projektarbeit beschreibt. Um an einer Schnitzeljagd teilnehmen zu können, muss sich der Teilnehmer mit dem gewählten Nutzernamen und Passwort aus der Participant Web-App anmelden. Daraufhin wird ihm eine Übersicht von allen Schnitzeljagden angezeigt, für welche er sich registriert hat. Hier wird in *Ongoing, Complete* und *Expired* unterschieden, dies wird durch den ParticipationStatus ermittelt. Wählt er eine valide Schnitzeljagd aus, so startet der Spielablauf.

4.5.2. Spielablauf

Abbildung 4.13 stellt den Spielablauf als UML-Programmablaufplan dar.

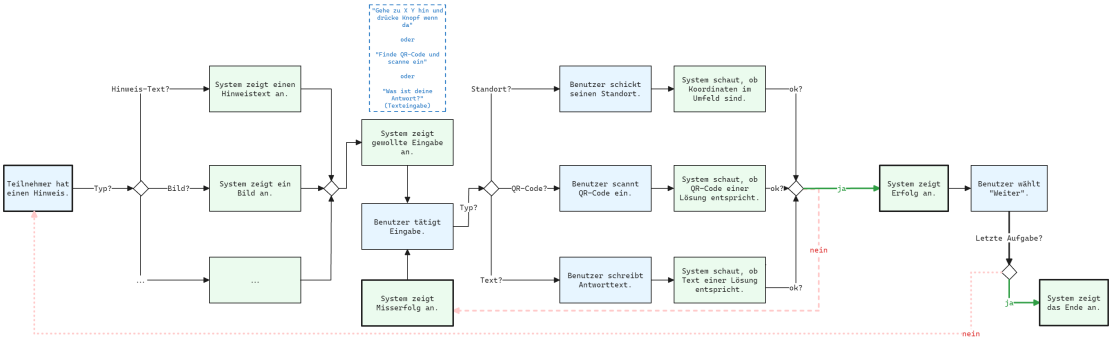

Abbildung 4.13.: Skizze Spielablauf als UML-Programmablaufplan

Nachdem der Teilnehmer eine valide Schnitzeljagd ausgewählt hat, beginnt der eigentliche Spielablauf. Zu Beginn wird dem Teilnehmer der aktuelle Hinweis präsentiert, der ihn zur nächsten Station oder Aufgabe führen soll. Die Hinweise sind in zwei Typen unterteilt: Text und Bild. Bei einem Text-Hinweis wird dem Teilnehmer ein beschreibender Text angezeigt, der Informationen oder Anweisungen zur nächsten Station enthält. Bei einem Bild-Hinweis wird stattdessen ein Bild angezeigt, das visuelle Hinweise oder Details enthält, die zur Lösung der Aufgabe beitragen.

Unter dem Hinweis befindet sich ein Button, über den der Teilnehmer seine Lösung einreichen kann. Hierbei gibt es drei verschiedene Lösungstypen, die je nach Aufgabe variieren:

- **Textlösung**: Der Teilnehmer gibt seine Lösung in ein Textfeld ein. Dies kann beispielsweise ein gesuchtes Wort, ein Satz oder eine Zahlenkombination sein.
- **QR-Code**: Bei Aufgaben, die einen QR-Code erfordern, wird die Kamera des Geräts aktiviert, um den QR-Code zu scannen. Dieser QR-Code kann an verschiedenen Orten versteckt sein und enthält die notwendigen Informationen oder Anweisungen, um zum nächsten Hinweis zu gelangen.
- **Location**: Wenn die Lösung in Form eines geografischen Standorts vorliegen soll, wird der Teilnehmer aufgefordert, den Standortzugriff zu erlauben. Der Browser ermittelt dann die aktuellen GPS-Koordinaten des Geräts und überprüft, ob diese mit der erwarteten Lösung übereinstimmen.

Der Spielablauf wiederholt sich, bis alle Aufgaben gelöst sind. Bei den Lösungstypen Text und Location erhält der Teilnehmer zusätzlich Hinweise, falls seine Lösung

nicht korrekt ist. Diese Hinweise geben an, dass die Lösung fast richtig ist, aber noch Anpassungen erfordert. Dies dient dazu, die Teilnehmer zu ermutigen und ihnen eine Chance zu geben, ihre Antwort zu verbessern.

5. Implementierung

5.1. Entwickeln der Hunt-Editor Web-App

5.1.1. Organisation

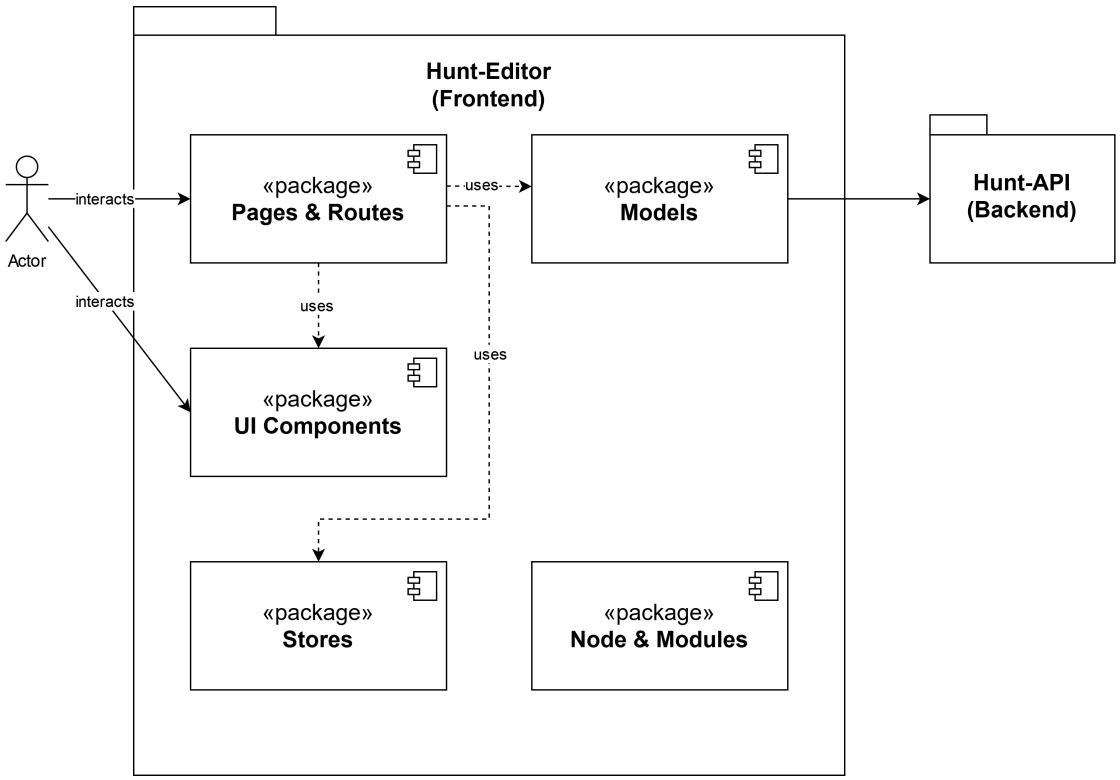

Abbildung 5.1.: UML-Diagramm Whitebox Frontend Hunt-Editor

Abbildung 5.1 skizziert die grundlegenden Elemente, aus welchen die Anwendung besteht. Für das Zwischenspeichern der Schnitzeljagd-Attribute (Titel, Beschreibung, Aufgaben) werden Svelte Stores verwendet (vgl. Kapitel 2.2.3).

Im Folgenden ist die Struktur des Routings als Baumdiagramm dargestellt. Wie eine Svelte-Anwendung aufgebaut ist, wird in Kapitel 2.2.3 näher beschrieben.

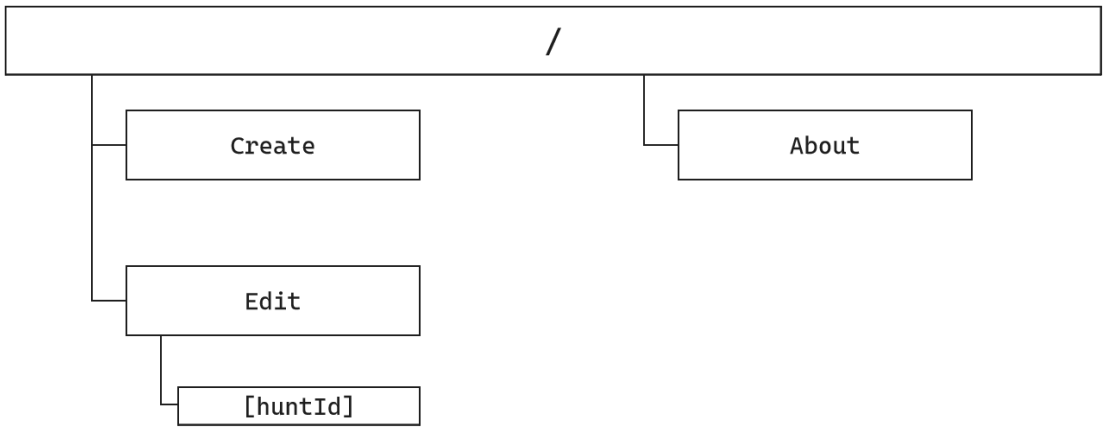

Abbildung 5.2.: Skizze Frontend Hunt-Editor Routes

Für das Anzeigen und Entfernen von Schnitzeljagden ist die Home Route / vorgesehen. Auf der Route /*about* werden Informationen zur Anwendung und zum Projekt bereitgestellt.

Der Workflow zum Erstellen einer Schnitzeljagd ist in der Route /*create* implementiert.

Zum Editieren einer Schnitzeljagd werden die Komponenten aus der /*create*-Route wiederverwendet, hier wird jedoch über die /*edit*/*[huntId]*-Route gearbeitet, wobei *[huntId]* ein Platzhalter für die *Id* der jeweiligen Schnitzeljagd ist. Dadurch können dynamisch über die URL Informationen zur Schnitzeljagd abgerufen werden. Diese Informationen (wie bspw. Titel und Beschreibung) werden dann in die Eingabefelder eingetragen.

5.1.2. UI-Design

Um den Editor zu erstellen, wurde zunächst versucht, mit DaisyUI zu arbeiten. Da dies nicht das gewünschte Ergebnis geliefert hat, wurde zu Flowbite gewechselt. Kapitel A.2.1 begründet diese Entscheidung.

5.2. Entwickeln einer AR-Anwendung mit Unity und AR-Foundation

Im Folgenden ist der Entwicklungsprozess für das Implementieren eines AR-Image-Trackers mithilfe des AR-Foundation Frameworks von Unity beschrieben. Im Anschluss wird erläutert, weshalb dieser Ansatz für das Projekt obsolet wurde.

Für die Entwicklung mit Unity wird die Installation des Unity-Hubs und einer unterstützten Version des Unity-Editor vorausgesetzt. Die im Projekt entwickelte Anwendung wurde mit der Editor-Version *2022.3.20f1* erstellt.

5.2.1. Anlegen einer Reference-Image-Library

Nachdem ein AR-Unity-Projekt nach [12] angelegt wurde und das nötige Framework eingebunden wurde, stehen verschiedene Funktionen des AR-Foundation-Frameworks im Editor zur Verfügung.

Zunächst soll eine *Reference-Image-Library* erstellt werden. Abbildung 5.3 markiert die grundlegenden Schritte hierfür.

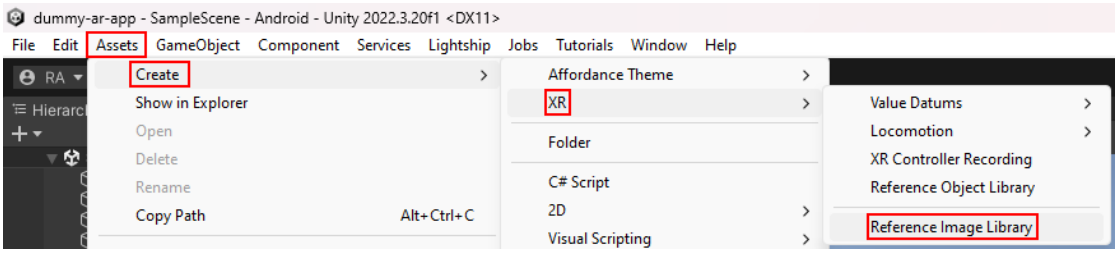

Abbildung 5.3.: Bildschirmabschnitt für das Erstellen einer Reference-Image-Library

In der geöffneten Szene in Abbildung 5.4 sollte nun ein leeres Reference-Image-Library Objekt existieren.

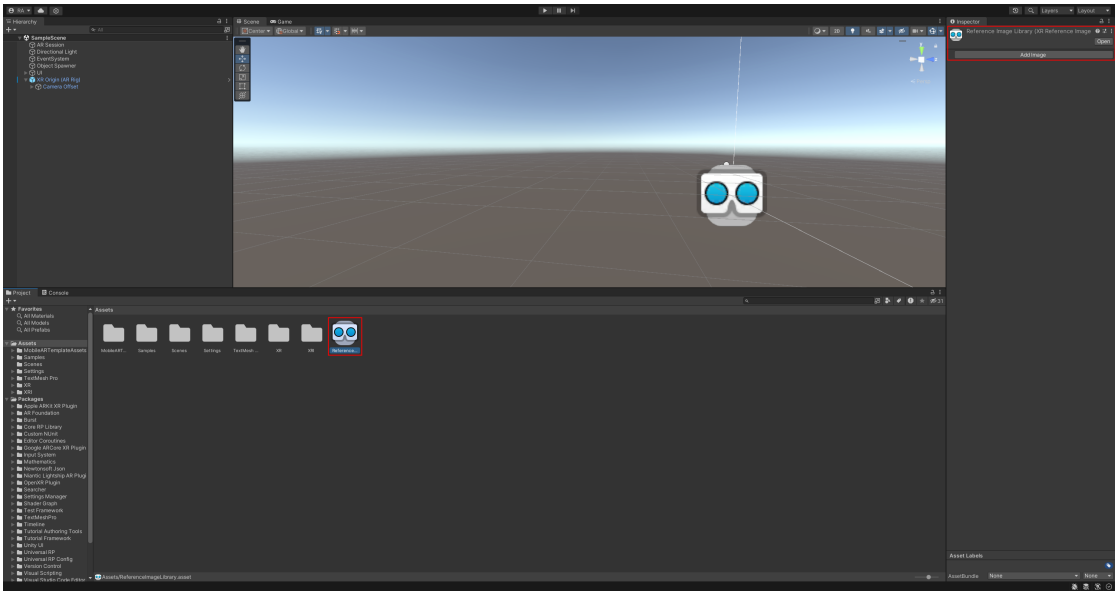

Abbildung 5.4.: Bildschirmabschnitt Reference-Image-Library Objekt im Editor

Die Reference-Image-Library wird benötigt, um vor der Laufzeit bereits Referenzbilder (*Reference-Images*) oder Objekte zu sammeln, die von einem AR-Tracker wie etwa die Komponente *XRImageTrackingSubsystem* erkannt werden können.

5.2.2. Anlegen eines AR-Tracked-Image-Managers

Die *AR Tracked Image Manager* Komponente erstellt *Game-Objects* für jedes gefundene Bild innerhalb des Viewports des Benutzers. Bevor ein Bild gefunden werden kann, muss die Komponente Referenz-Bilder kennen, die in der *Reference-Image-Library* gespeichert sind.

Das Einbinden erfolgt, indem ein Skript an die XR-Origin Komponente angebunden wird und die dementsprechende Reference-Image-Library per Drag-and-Drop eingebunden wird. Dies aktiviert das Tracken von Bildern im AR-Raum.

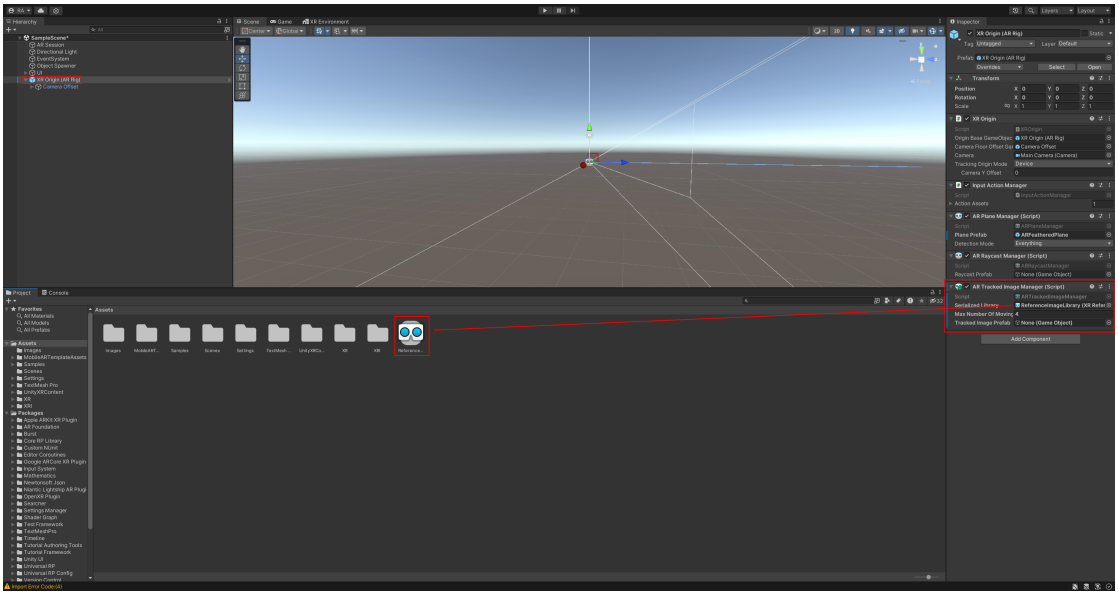

Abbildung 5.5.: Bildschirmabschnitt für das Erstellen eines AR-Tracked-Image-Managers

5.2.3. QR-Code-Daten aus einem Referenz-Bild lesen

Das erkennen eines QR-Codes erfordert, dass dieser in der Reference-Image-Library bereits angelegt wurde. Nachdem dies erfolgt ist, kann der QR-Code als Textur gelesen und über einen QR-Code-Konverter wie *ZXing* als Text konvertiert werden. Abbildung 5.6 zeigt das erfolgreiche Umkonvertieren eines QR-Codes im AR-Raum.

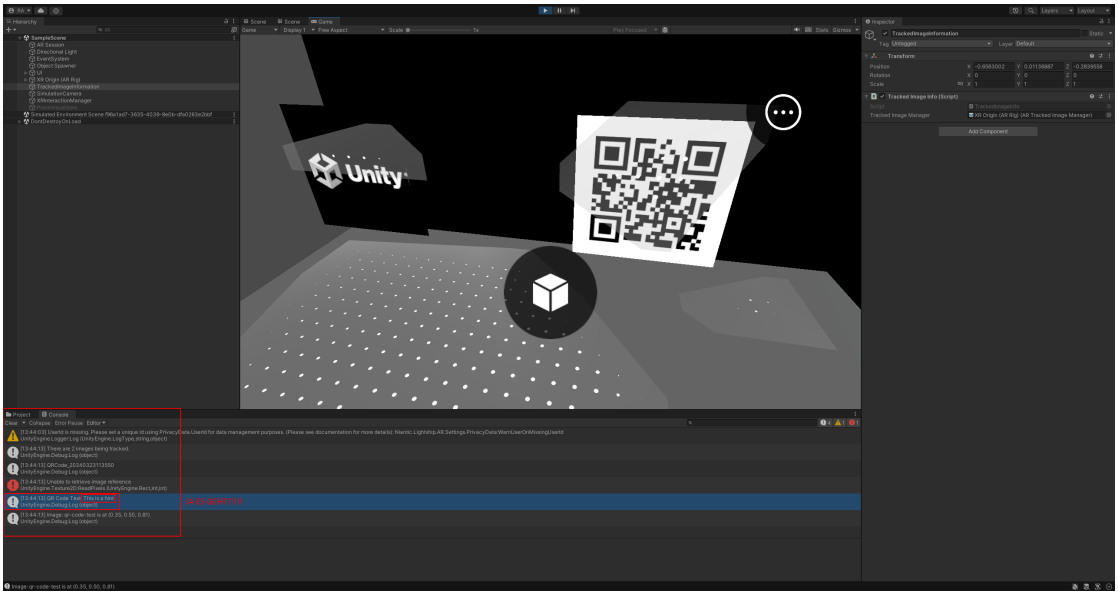

Abbildung 5.6.: Bildschirmabschnitt für das Lesen eines QR-Codes

5.2.4. Erfahrung mit Unity und AR-Foundation

Leider kann über den in Kapitel 5.2 beschriebenen Ansatz kein gewünschtes Ergebnis erzielt werden.

Ein Hinweistext oder -bild an der Position des QR-Codes zu platzieren, könnte theoretisch funktionieren, wie in Abbildung 5.6 bereits dargestellt wird. Allerdings stellt die mangelnde Flexibilität der Reference-Image-Library ein erhebliches Hindernis für eine funktionierende Implementierung dar.

Um dieses Problem zu umgehen, müsste die Anwendung beim Start alle im System vorhandenen QR-Codes und ihre entsprechenden Lösungen über das Backend abrufen. Dies würde jedoch bei einer großen Anzahl von QR-Codes zu einer suboptimalen Benutzererfahrung führen, da die Anwendung erheblich verzögert wäre. Zusätzlich verursacht das Laden mehrerer QR-Codes einen hohen Speicheraufwand.

Außerdem werden die Lösungen vor dem Erhalt eines ersten Hinweises auf dem Gerät des Teilnehmers geladen. Dies könnte zur Folge haben, dass manche Lösungen aus den geladenen Bildern extrahiert werden können, wodurch das Konzept der Schnitzeljagd umgangen wird.

Diese Herausforderungen zeigen deutlich, dass Unity und AR Foundation in diesem Kontext nicht die optimale Lösung darstellen. Die genannten Einschränkungen machen es schwierig, eine flexible, effiziente und sichere Anwendung zu entwickeln, die

den Anforderungen einer Schnitzeljagd entspricht. Daher ist es ratsam, alternative Technologien oder Ansätze zu berücksichtigen, die eine bessere Anpassungsfähigkeit und Sicherheit bieten.

Die Entscheidung, Unity nicht für das Schnitzeljagd-Spiel zu verwenden, wird zudem in folgenden Aspekten begründet.

Entwicklungsumgebung und Tooling

Die Entwicklungsumgebung und der Entwicklungsprozess, die für die Erstellung einer Anwendung in Unity erforderlich sind, weisen erhebliche Nachteile auf. Das Tooling von Unity ist oft unhandlich und ineffizient. Häufig auftretende Probleme müssen durch Neustarts des Unity-Hubs oder des Unity-Editors gelöst werden. Diese häufigen Unterbrechungen führen zu einem erheblichen Zeitverlust, der den Entwicklungsprozess stark behindert.

Ein weiteres Problem ist die resultierende Größe der Anwendung nach dem Build. Die Unity-Engine erzeugt sehr große Dateien, selbst wenn nur grundlegende Funktionen verwendet werden. Eine Optimierung auf das Nötigste ist kaum möglich, was die Anwendung unnötig aufbläht und somit unpraktisch für mobile Geräte macht.

Anpassung auf Projektanforderungen

AR Foundation, die AR-Entwicklungsplattform von Unity, bietet für unsere Schnitzeljagd-Anwendung keine entscheidenden Vorteile. Unsere Anwendung konzentriert sich hauptsächlich auf die Darstellung von Texten und Bildern sowie das Scannen von QR-Codes, was keine umfassende AR-Funktionalität erfordert. Eine einfache und ressourcenschonende Lösung wäre hier weitaus effizienter.

Lizensierung und Business

Das Unternehmen hinter Unity, Unity Technologies, hat in den letzten Monaten erheblich an Reputation verloren. Grund dafür sind umstrittene Änderungen in den Lizenzbedingungen und Preismodellen. Diese Änderungen haben bei Entwicklern für Unmut gesorgt und das Vertrauen in das Unternehmen erschüttert. [13]

Die negativen Schlagzeilen und die Unsicherheit über zukünftige Geschäftsstrategien machen Unity zu einer risikobehafteten Wahl für langfristige Projekte.

5.2.5. Alternativen

Vuforia

WebAr

Eine vielversprechende Alternative zur nativen AR-Entwicklung mit Unity ist WebAR. WebAR ermöglicht AR-Erlebnisse direkt im Webbrowser ohne die Notwendigkeit einer separaten App-Installation. Dies bietet mehrere Vorteile:

- **Zugänglichkeit**: WebAR ist auf jedem modernen Webbrowser verfügbar, was die Zugänglichkeit für Benutzer erleichtert und die Notwendigkeit, eine spezielle App herunterzuladen, beseitigt. Dies kann die Benutzerfreundlichkeit und die Reichweite der Anwendung erhöhen.
- **Einfachere Entwicklung**: WebAR nutzt Web-Technologien wie JavaScript und WebGL, die für viele Entwickler vertrauter und zugänglicher sind als die speziellen Tools und Skripte von Unity. Die Entwicklung kann daher schneller und einfacher sein, besonders wenn bereits Erfahrung mit Web-Technologien besteht.
- **Kosteneffizienz**: Da WebAR keine zusätzlichen Lizenzgebühren oder Laufzeitgebühren erfordert, ist es eine kostengünstigere Lösung.

Schlussfolgerung

Falls die Schnitzeljagd Anwendung in Zukunft AR-Funktionen beinhalten soll, stellt WebAR die effizienteste und kostengünstigste Lösung dar. Es bietet eine einfache und zugängliche Möglichkeit, grundlegende AR-Funktionen bereitzustellen, ohne die Notwendigkeit für eine komplexe App-Installation oder hohe Entwicklungs- und Betriebskosten. Vuforia und Unity bieten erweiterte Funktionen, die für diese Anwendung nicht erforderlich sind und könnten daher überdimensioniert sein.

5.3. Algorithmen zur Hinweisbestimmung für Nutzerlösungen

Um dem Nutzer Hinweise zu geben, dass eine von ihm eingesandte Lösung nahezu korrekt ist, wurden zwei Algorithmen implementiert. Diese Algorithmen helfen dabei, den Grad der Übereinstimmung zwischen der vom Nutzer angegebenen Lösung und der tatsächlich gesuchten Antwort zu bestimmen.

5.3.1. Haversine-Algorithmus zur Bestimmung geografischer Distanzen

Einer der Algorithmen kommt zum Einsatz, wenn die Lösung einen geografischen Ort umfasst, der durch Breiten- und Längengrade (Latitude und Longitude) definiert ist. Hier wird die Haversine-Formel verwendet, um die Entfernung zwischen den beiden Punkten zu berechnen.

Die Haversine-Formel ist besonders nützlich, um die kürzeste Entfernung über die Erdoberfläche zu bestimmen, da sie die Krümmung der Erde berücksichtigt. Diese Berechnung ist essenziell, um zu bestimmen, wie nahe ein Nutzer mit seiner Antwort an der korrekten Position liegt.

Der folgende C#-Code zeigt die Implementierung des Haversine-Algorithmus:

```
1  public static double Haversine(double lat1, double lon1,
       double lat2, double lon2)
2  {
3      // Convert degrees to radians
4      double dLat = ToRadians(lat2 - lat1);
5      double dLon = ToRadians(lon2 - lon1);
6
7      lat1 = ToRadians(lat1);
8      lat2 = ToRadians(lat2);
9
10     // Haversine formula
11     double a = Math.Sin(dLat / 2) * Math.Sin(dLat / 2) +
12                Math.Sin(dLon / 2) * Math.Sin(dLon / 2) *
       Math.Cos(lat1) * Math.Cos(lat2);
13     double c = 2 * Math.Atan2(Math.Sqrt(a), Math.Sqrt(1 -
       a));
14
15     // Distance in meters
16     return RADIUS_OF_EARTH_METERS * c;
17 }
18
19 private static double ToRadians(double angleInDegrees)
20 {
21     return angleInDegrees * Math.PI / 180.0;
22 }
```

Quelltext 5.1: Code Ausschnitt zum Haversine Algorithmus in C#

In diesem Algorithmus wird zunächst die Differenz zwischen den Längen- und Breitengraden der beiden Punkte berechnet und in Radianten umgewandelt. Anschließend wird die Haversine-Formel angewendet, um den Anteil des großen Kreises

(die kürzeste Strecke zwischen zwei Punkten auf der Oberfläche einer Kugel) zu berechnen, den der Abstand zwischen den Punkten darstellt. Dieser Anteil wird dann mit dem Radius der Erde multipliziert, um die tatsächliche Entfernung in Metern zu erhalten.

Durch diese Berechnung kann das System bestimmen, wie weit der vom Nutzer angegebene Ort von der korrekten Position entfernt ist, und so ein Feedback geben, das darauf hinweist, dass die Lösung nahe dran ist, wenn der Abstand gering ist.

Anwendungsbeispiel des Haversine-Algorithmus

Im folgenden wird der Harvesine-Algorithmus zum Bestimmen der Distanz des Benutzers zur Lösung verwendet.

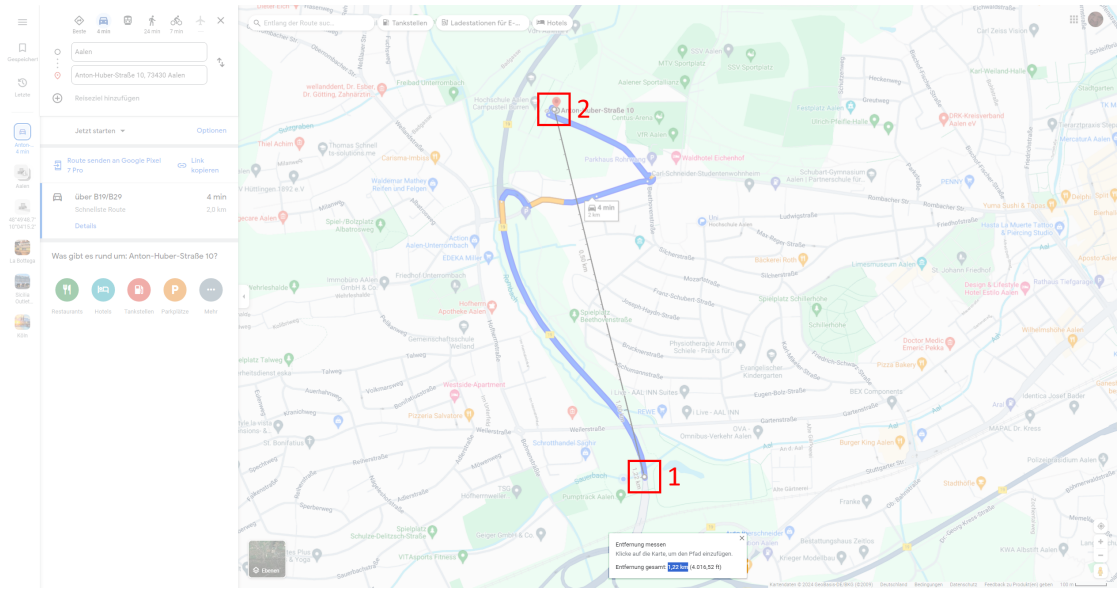

Abbildung 5.7.: Bildschirmabschnitt Abstand Teilnehmer und Lösung

Abbildung 5.7 stellt den Standort des Teilnehmers (Abbildung 5.7 - 1) zu einem gesuchten Standort (Abbildung 5.7 - 2) dar. Die tatsächliche Distanz entspricht in diesem Beispiel *1213.14* Meter.

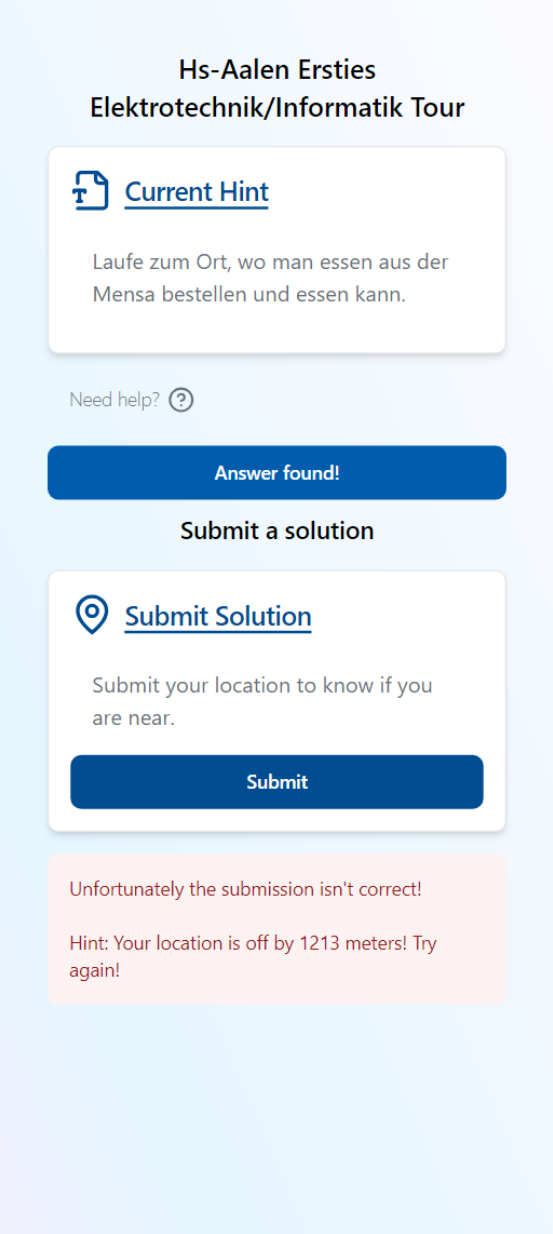

Abbildung 5.8.: Bildschirmabschnitt Abstands-Hinweis an Teilnehmer

Abbildung 5.8 zeigt den Zustand der Benutzeroberfläche an, nachdem ein Teilnehmer seine Position als Lösung an das System schickt. Es wurde korrekterweise erkannt, dass der Teilnehmer *1213.14* Meter von der gewollten Lösung entfernt ist.

5.3.2. Levenshtein-Algorithmus zur Bestimmung textueller Unterschiede

Um den Nutzern Feedback darüber zu geben, wie nah ihre eingesandte Antwort an der korrekten Lösung liegt, wird neben der Haversine-Formel auch ein Algorithmus zur Berechnung der Levenshtein-Distanz eingesetzt. Diese Methode wird verwendet, wenn die Lösung aus Textdaten besteht, wie etwa bei der Eingabe von Textantworten auf Rätsel oder Aufgabenstellungen.

Die Levenshtein-Distanz misst den Unterschied zwischen zwei Zeichenketten in Bezug auf die minimale Anzahl von Einfüge-, Lösch- und Ersetzungsoperationen, die erforderlich sind, um die eine Zeichenkette in die andere zu transformieren. Dies ist besonders nützlich, um Tippfehler oder leichte Abweichungen in den Antworten der Nutzer zu identifizieren und entsprechende Rückmeldungen zu geben, die darauf hinweisen, dass die Antwort fast richtig ist, wenn nur geringe Abweichungen vorliegen.

Der folgende C#-Code zeigt die Implementierung der Levenshtein-Distanz:

```
1  public static int LevenshteinDistance(string givenText,
       string expectedText)
2   {
3       int n = givenText.Length;
4       int m = expectedText.Length;
5       int[,] d = new int[n + 1, m + 1];
6
7       // Initialize the distance matrix
8       for (int i = 0; i <= n; i++)
9       {
10          d[i, 0] = i;
11      }
12      for (int j = 0; j <= m; j++)
13      {
14          d[0, j] = j;
15      }
16
17      // Compute the distances
18      for (int i = 1; i <= n; i++)
19      {
20          for (int j = 1; j <= m; j++)
21          {
22              int cost = (givenText[i - 1] ==
       expectedText[j - 1]) ? 0 : 1;
23
24              d[i, j] = Math.Min(
25                  Math.Min(d[i - 1, j] + 1, d[i, j - 1] +
       1),
26                  d[i - 1, j - 1] + cost);
27          }
28      }
29
30      return d[n, m];
31   }
```

Quelltext 5.2: Code Ausschnitt zum Levenshtein Distanz Algorithmus in C#

Die Matrix `d` wird so initialisiert, dass die ersten Zeilen und Spalten mit aufsteigenden Werten gefüllt sind. Diese Werte entsprechen den Kosten für das Einfügen oder Löschen von Zeichen, um die Länge der anderen Zeichenkette zu erreichen. Die Hauptlogik des Algorithmus durchläuft die Matrix und berechnet die minimalen Kosten für die Transformation von Substrings. Dies geschieht durch den Vergleich der aktuellen Zeichen der beiden Zeichenketten und der Anwendung der entspre-

chenden Operationen (Einfügen, Löschen, Ersetzen). Der letzte Wert in der Matrix `d[n, m]` gibt die minimale Anzahl der Transformationen an, die erforderlich sind, um die beiden Zeichenketten anzugleichen.

Die Levenshtein-Distanz ermöglicht es dem System, präzise Rückmeldungen zu geben, wie nahe der Nutzer an der richtigen Antwort ist, und unterstützt so ein positives Benutzererlebnis durch konstruktives Feedback.

Anwendungsbeispiel des Levenshtein-Algorithmus

Im Folgenden wird der Levenshtein-Algorithmus zum Bestimmen der Anzahl an abweichenden Ziffern zu einem gesuchten Lösungstext verwendet.

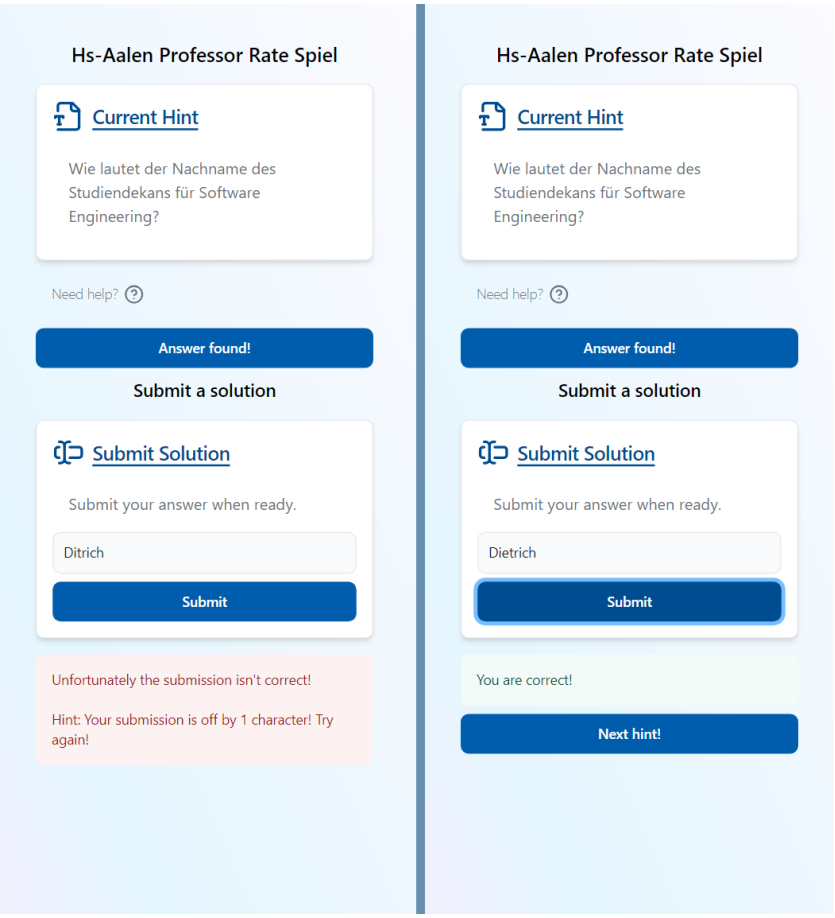

Abbildung 5.9.: Bildschirmabschnitte Text Unterschied an Teilnehmer

Der linke Bildschirmabschnitt in Abbildung 5.9 zeigt den Hinweis, wenn ein Teilneh-

mer sich bei seiner Lösung um eine Ziffer vertan hat. Der rechte Bildschirmabschnitt in Abbildung 5.9 wird angezeigt, wenn der Benutzer eine korrekte (Text-)Lösung eingegeben hat.

6. Inbetriebnahme

6.1. Einführung

In diesem Kapitel werden die Maßnahmen für das Deployment der Anwendung näher beschrieben. Anhand einer Übersicht wird gezeigt, wie das System im laufenden Zustand aussieht und wie die unterschiedlichen Anwendungen mit unserem Backend interagieren.

6.2. Backend Containerisierung

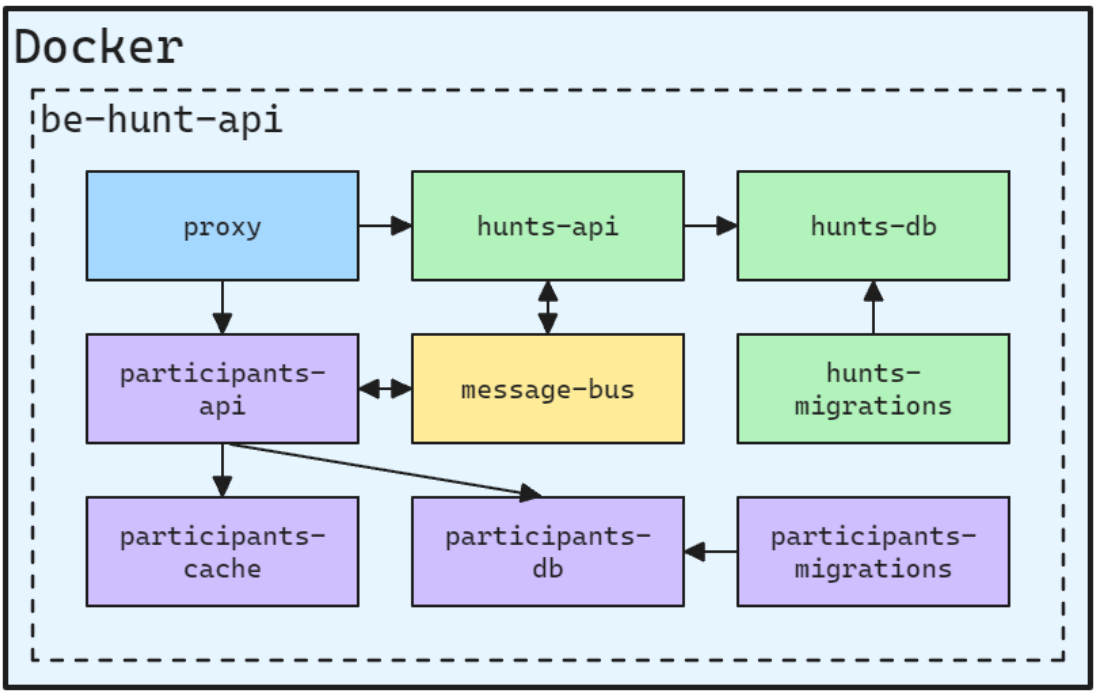

Abbildung 6.1.: UML-Diagramm für das Deployment

Abbildung 6.1 stellt die Deployment-Ansicht des Backends dar. Das Deployment wird durch die Docker Containerisierung ermöglicht (vgl. Kapitel 2.2.4). Die Konfiguration der verschiedenen Dienste wird über eine *Docker-Compose* Datei definiert.

Der Proxy (vgl. Abbildung 6.1 - blau) fungiert als zentraler Einstiegspunkt für die Anwendungen von außen. Er leitet alle Anfragen an den jeweils richtigen Dienst weiter. Bei falschen oder ungültigen Anfragen bricht die Verbindung ab. Die Implementierung und Konfiguration des Proxies ist im Anhang beschrieben.

Die beiden Container Participants-Api und Hunts-Api (vgl. Abbildung 6.1 - grün und violett) sind die deployten Release-Builds der entworfenen Services aus Kapitel 4.2. Diese haben jeweils eigene Datenbank-Container für die Persistierung von Daten.

Für die initiale Erstellung und Aktualisierung bestehender Datenbank-Container sind zusätzlich die beiden Migration-Container *Participants-Migrations* und *Hunts-Migrations* vorgesehen (vgl. Abbildung 6.1 - grün und violett). Beide öffnen eine Verbindung zur jeweiligen Datenbank und erstellen die initial benötigten Datenbanktabellen, damit diese für die Verwendung im jeweiligen Service mit den aktuellen Datenbanktabellen existieren.

Die jeweiligen Dockerfiles sind im Anhang näher beschrieben (vgl. Anhang B.1.1).

6.3. Frontend Containerisierung

Da das Frontend mit Web-Technologien (vgl. Kapitel 2.2.3) implementiert wurde, kann es als plattformunabhängige Browseranwendung auf einem Webserver deployt werden. Dies ist auch über Docker Containerisierung möglich, wurde aber im Rahmen des Projektes nicht entworfen und getestet.

Für das Deployment der Hunt-Editor Web-App für Desktops wäre es auch möglich, eine echte Anwendung daraus zu machen. Hierfür stehen bereits implementierte Technologien wie *Tauri* zur Verfügung (siehe [14] und [15]).

7. Zusammenfassung und Ausblick

7.1. Erreichte Ergebnisse

Die Zusammenfassung dient dazu, die wesentlichen Ergebnisse des Praktikums und vor allem die entwickelte Problemlösung und den erreichten Fortschritt darzustellen. (Sie haben Ihr Ziel erreicht und dies nachgewiesen).

7.2. Ausblick

Im Ausblick werden Ideen für die Weiterentwicklung der erstellten Lösung aufgezeigt. Der Ausblick sollte daher zeigen, dass die Ergebnisse der Arbeit nicht nur für die in der Arbeit identifizierten Problemstellungen verwendbar sind, sondern darüber hinaus erweitert sowie auf andere Probleme übertragen werden können.

7.2.1. Erweiterbarkeit der Ergebnisse

Hier kann man was über die Erweiterbarkeit der Ergebnisse sagen.

7.2.2. Übertragbarkeit der Ergebnisse

Und hier etwas über deren Übertragbarkeit.

Literatur

[1] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Part 1, Chapter 1: The Goal. Prentice Hall, 2018, S. 33.

[2] James Lewis und Martin Fowler. *Microservices: a definition of this new architectural term*. 20114. URL: https://martinfowler.com/articles/microservices.html (besucht am 20. 04. 2024).

[3] Code Maze. *Onion Architecture in ASP.NET Core*. 2024. URL: https://code-maze.com/onion-architecture-in-aspnetcore/ (besucht am 13. 07. 2024).

[4] Übersetzung von Jørgen W. Lang Mark Richards Neal Ford. *Handbuch moderner Softwarearchitektur*. O'REILLY, 2021, S. 184–189.

[5] Red Hat. *What are application programming interfaces (APIs)?* 2023. URL: https://www.redhat.com/de/topics/api/what-are-application-programming-interfaces (besucht am 20. 04. 2024).

[6] Unity. *AR Foundation*. 2024. URL: https://unity.com/de/unity/features/arfoundation (besucht am 20. 04. 2024).

[7] Salad Lab. *Json.Net for Unity3D*. 2016. URL: https://github.com/SaladLab/Json.Net.Unity3D/#upm (besucht am 23. 04. 2024).

[8] Daniel Fortes. *AR + GPS Location*. 2023. URL: https://assetstore.unity.com/packages/tools/integration/ar-gps-location-134882 (besucht am 23. 04. 2024).

[9] Svelte contributors. *Svelte - Cybernetically enhanced web apps*. 2024. URL: https://svelte.dev/ (besucht am 20. 03. 2024).

[10] Alex Bespoyasov. *My Experience with Svelte and SvelteKit*. 2022. URL: https://dev.to/bespoyasov/my-experience-with-svelte-and-sveltekit-342e (besucht am 06. 06. 2024).

[11] Tien Nguyen. *Svelte vs SvelteKit: Understanding Key Differences and Similarities*. 2023. URL: https://www.frontendmag.com/insights/svelte-vs-sveltekit/#Pros_and_Cons (besucht am 20. 03. 2024).

[12] Unity. *Project Setup*. 2024. URL: https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@5.1/manual/project-setup/project-setup.html (besucht am 20. 04. 2024).

[13] Paul Tassi. *The Entire Gaming Industry Unites Against Unity's Baffling Pricing Change*. 2023. URL: https://www.forbes.com/sites/paultassi/2023/09/13/the-entire-gaming-industry-unites-against-unitys-baffling-pricing-change/ (besucht am 26. 07. 2024).

[14] *GitHub.com - Tauri*. 2024. URL: https://github.com/tauri-apps/tauri (besucht am 30. 07. 2024).

[15] *Tauri*. 2024. URL: https://tauri.app/ (besucht am 30. 07. 2024).

A. Anhang: Entscheidungen bei der Implementierung

A.1. Architektonische Entscheidungen

A.1.1. Wahl eines Message-Bus für das Hunt-Api Backend

A.1.2. Wahl eines Api-Gateways für das Hunt-Api Backend

A.1.3. Hinweise und Lösungen ohne Abstrakte Klassen

Kontext

Für das Speichern der Aufgaben (*Assignments*) einer Schnitzeljagd soll es möglich sein, aus unterschiedlichen Arten von Hinweisen (*Hints*) und Lösungen (*Solutions*) zu wählen.

Entscheidung

Statt für jeden Hinweis-Typ und Lösungs-Typ unterschiedliche abstrakte Datenklassen zu definieren, werden diese lediglich als Daten-String gespeichert.

Begründung

Es wurde im Rahmen der Projektdurchführung versucht, mit abstrakten Datenklassen zu arbeiten. Jedoch erhöht dies die Komplexität enorm.

Zudem ist diese Lösung nicht skalierbar, denn es wird mehr Aufwand erfordert, einen neuen Hinweis-Typ oder Lösungs-Typ in das System einzubauen. Zunächst muss eine neue Datenklasse definiert werden, Dann muss diese in das Datenbank-Modell als konkrete neue Tabelle eingefügt werden.

In den meisten Fällen unterscheiden sich die Hinweis-Typen oder Lösungs-Typen

kaum am Datenformat. Texte oder Bilder werden im System als String gespeichert. Daher entspringt die zentrale Frage: Wieso überhaupt mit abstrakten Klassen alles verkomplizieren?

Konsequenzen

- **Keine Type-Safety**: Geo-Locations im Format *lat;lon* werden als verketteter String gespeichert und müssen beim Ausleseprozess jeweils umgewandelt werden, was sehr fehleranfällig ist.

A.2. Entscheidungen im Frontend

A.2.1. Wahl von Flowbite statt DaisyUI

Unterschied zwischen Flowbite-Svelte und DaisyUI

DaisyUI ist eine Komponentenbibliothek für Tailwind CSS, die eine Vielzahl von vorgefertigten UI-Komponenten bereitstellt. Es ist darauf ausgelegt, die Nutzung von Tailwind CSS zu vereinfachen, indem es gebrauchsfertige Klassen und Designs bietet. DaisyUI ist besonders bekannt für seine einfache Integration und die Möglichkeit, schnell und effizient ästhetisch ansprechende Benutzeroberflächen zu erstellen, ohne viel eigene CSS-Regeln schreiben zu müssen.

Im Folgenden ein Beispiel für die Verwendung von DaisyUI, um ein Modal zu erstellen:

```
1  <button class="btn" onclick="my_modal_1.showModal()">open
       modal</button>
2  <dialog id="my_modal_1" class="modal">
3    <div class="modal-box">
4      <h3 class="text-lg font-bold">Hello!</h3>
5      <p class="py-4">Press ESC key or click the button below
       to close</p>
6      <div class="modal-action">
7        <form method="dialog">
8          <!-- if there is a button in form, it will close
       the modal -->
9          <button class="btn">Close</button>
10       </form>
11     </div>
12   </div>
```

```
13 </dialog>
```

Quelltext A.1: Code Ausschnitt DaisyUI Beispiel

Flowbite-Svelte ist eine auf Tailwind CSS basierende UI-Komponentenbibliothek, die speziell für die Integration mit Svelte entwickelt wurde. Sie bietet eine umfassende Sammlung von UI-Komponenten wie Buttons, Modale, Formulare und vieles mehr. Flowbite-Svelte kombiniert die Leistungsfähigkeit von Tailwind CSS mit der reaktiven Natur von Svelte, um die Entwicklung dynamischer und interaktiver Webanwendungen zu erleichtern. Zudem bietet Flowbite-Svelte eine nahtlose Svelte-Integration, was bedeutet, dass die Komponenten als echte Svelte-Komponenten bereitgestellt werden, die sich perfekt in das Svelte-Ökosystem einfügen.

Im Folgenden ein Beispiel für die Verwendung von Flowbite-Svelte, das das gleiche Modal erstellt wie im DaisyUI-Beispiel:

```
1  <script>
2    import { Button, Modal } from 'flowbite-svelte';
3    let defaultModal = false;
4  </script>
5
6  <Button on:click={() => (defaultModal = true)}>open
       modal</Button>
7  <Modal title="Hello!" bind:open={defaultModal} autoclose>
8    <p class="text-base leading-relaxed text-gray-500
       dark:text-gray-400">Press ESC key or click the button
       below to close</p>
9    <svelte:fragment slot="footer">
10     <Button on:click={() => (defaultModal =
       false)}>Close</Button>
11   </svelte:fragment>
12 </Modal>
```

Quelltext A.2: Code Ausschnitt Flowbite-Svelte Beispiel

Begründung für den Wechsel zu Flowbite-Svelte

Der Wechsel von DaisyUI zu Flowbite-Svelte ist aus mehreren Gründen gerechtfertigt. Erstens bietet Flowbite-Svelte eine speziell auf Svelte abgestimmte Komponentenbibliothek, die eine engere und effizientere Integration ermöglicht. Dies bedeutet, dass die UI-Komponenten in Flowbite-Svelte als native Svelte-Komponenten verfügbar sind, was die Entwicklungsarbeit vereinfacht und die Nutzung von Svelte-spezifischen Features erleichtert.

Zweitens ist die Anpassungsfähigkeit und Erweiterbarkeit von Flowbite-Svelte ein wesentlicher Vorteil. Während DaisyUI zwar eine einfache Möglichkeit bietet, Tailwind CSS zu nutzen, ist Flowbite-Svelte besser darauf ausgelegt, komplexere und dynamischere Benutzeroberflächen zu unterstützen. Dies ist besonders wichtig für eine Anwendung wie den Schnitzeljagd-Editor, die eine hohe Interaktivität und eine benutzerfreundliche Oberfläche erfordert.

Darüber hinaus bietet Flowbite-Svelte eine breitere Palette an vorgefertigten Komponenten und Layout-Optionen, was die Entwicklung beschleunigt und gleichzeitig die Konsistenz und Qualität der Benutzeroberfläche verbessert. Dies trägt dazu bei, dass die Anwendung nicht nur funktional, sondern auch optisch ansprechend und intuitiv zu bedienen ist.

B. Anhang: Code Abschnitte

B.1. Inbetriebnahme

B.1.1. Docker-Compose für das Backend