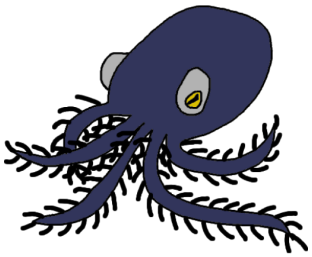
intension

DHBW
Duale Hochschule
Baden-Württemberg

Featherkraken: Bestpreissuche für Flugangebote mit variablen Abflughäfen



STUDIENARBEIT

des Studienganges Informatik
an der Dualen Hochschule Baden-Württemberg Stuttgart
von
Ingo Kuba

Matrikelnummer, Kurs	place, holder
Ausbildungsfirma	intension GmbH
Betreuer	Place Holder

Erklärung zur Eigenleistung

Hiermit erkläre ich, dass ich die vorliegende Studienarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Die Stellen der Studienarbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Ostfildern, den 4. Juni 2020 _____

Zusammenfassung

Bei Flügen über lange Strecken ist es häufig günstiger, Routen über nahe gelegene Flughäfen zu buchen. Um diese alternativen Routen zu finden, bieten Flugsuchdienste oft nur die Eingabe mehrerer Flughäfen an. Da diese für Reisende nicht alle bekannt sein können, wäre es besser einen Radius um den geplanten Abreiseort angeben zu können. Die Suchmaschine sucht darin nach Flughäfen und ermittelt Flüge von diesen zum Ziel.
Im Rahmen dieser Studienarbeit wurde eine Softwarelösung entwickelt, welche diese Funktionalität in Form einer Webanwendung umsetzt.

Abstract

For long flights, it is often cheaper to book routes via nearby airports. To find these alternative routes, flight booking services often only offer the entry of multiple airports. As travellers may not know all of them, it would be better to enter a radius around the planned departure location. The search engine will then search for airports in the given radius and find flights from these airports to the destination.
In the context of this study thesis a software solution was developed, which implements this functionality in form of a web application.

Inhaltsverzeichnis

Abbildungsverzeichnis

Akronyme

CDI Context Dependency Injection. 3

CI Continuous Integration. 14, 18

ERM Entity Relationship Model. 2, 6–8

IATA IATA airport code. 11, 12

Jakarta EE Jakarta Enterprise Edition. 3

UML Unified Markup Language. 2

1 Einleitung

1.1 Motivation

In der Regel möchte ein Fluggast den günstigsten Preis für eine bestimmte Route von A nach B. Jede Flugsuchmaschine im Internet bietet dieses Feature. Manchmal sucht ein Fluggast auch einfach nach Inspiration und möchte Angebote von A nach X, wobei X variabel ist. Einige Suchmaschinen bieten diese Suche ebenfalls bereits an. Worum es in dieser Studienarbeit geht, ist der umgekehrte Fall: X nach B. Also von welchem beliebigen Flughafen man möglich günstig an ein festes Ziel kommt. Gerade auf hochpreisigen Strecken kann es sich lohnen einen Umweg zu fliegen.

1.2 Aufgabenstellung

Bei der Suche sollen die klassischen Filterkriterien implementiert werden, welche als Pflichtanforderungen bezeichnet werden. Das heißt die Unterscheidung ob man nur einen Hinflug oder Hin- und Rückflug buchen möchte. Des Weiteren soll man jeweils ein Datum für An- und Abreise festlegen können, welches um drei Tage flexibel sein soll. Neben der Buchungsklasse (Economy, Business, First Class) soll auch die Wahl der Airline oder Allianz eingeschränkt werden können. Außerdem soll man Passagier- und Umsteigeanzahl wählen können.
Zusätzlich soll ein Entfernungsfilter um einen möglichen Abflughafen bereitgestellt werden. Zum Beispiel wird nur nach Angeboten gesucht, bei dem sich der Startflughafen maximal 800km (Entfernungsfilter) vom Flughafen Stuttgart (möglicher Abflughafen) entfernt befindet.
Diese Flugsuchmaschine soll über ein Web-Frontend vom Nutzer bedient werden können.

2 Wissenschaftliche Vertiefung

In diesem Kapitel werden technische Grundlagen aufgeführt und erläutert, welche für die Entwicklung des Projekts benötigt wurden.

2.1 Entity Relationship Model

Um das Datenmodell der Anwendung darzustellen wurde eine vereinfachte Variante des Entity Relationship Model (ERM) verwendet. Hier werden zunächst nur die Grundlagen zu dieser Modellvariante beschrieben ohne auf das eigentliche Datenmodell einzugehen, welches dann in Kapitel 3.3 vorgestellt wird.

Entities

Ein Objekt oder auch Entity wird in einem Rechteck dargestellt und kann Attribute besitzen, wobei komplexe Attribute als Beziehungen zu anderen Objekten dargestellt werden. Der Name der Beziehung entspricht hierbei dem Attributnamen. Die Anzahl der möglichen Relationen wird in UML-Notation angegeben. Zum Beispiel ist hier in Abbildung 1 zu sehen, dass eine Person null bis n Autos besitzen kann.

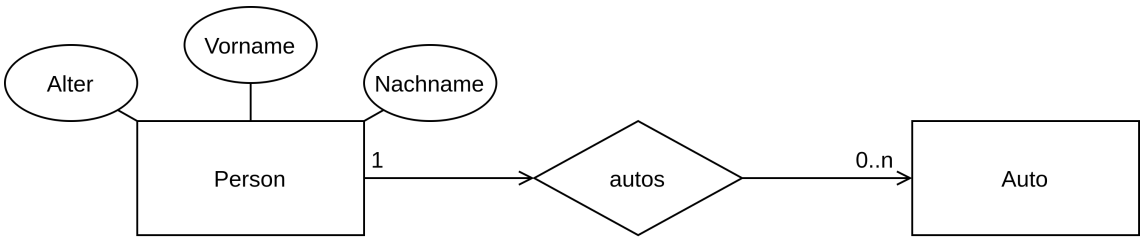


Abbildung 1: Beispiel für eine Entity mit Attributen und einer Beziehung

Attribute

Attribute können dabei eindeutig, optional oder mehrwertig sein. Die Unterscheidung zwischen Datum, Zahl oder Zeichenkette wird dabei in einem Entity Relationship Model nicht dargestellt.

Abbildung 2: Attribute (v.l.): eindeutig, optional und mehrwertig

2.2 Jakarta EE

Jakarta Enterprise Edition (Jakarta EE) ist eine Java-Spezifikation, welche Java SE (Standard Edition) erweitert und ermöglicht damit das Entwickeln von mehrschichtigen, zuverlässigen und sicheren Netzwerkanwendungen. Zuvor war Jakarta EE als Java EE bekannt.
(*Differences between Java EE and Java SE*, 2012)

2.3 MicroProfile

Eclipse MicroProfile ist eine Erweiterung von Jakarta EE und eine Alternative zu Spring, welches ein Framework der Firma Pivotal und vielgenutzte Lösung für Jakarta EE Anwendungen und Microservices ist.
MicroProfile bietet hingegen unter anderem Unterstützung für Context Dependency Injection (CDI), sowie eine bessere Konfigurierbarkeit der Anwendung.
(Monteiro, 2018)

2.4 Payara

Payara ist ein Application Server für Java-Anwendungen, insbesondere Jakarta EE. Ein Application Server führt Anwendungsprogramme aus und bietet zum Beispiel Dienste für Restschnittstellen, Authentifizierung oder Datenbankzugriff.
(Ottinger, 2008)
Payara basiert auf dem GlassFish Server, erweitert diesen jedoch um besseren Support sowie regelmäßigere Releases und Sicherheitsupdates.
(*Payara Server vs GlassFish*, o. J.)

2.5 Docker

Docker ist eine quelloffene Software zur Isolierung von Anwendungen in sogenannten Containern. Mit ihr können zum Beispiel Webserver oder Datenbanken schnell eingerichtet werden, ohne diese vorher aufsetzen zu müssen.

2.5.1 Image

Images sind die Vorlage zur Erstellung neuer Container. Sie können aus mehreren Schichten von anderen Images bestehen und so können komplexe Systeme leichtgewichtig und portabel erstellt werden.

2.5.2 Container

Ein *Container* ist die aktive Instanz eines Docker images und kann beliebig konfiguriert und bearbeitet werden. Wird ein Container beendet und ein neuer nach der Vorlage desselben Images gestartet, gehen alle Änderungen verloren. Dies lässt sich durch die Verwendung von *Volumes* verhindern, welche die Konfiguration eines Containers persistieren.

2.6 Postman

Postman ist ein Programm um HTTP-Schnittstellen aufzurufen um diese zu testen. Die einzelnen Aufrufe können in sogenannten *Collections* in Json-Format gespeichert werden, sodass sie in einer anderen Postman-Instanz importiert und wiederverwendet werden können. Durch diese beispielhaften Aufrufe ist es einfach, den Aufbau einer Schnittstelle zu verstehen.

2.7 React.js

React.js ist eine JavaScript-Bibliothek um Webanwendungen zu entwickeln. Damit lassen sich einfach interaktive Oberflächen erstellen, wobei man für jede Komponente der Anwendung sogenannte *Views* definiert. Dabei rendert React.js eine Komponente nur neu, wenn sich dessen Daten ändern. Dadurch wird die Anwendung effizient und der Code wird übersichtlicher und einfacher zu debuggen. Außerdem lassen sich Komponenten abstrakt definieren und mehrfach verwenden, was der gesamten Anwendung ein einheitliches Aussehen gibt. (*React*, 2020)

2.8 TypeScript

TypeScript ist eine auf JavaScript basierende Programmiersprache und erweitert diese um viele Funktionen aus der objektorientierten Programmierung. Der geschriebene Code wird zu JavaScript kompiliert, dadurch sind auch Bibliotheken und Syntax von JavaScript einsetzbar. Ein großer Vorteil gegenüber JavaScript ist, dass TypeScript strikter sein kann und somit logischer und verständlicher Code entsteht. (Technologies, 2020)

3 Entwurf

In diesem Kapitel werden Ansätze zur Implementierung der Anwendung verglichen und die jeweils gewählte Lösung aufgeführt. Die Ansätze reichen durch alle Abstraktionsebenen von Überlegungen zum Aufbau des Projekts, über Modellierung der Daten bis hin zu gewählten Frameworks.

3.1 Auswahl der externen Schnittstelle

Um Flugdaten zu erhalten muss eine externe Schnittstelle benutzt werden, welche die Pflichtanforderungen[1.2] erfüllt und dabei leicht zu verwenden ist. Die Wahl der Schnittstelle fiel dabei auf eine Rest-Schnittstelle, da diese sehr einfach zu benutzen sind. Für die Auswahl des Anbieters wurde eine Tabelle (siehe Abbildung 3) erstellt, welche die Pflichtanforderungen mit den Funktionalitäten der jeweiligen Schnittstelle abgleicht. Dabei erfüllte die Schnittstelle von Kiwi nicht nur alle Anforderungen, sondern war auch sehr gut dokumentiert und mit Beispielen beschrieben. Des Weiteren bot Kiwi auch eine Rest-Schnittstelle um Flughäfen mit Teilen des Städte- oder Flughafennamens zu suchen.

	Skyscanner	**Hipmunk**	**Kajak**	**Flight Data**	**Flight Bookings**	**Kiwi Flights**
Single flight	yes	yes	yes	yes	yes	yes
Two directions	no	no	no	yes	no	yes
Specific date	yes	yes	yes	yes	yes	yes
Flexible date	no	limited	limited	yes	no	yes
Class	yes	yes	yes	limited	yes	yes
Passengers	yes	yes	yes	no	yes	yes
Airline	yes	no	no	no	no	yes
Stops	no	no	no	no	no	yes

Abbildung 3: Tabelle zum Vergleich der Schnittstellen

3.2 Aufbau der Anwendung

Bei dem ersten Entwurf der Anwendung fiel auf, dass die gesamte Funktionalität mit zwei verschiedene Ansätzen gelöst werden kann. Zum einen könnte man externe Schnittstellen direkt aus dem Frontend aufrufen, sodass eine Serveranwendung nicht notwendig wäre. Dies hätte den Vorteil, dass der Technologie-Stack reduziert würde. Andererseits hätte es den Nachteil, dass das Projekt unübersichtlich werden könnte. Deshalb ist es sinnvoll, ein clientseitiges User Interface und eine separate Serveranwendung für die Funktionalität zu entwickeln und jeweils passende Technologien für beide zu finden.

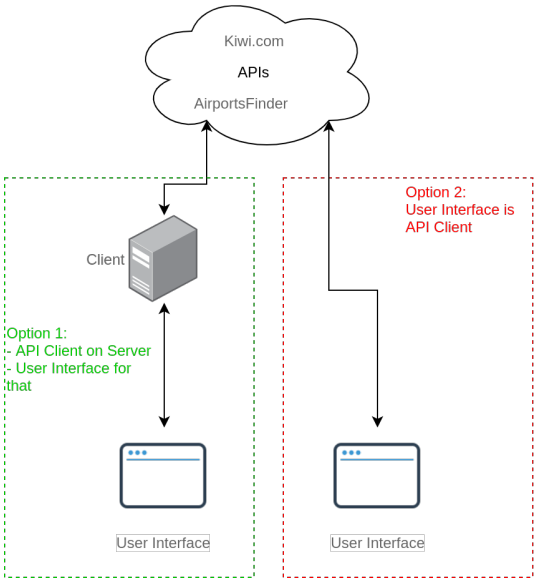


Abbildung 4: Aufbau der Anwendung

3.3 Datenmodell

Das Entity Relationship Model für das Datenmodell wurde hier aufgeteilt in Suchanfrage (`SearchRequest`) und Suchergebnis (`SearchResult`). Sowohl die Serveranwendung als auch das User Interface implementieren jeweils dieses Datenmodell, sodass Daten in einem einheitlichen und überschaubaren Format ausgetauscht werden können.

3.3.1 SearchRequest

Eingehende Anfragen an den Service (`SearchRequest`) haben verschiedene Parameter, welche die zu Beginn genannten Pflichtanforderungen[1.2] abdecken. Komplexe Attribute wie die Zeitspanne (`Timespan`) für An- und Abreise sowie der Ursprungs- und Ziel-Flughafen (`Airport`) wurden in eigene Objekte ausgelagert, damit sie so wiederverwendet werden können.

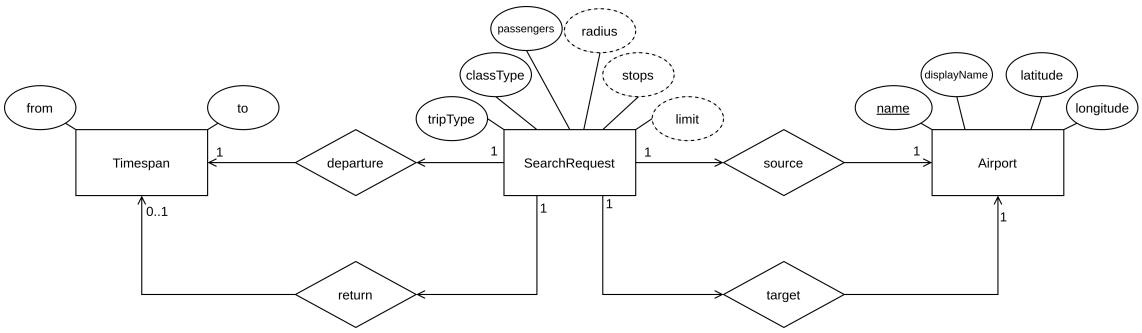


Abbildung 5: Entity Relationship Model des SearchRequest Objekts

3.3.2 SearchResult

Antworten des Service (`SearchResult`) sind etwas komplexer als Anfragen aufgebaut, teilen jedoch manche Attribute mit diesen. So enthält ein Suchergebnis eine Ansammlung von sogenannten `Trips`, welche sich aus Flügen (`Flight`) der An- und Abreise zusammensetzen. Diese Flügen haben widerrum selbstverständlich selbst jeweils einen Start- und Zielflughafen, welche in dem `Route`-Objekt mit Informationen zu Abflug- und Ankunftzeiten sowie der Airline enthalten sind.
Bei erster Betrachtung fällt auf, dass Informationen über Airlines und Zeiten redundant sind, was jedoch für die optimale Verarbeitung in der Oberfläche von Vorteil ist.

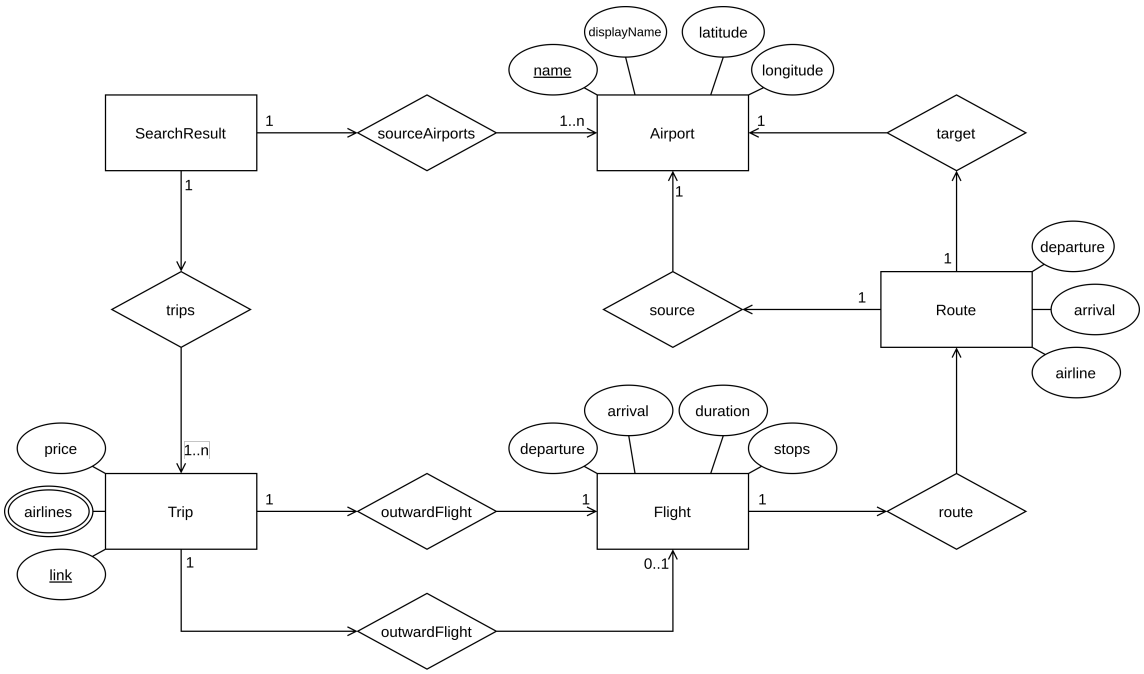


Abbildung 6: Entity Relationship Model des SearchResult Objekts

3.4 Framework

Die Anwendung wurde in eine Server- und eine Web-Anwendung unterteilt, für welche jeweils eine Framework-Technologie gewählt werden muss. Die Serveranwendung übersetzt Suchanfragen von der Oberfläche zu einem Format, welches die externe Schnittstelle akzeptiert und leitet das Sucherergebnis daraufhin an die Oberfläche zurück. Dies bedeutet, dass die Serveranwendung als Rest-Client für die externe Schnittstelle und selbst als Rest-Schnittstelle für die Oberfläche dient. Für die Komponenten war das Thema Plattformunabhängigkeit sehr wichtig und aus eigener Erfahrung eignet sich dafür Java sehr gut, da es eine große Community hat und eine Anwendung in kurzer Zeit aufgesetzt werden kann. Außerdem bietet Java einfache Möglichkeiten eine externe Rest-Schnittstelle anzusprechen. Als Bedingung für die Web-Oberfläche ist es unabdingbar, dass die Anwendung intuitiv bedienbar ist und auf allen Endgeräten gut dargestellt werden kann. Als Programmiersprache wurde dabei JavaScript gewählt, aus denselben Gründen der Dokumentation und persönlicher Erfahrung. Dabei kommen drei bekannte Frameworks in Frage: Angular, React.js und Vue.js. Dabei wurde React.js gewählt, welches sich besser als Vue.js und Angular für kleine Anwendungen eignet, wie es in diesem Fall zutrifft. Bei dem hier vorliegenden Fall handelt es sich sogar um eine sogenannte Single-Page-Webapplication, das heißt die gesamte Funktion kann auf einer Webseite dargestellt werden.

4 Implementierung

In diesem Kapitel wird die Umsetzung der Anwendung beschrieben. Wie bereits erwähnt, wurde die Anwendung in eine Serveranwendung (Backend) und eine Webanwendung (Frontend) unterteilt.

4.1 Backend

Das Backend der Anwendung dient als Proxy zwischen Frontend und externer Schnittstelle und wurde in Java geschrieben. Dabei wurde Jakarta EE[2.2] und Eclipse Microprofile[2.3] verwendet, um eine Rest-Schnittstelle für das Frontend zu bieten. Durch das Verwenden von Microprofile lässt sich die Anwendung einfach in einem Payara Application Server starten. Dieser wurde mithilfe eines Docker Images umgesetzt, wodurch er sich einfach in einer Cloud Umgebung oder klassisch auf einem Windows- oder Linux-Server bereitstellen lässt.

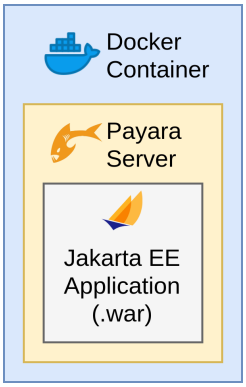


Abbildung 7: Aufbau des Backends

4.1.1 Services

Das Backend bietet verschiedene Rest-Services für ein User Interface, um nach Flughäfen und Flügen suchen zu können.

Flughafensuche: GET /airports Um bestimmte Flughäfen mit einem Teil des Flughafen- oder Städtenamens suchen zu können, gibt es den Endpoint `/airports` der mit einem GET-Request erreicht werden kann. Die Eingabe wird über den Query-Parameter 'query' in der URL des Requests übergeben. Ein beispielhafter Request für die Suche von Flughäfen in Amsterdam kann zum Beispiel folgendermaßen aufgebaut sein:

```
GET www.featherkraken-example.com/airports?query=Ams
```

Damit werden alle Flughäfen gesucht, welche den Wortteil 'Ams' im IATA airport code oder Städtenamen enthalten. Die Antwort des Services besteht aus einer Liste gefundener Flughäfen, welche jeweils IATA airport code, vollen Anzeigenamen und Koordinaten enthalten.

Flugsuche: POST /flights Die Kernfunktion des Backends steckt hinter dem Endpoint `/flights`, welcher eingehende Suchanfragen in ein für die externe Schnittstelle verständliches Format übersetzt und das Suchergebnis dann zurückgibt. Suchparameter werden hier nicht in der URL des Requests übergeben, sondern mit einem POST im Request-Body, da die Suchanfrage so übersichtlicher und leichter zu dokumentieren ist. Bei der Suche kann wahlweise ein Radius angegeben werden, dann werden Flüge von umliegenden Abflughäfen gesucht.

So kann eine Suchanfrage für Flüge von Frankfurt und Umgebung (100km) zum Beispiel so aussehen:

```
{
        "source": {
                "name": "FRA",
                "displayName": "Frankfurt International Airport",
                "latitude": 50.033056,
                "longitude": 8.570556
        },
        "target": {
                "name": "LAX",
                "displayName": "Los Angeles International",
                "latitude": 33.9425,
                "longitude": -118.40806
        },
        "radius": 100,
        "passengers": 1,
        "departure": {
                "from": "11.11.2020"
                "to": "13.11.2020"
        },
        "return": {
                "from": "22.11.2020"
        },
        "classType": "Economy",
        "tripType": "Round trip"
}
```

Für die Flughafenobjekte in `source` und `target` ist nur der eindeutige IATA airport code in `source.name` entscheidend. Der Radius wird in Kilometern angegeben. In diesem Beispiel werden Hin- und Rückflüge zwischen Frankfurt und Los Angeles in der Klasse 'Economy' gesucht. Außerdem ist zu erkennen, wie ein variables Abflugdatum (11.11. - 13.11.2020) angegeben werden kann.

4.2 Frontend

Das Frontend ist eine Web-Anwendung, die mit dem Framework React.js[2.7] in TypeScript[2.8] geschrieben wurde.
Durch den Einsatz von der Bibliothek `react-bootstrap` lassen sich Komponenten im Material Design einfach zusammenbauen. Material Design wurde von Google entwickelt und ist eine Design-Richtlinie für Oberflächen. Der Vorteil ist, dass man dabei kein Grafikdesigner sein muss, um eine ansprechende Webanwendung zu bauen.

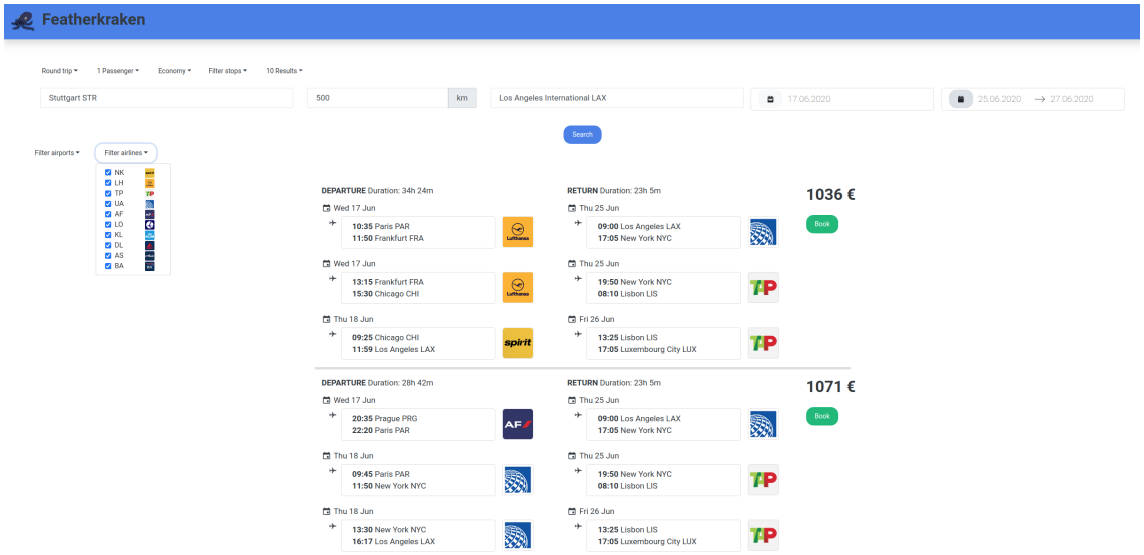


Abbildung 8: Screenshot des Frontend

Wie in Abbildung 8 zu sehen ist, kann bei der Suche nach Art des Fluges (nur Hinflug oder Hin- und Rückflug), Anzahl Passagiere, Klasse (Economy, Premium Economy, Business, First class), Anzahl der Zwischenstopps (Non-Stop, 1, 2 oder mehr) und Anzahl der Suchergebnisse gefiltert werden.
In den Eingabefeldern darunter können Suchparameter wie Startflughafen, Entfernungsfilter, Zielflughafen und An- und Abreisedatum eingegeben werden. Die Eingabefelder für Start- und Zielflughafen benutzen den Service zur Flughafensuche[4.1.1] um Flughäfen mit dem eingegebenen Wortteil in Echtzeit zu suchen.
Liefert das Backend nach Klicken des Suchbuttons Suchergebnisse zurück, lassen sich diese nach bestimmten Flughäfen und Airlines filtern. Letzteres ist an dem Dropdown auf dem Screenshot beispielhaft zu sehen.
Die Suchergebnisse sind in einer Liste aufgeführt und unterteilen sich jeweils in Hin- und Rückflug, wenn nicht nur nach Hinflug gesucht wurde. Ein Logo informiert den Benutzer über die Airline des jeweiligen Flugs.

5 Testing

Das Testen der Anwendung war ein wichtiger Teil für den Erfolg des Projekts. Deshalb wurden nicht nur alle Software-Komponenten mit Unit-Tests abgedeckt, sondern auch Integrations-Tests geschrieben. All diese Tests werden programmatisch ausgeführt und werden von einer CI in GitHub automatisch bei jedem *push* ausgeführt. Ein *merge* in den Master ist nur möglich wenn alle Tests erfolgreich ausgeführt wurden. Für alle Testfälle wurden realitätsnahe Bedingungen und Daten gewählt.
Zu guter Letzt wurden die Rest-Schnittstelle und die Oberfläche mit der Erwartung der gewünschten Ergebnisse regelmäßig manuell getestet und abgeglichen.

5.1 Automatische Tests

Das Java-Backend wurde mit Unit- und Integrationstests getestet, welche mit dem Testframework JUnit 5 geschrieben wurden. Dadurch fallen neue Bugs schon direkt bei der Entwicklung auf. Des Weiteren wird die Qualität des Codes mit *Sonar* überprüft. Dazu zählen unter anderem Aspekte wie Bugs, Schwachstellen (*Vulnerabilities*) oder Verletzung von Java-Richtlinien (sogenannte *Code Smells*). Um Schwachstellen in externen Bibliotheken der Anwendung aufzudecken wird *OWASP* verwendet.
All diese Tests werden automatisch in der Continuous Integration (CI) von *CircleCI* ausgeführt, wenn die Änderungen in das GitHub Repository gepusht werden. Pull Requests sind so konfiguriert, dass sie nur nach erfolgreichem Durchlauf der Tests akzeptiert werden können. Außerdem wird die Testabdeckung bei jedem Durchlauf dokumentiert und überprüft und darf niemals abnehmen. Die Konfiguration dieser Überprüfungen kann dem Dokument `featherkraken/.circleci/config.yml` entnommen werden.
Die Komponenten des React.js-Frontends wurden aus Zeitgründen nicht mit automatischen Tests abgedeckt, jedoch wird die Kompilierbarkeit der Anwendung in *CircleCI* überprüft und der Stand des *master*-Branches in GitHub Pages automatisch veröffentlicht. Dies kann in `featherkraken-ui/.circleci/config.yml` eingesehen werden.

5.2 Manuelle Tests

Zusätzlich zu den automatischen Tests gibt es eine *Postman-Collection*[2.6] für die Rest-Schnittstellen des Backends. Diese kann unter `src/test/postman/Collection.json` gefunden und in *Postman* importiert werden. Diese Anfragen wurden regelmäßig bei der Entwicklung des Backends verwendet, um Fehlerursachen zu finden und die Funktion sicherzustellen.
Außerdem gibt es Testfälle für das Frontend, um die Umsetzung der Anwendung mit den tatsächlichen Anforderungen vergleichen zu können. Folgend sind einige dieser Testfälle aufgeführt:

Abflughafen	**Zielflughafen**	**Klasse**	**Erwartung**
Inverness INV	John F. Kennedy International JFK	First class	Umstieg mit Economy Klasse wird gefunden
Heathrow LHR	John F. Kennedy International JFK	First class	Flüge werden gefunden
Munich MUC (+600km)	Kyiv International Airport (Zhuliany) IEV	Economy	Flug von LEJ gefunden
Stockholm Arlanda ARN	Dubai International DXB	First class	Umstieg mit Business Klasse wird gefunden
Sofia SOF	San Francisco International	Business	Umstieg mit Economy wird gefunden

Abbildung 9: Tabelle für die Testfälle des Frontends

6 Fazit

In diesem Kapitel wird zunächst in einem Ausblick auf Möglichkeiten zur Verbesserung der Anwendung eingegangen, denn Software-Entwicklung ist ein niemals abgeschlossener Vorgang. Danach werden bekannte Probleme und Einschränkungen aufgelistet, die noch nicht gelöst werden konnten, wobei jedoch Lösungsansätze zum weiteren Vorgehen aufgeführt werden. Alle Probleme sind in GitHub Issues festgehalten, sodass sie nicht in Vergessenheit geraten können. Zum Abschluss wird das gesamte Projekt noch einmal zusammengefasst.

6.1 Ausblick

Um die Funktion der Anwendung zu erweitern, könnte man zum Beispiel weitere externe Schnittstellen anbinden, um eventuell billigere Flüge als die von Kiwi.com zu finden. Dazu wurden die betroffenen Klassen des Java-Backends schon so abstrakt geschrieben, sodass dies mit angemessenen Aufwand umsetzbar ist.
Dabei ist jedoch anzumerken, dass diese externen Aufrufe bist jetzt nicht asynchron erfolgen. Das heißt, dass gewartet wird bis alle externen Schnittstellen geantwortet haben. Das hat zur Folge, dass die Antwortzeit äquivalent zur langsamsten Schnittstelle ist. Um das zu ändern, muss zunächst evaluiert werden, ob diese asynchron beantwortet werden können.
Als weitere Verbesserung kann die Geschwindigkeit der Aufrufe dokumentiert werden. Damit kann entschieden werden, ob bestimmte externe Schnittstellen zu langsam sind und damit wieder entfernt werden sollten. Für diese Art von Monitoring gibt es bereits Lösungen wie *Prometheus* oder die eingebaute Funktion von Eclipse Microprofile mithilfe sogenannter *Metrics*.
Des Weiteren kann auch ein Entfernungsfilter für den Zielflughafen auf die selbe Art wie für den Startflughafen hinzugefügt werden. Dazu müsste das Backend und die Oberfläche angepasst werden. Da dies jedoch nicht im Fokus der Aufgabe stand, sei das jedoch nur als Idee anzumerken.
Bei der Entwicklung der Anwendung fiel außerdem auf, dass es neben der Rest-Schnittstelle von Kiwi.com eine äquivalente GraphQL-Schnittstelle (siehe hier) gibt. Das könnte bedeuten, dass die Rest-Schnittstelle veraltet ist und deshalb bestimmte Probleme auftreten (siehe 6.2.2). In diesem Fall wäre es ratsam, auf die GraphQL-Schnittstelle umzusteigen.

6.2 Bekannte Probleme

6.2.1 AirportsFinder

Die externe Schnittstelle AirportsFinder zur Suche von Flughäfen in einem bestimmten Radius scheint eine Priorisierung der Flughäfen oder eine Einschränkung der Ergebnismenge vorzunehmen. So kann man zum Beispiel den Flughafen 'Leipzig/Halle LEJ' von Berlin aus mit einem Radius von 150km finden, jedoch nicht von München mit einem Radius von 360km. Dies hat zur Folge, dass für manche Flüge nicht der absolut günstigste Flug gefunden werden kann. Deshalb muss in Zukunft eine andere externe Schnittstelle zum Suchen von Flughäfen in einem gegebenen Radius gefunden werden.

6.2.2 Suche mit gemischten Klassen

Die Schnittstelle für die Flugsuche von Kiwi.com bietet die Möglichkeit, mit dem Parameter `mix_with_cabins` über mehrere Klassen zu suchen, wenn keine Flüge in der angegebenen Klasse gefunden werden können. Diese Funktion ist bei der Schnittstelle von Kiwi.com nicht gut dokumentiert oder fehlerhaft umgesetzt. Zur Lösung für dieses Problem könnten weitere Anfragen mit anderen Klassen abgeschickt werden, wenn die gegebene Klasse keine Ergebnisse liefert.

6.2.3 Deployment des Backend

Es wurde versucht, das Java-Backend auf *Heroku* zu hosten, damit sie jederzeit im Internet genutzt werden kann. Dies war aber selbst nach vielen Tagen nicht nachvollziehbar, also wurde dieser Ansatz verworfen. Als Zwischenlösung könnte die Anwendung auf einem Linux-Server bereitgestellt werden. Dies sollte automatisch mit jeder Änderung im GitHub Repository erfolgen, wie es mit *Heroku* versucht wurde.

6.3 Zusammenfassung

Nachdem die Aufgabenstellung evaluiert wurde, konnte schrittweise eine Softwarelösung entworfen werden. Dabei wurden verschiedene Abstraktionsebenen des Problems betrachtet und jeweilige Lösungen gefunden. Mit den gewählten Technologien wurde eine Webanwendung und eine Serveranwendung entwickelt, welche sich durch hohe Plattformunabhängigkeit auszeichnen. Es wurde darauf geachtet, dass die Anwendung leicht erweitert werden kann und der Code verständlich ist. Die Anwendungen werden mit Continuous Integration (CI) automatisch getestet und im Falle der Webanwendung auch immer mit dem neuesten Stand im Internet veröffentlicht. Abschließend wurden aufgetretene Probleme dokumentiert und Lösungsansätze aufgeführt.

Literatur

Differences between Java EE and Java SE. (2012). Zugriff auf https://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html

Monteiro, J.-L. (2018). *What is Eclipse MicroProfile?* Zugriff auf https://www.tomitribe.com/blog/what-is-eclipse-microprofile/

Ottinger, J. (2008). *What is an App Server?* Zugriff auf https://www.theserverside.com/news/1363671/What-is-an-App-Server

Payara Server vs GlassFish. (o. J.). Zugriff auf http://www.payara.org/payara-server-vs-glassfish

React. (2020). Zugriff auf https://reactjs.org/

Technologies, I. A. . (2020). *JavaScript VS TypeScript: Which is better? (2020 Updated).* Zugriff auf https://medium.com/@infinijith/javascript-vs-typescript-which-is-better-2020-updated-871866a3c68c