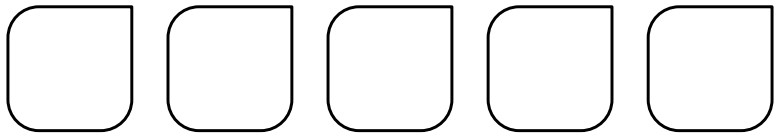


# 5 Pictures - Projektdokumentation

Projekt 2

Nils Blessing
Matrikelnummer: 32551
Betreuer: Prof. Dr. Marius Hofmeister

# Übersicht

# Anwendung

# Überblick

## Idee

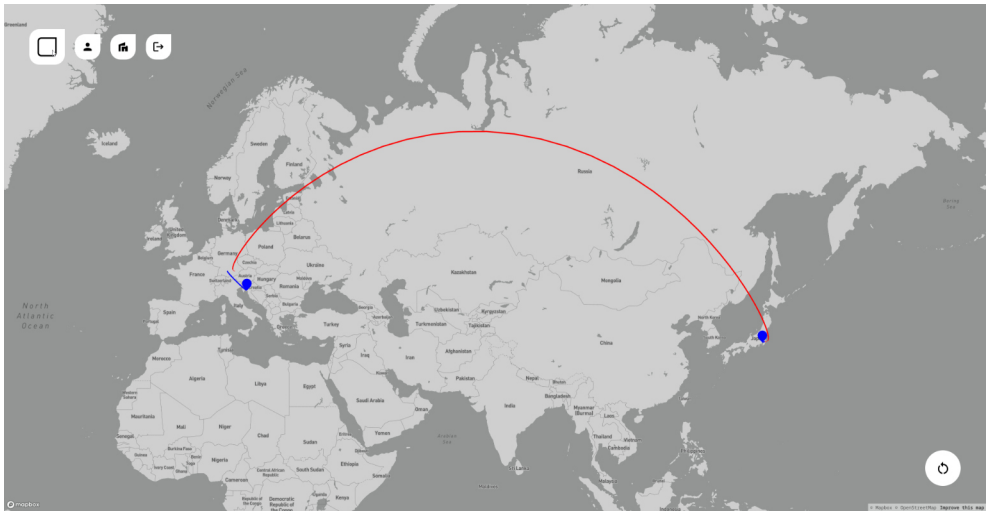
Wer an neue Orte in den Urlaub fährt, macht in aller Regel Fotos. Meist nicht gerade sparsam. Doch oft landen diese Unmengen an Bildern unsortiert auf dem heimischen PC oder einer NAS, um danach niemals wieder aufgerufen zu werden.

Mit „5 Pictures“ war die Idee, dem Nutzer eine interaktive Karte zu geben, auf der er seine Reisen grafisch und mit Zusatzinformationen speichern kann - inklusive fünf Bildern. Dadurch wird der Nutzer angeregt, genau zu überlegen, welche Bilder ihm am wichtigsten sind und den Urlaub seiner Meinung nach am besten widerspiegeln.

Angelehnt an Weltkarten, auf denen man freikratzen kann, wo man schon überall war, erhält der Nutzer so auch einen guten Überblick, über die Länder, in denen er bereits war.

# Überblick

## Screens

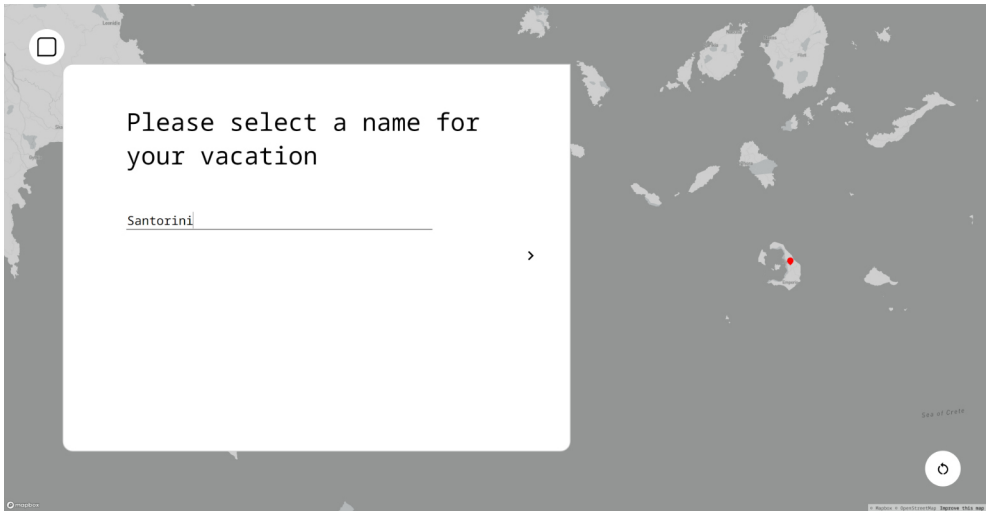


Start-Screen mit Flugrouten (rot) und regulären Routen (blau). Jeder Pin stellt eine Reise dar


Ansicht einer Reise mit nutzerspezifischen Informationen wie dem Abflug- und Zielflughafen, einer Bewertung und natürlich den Bildern

# Überblick

## Screens



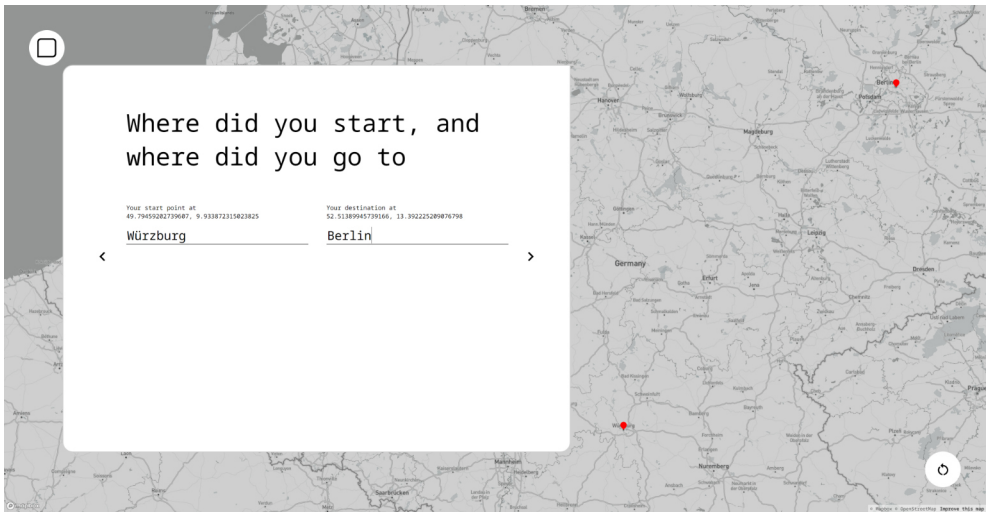

Ansicht zum Eintragen einer Reise. Der rote Pin auf der Karte entspricht hierbei dem Zielpunkt


Da die Reiseart für das Darstellen der Routen wichtig ist, wird der Nutzer aufgefordert, sein Beförderungsmittel auszuwählen

# Überblick

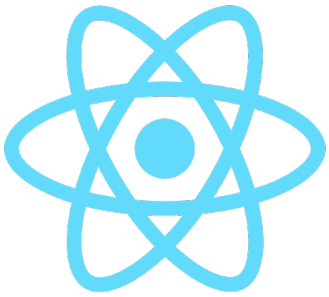
## Screens


Ist der Nutzer per Flugzeug gereist, erhält er statt einem Freitextfeld ein Dropdown mit allen Flughäfen mit IATA-Code. Gesucht werden kann entweder nach IATA-Code oder per Volltextsuche




Ist der Nutzer per Auto gefahren, so bekommt er zusätzlich zu seinem Ziel-Pin einen Pin an seiner aktuellen Geolocation, der seinen Startpunkt widerspiegelt. Die Beschreibungen der Punkte werden als Freitext eingegeben

# Frontend




deck.gl
{ data visualization at scale }

# Frontend

## React

Das Frontend wurde aufgrund des einfachen Navigationsansatzes mit React (npx create-react-app) umgesetzt. Die größte Herausforderung hierbei lag in der Struktur, in der einzelne Komponenten auf Daten aus jeweils über- oder untergeordneten Komponenten zugreifen müssen. Am stärksten wurde dieses Problem beim Erstellen einer Vacation deutlich. Da das Modal mit den Dateneingaben Informationen über die Koordinaten des Markers, der den finalen Ort eines Urlaubs darstellt, braucht, musste hier irgendwie ein Datenaustausch stattfinden, ohne extremes Propdrilling zu betreiben. Die Lösung hierfür liegt in der React Context API.

Diese erlaubt, mehrere Komponenten als children in einen Provider zu wrappen. Jedes Kind dieses Providers erhält dann über den useContext-Hook Zugriff auf die Daten, die der Provider exportiert:

Der Provider folgt hierbei regulärer HTML-Syntax:

```
<CreateVacationProvider>
    {/*displays the normal map*/}
    <Map markers={!error ? data.getVacation : []} isModalOpen={!!mapController.modalToShow}/>
    {//if user clicked on create, show the create dialoge
        (mapController.fromToCoords[1].length !== 0) && <CreateVacation refetch={refetch} />
    }
</CreateVacationProvider>
```

Jede Komponente, die sich in <CreateVacation/> befindet, kann über

```
const vacation = useContext(CreateVacationController);
console.log(vacation.createState)
//oder
const {createState, setCreateState} = useContext(CreateVacationController);
console.log(createState)
```

Zugriff auf alle Variablen und Funktionen erhalten, die der Provider exportiert.

# Frontend

## React

Der Inhalt dieses Providers spiegelt die Informationen wieder, die der Nutzer beim Erstellen eines Urlaubs eingeben muss. Das Objekt createState und die Funktion setCreateState werden dann im Rückgabewert der Funktion als value an den Provider übergeben.

```
export const CreateVacationProvider = ({children}) => {

    const mapController = useContext(MapController)

    const [createState, setCreateState] = useState({
        name: "",
        traveledBy: null,
        fromLocation: {
            iata: "",
            name: "",
            latLong: mapController.fromToCoords[0]},
        toLocation: {
            iata: "",
            name: "",
            latLong: mapController.fromToCoords[1],
            ifAirportFinalDestination: []},
        fromDate: null,
        toDate: null,
        rating: null,
        personalText: "",
        keys: [],
    })

    useEffect(() => {
        setCreateState({
            ...createState,
            fromLocation: {
                ...createState.fromLocation,
                latLong: mapController.fromToCoords[0]
            },
            toLocation: {
                ...createState.toLocation,
                latLong: mapController.fromToCoords[1]
            }
        })
    }, [mapController.fromToCoords[0][0],
        mapController.fromToCoords[0][1],
        mapController.fromToCoords[1][0],
        mapController.fromToCoords[1][1]
    ])

    const valuesForCreation = {
        createState,
        setCreateState,
    }

    return (
        <CreateVacationController.Provider value={valuesForCreation}>
            {children}
        </CreateVacationController.Provider>
    )
}
```

# Frontend

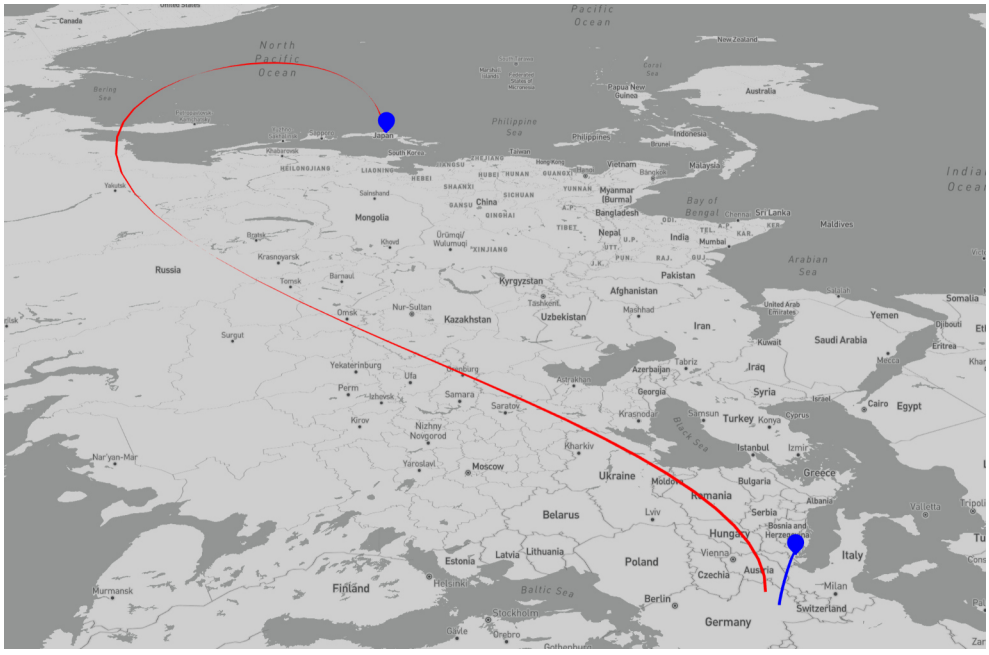
## Mapbox und deck.gl

Die Kartendaten werden über Mapbox bereitgestellt und durch react-map-gl in der Anwendung dargestellt.

Mapbox zeichnet sich besonders durch seine vielen Möglichkeiten im Design aus. Das Mapbox Studio ist eine Anwendung, mit der Karten beliebig personalisiert werden können. So kann beispielsweise angepasst werden, welche Farben Länder, Landesgrenzen, Gewässer, Flughäfen und Naturschutzgebiete haben sollen. Ein weiteres sehr schönes Feature ist die Bereitstellung von Höhendaten und Geometrie von Gebäuden. So bekommen Karten eine gewisse Tiefe.

Ergänzt werden diese Kartendaten durch deck.gl, einem sehr ausführlichem Package vom Taxiunternehmen Uber. Dieses bietet hauptsächlich Features zur Visualisierung von großen Datenmengen an, da rechenintensive Operationen mit diesem Package von Grafikkarten oder APUs übernommen werden.

Allerdings bietet deck.gl auch einen Arc Layer an, welcher zwei Punkte über Linien miteinander verbindet. Das interessante hierbei ist, dass diese Verbindungen nicht zwingend der Luftlinie auf einer Merkator-Projektion entsprechen müssen, sondern auch als „great circle“ berechnet werden können, sprich der Luftlinie auf dem Globus entsprechen. Diese Strecke wird dann wiederum auf die Merkator-Ansicht projiiert, wodurch sich eine interessante Krümmung ergibt.



# Frontend

## Bundling

Gebundled wurde das gesamte Frontend mit Webpack. Aus reiner Neugierde habe ich für dieses Projekt die Webpack Konfiguration selbst gemacht.  Neben CSS Autoprefixing unterstützt sie SVGs, welche auf der Karte in Form der Marker genutzt werden.

Gerade am Ende des Projektes hat sich das allerdings als wenig sinnvoll herausgestellt, da die Standard React Webpack Config komplett ausgereicht hätte.

# Backend


mongoDB

# Backend

## mongoDB - eine Dokumentendatenbank

Während sich sql-Datenbanken dadurch auszeichnen, dass jeder Datensatz der Struktur eines Tables folgen muss und jede Table Column einen Wert bekommen muss, geht mongoDB einen sehr viel lockeren Weg.

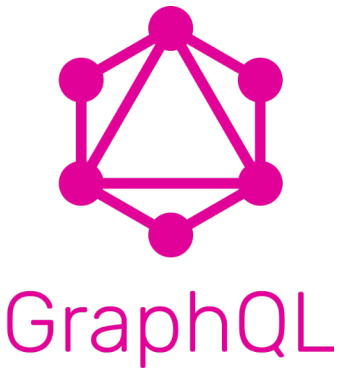
In einer mongoDB gibt es keine JOINS und Beziehungen zwischen Datensätzen, sondern jeder Datensatz steht für sich alleine. Dabei ist es in mongoDB auch egal, welche Daten ein Datensatz enthält. So kann beispielsweise ein Datensatz Informationen zu einem Buch enthalten, während der nächste Datensatz zu einem Autor gehört. Das heißt also, dass man in mongoDB keiner festen Datenstruktur folgen muss.

In gewisser Weise wird dieses Verhalten durch das ORM mongoose limitiert:

```
const requiredString = {
    type: String,
    required: true
}

const VacationScheme = new Schema({
    name: requiredString,
    traveledBy: requiredString,
    fromLocation: {
        iata: String,
        name: requiredString,
        latLong: [Number, Number],
    },
    toLocation: {
        iata: String,
        name: requiredString,
        latLong: [Number, Number],
        ifAirportFinalDestination: [Number, Number]
    },
    fromDate: {type: Date, required: true, default: Date.now},
    toDate: {type: Date, required: true, default: Date.now},
    rating: {type: Number, required: true},
    personalText: requiredString,
    keys: {type: Array, required: true},
    uid: requiredString,
})
```

Durch das Schema wird eine Struktur vorgegeben, an die man sich halten muss, wenn man ein Objekt dieses Schemas in die Datenbank speichern muss. Allerdings lässt sich durch das Feld „required“ definieren, dass gewisse Daten vorhanden sein müssen. Sind Felder nicht mit diesem Zusatz versehen, kann man sie leer lassen. Demnach werden sie auch nicht in die Datenbank geschrieben.


GraphQL

# Backend

graphQL - eine REST-Alternative

Entgegen einer REST-API mit vielen verschiedenen Routen, die alle HTTP-Methoden folgen, Params, Bodies und Header haben, gibt es in graphQL nur einen Endpunkt, nämlich /graphql.

Statt Informationen über die Nutzung eines Endpunkts über die URL oder den Body zu erhalten, gibt man in graphQL eine Query (vergleichbar mit GET) oder eine Mutation (vergleichbar mit POST) mit, die in graphqls eigener Sprache geschrieben ist.

Zwei Informationen sind hier zum Verständnis wichtig:

1. Jede Query/Mutation hat einen Namen und - wenn nötig - Parameter

Anhand dieses Namens matched die graphQL Middleware, welche Funktion ausgeführt werden soll. Erfüllt das request Objekt nicht die Anforderungen der query und hat beispielsweise keine Parameter, gibt graphQL direkt einen Error 404 zurück.

```
query VACATION($uid: String!){
        getVacation(uid: $uid) {
            name,
            id,
            traveledBy,
            fromLocation {
                latLong
            },
            toLocation {
                latLong,
                ifAirportFinalDestination
            }
        }
    }



Query: {
        getVacation: async (_, {uid}) => {
            let allVacationsFromUser = Vacation.find({uid: uid})
            return allVacationsFromUser
        }
}
```

# Backend

graphQL - eine REST-Alternative

2. Jede Query/Mutation hat einen festgelegten Rückgabetypen

In der typeDefs.js wird für graphQL definiert, welche Datenstrukturen genutzt werden. Sogenannte Types sind Objekte, die jedem Feld des Types einen festen Datentyp zuweisen. Diese Types können auch genestet werden, das heißt Type Vacation kann für das Feld fromLocation den custom Type AirportData haben.

Jede Query und Mutation hat als Rückgabetyp entweder einen primitiven Datentyp oder einen eigenen Type.

```
type Vacation {
    name: String!,
    traveledBy: String!,
    fromLocation: AirportData,
    toLocation: AirportData,
    fromDate: String!,
    toDate: String!,
    rating: Int!,
    personalText: String!,
    keys: [String]!,
    id: String!,
    uid: String!
}

input Airport {
    iata: String,
    name: String!,
    latLong: [Float!],
    ifAirportFinalDestination: [Float!]
}

type AirportData {
    iata: String,
    name: String!,
    latLong: [Float!],
    ifAirportFinalDestination: [Float!]
}

type Query {
    getVacation(uid: String!): [Vacation],
    getVacationByVacationId(id: String!, uid: String!): Vacation!,
    getImagesViaID(id: String!, uid: String!): [String!]!
}

type Mutation {
    tryIfItWorks(testString: String!): String!,
    createVacation(
        name: String!,
        traveledBy: String!,
        fromLocation: Airport,
        toLocation: Airport,
        fromDate: String!,
        toDate: String!,
        rating: Int!,
        personalText: String!,
        keys: [String!]!,
        uid: String!
    ): Vacation!,
    deleteVacationById(id: String!, uid: String!): Boolean
}
```

# Backend

graphQL - eine REST-Alternative

Möchte man die Query „getVacation“ ausführen, so muss beispielsweise eine uid (für den Nutzer) mitgegeben werden, damit die Funktion ausgeführt werden kann.

```
type Query {
    getVacation(uid: String!): [Vacation]!
}
```

Das Ausrufezeichen hinter dem String bedeutet in graphQL-Syntax, dass ein Wert zwingend erforderlich ist.  Ohne ID kann keine Datenbankabfrage stattfinden, demnach würde der Server einen Error-404 zurückgeben, wenn die ID in der request fehlt.
Selbiges gilt für den Rückgabewert der Funktion, welcher hinter dem Doppelpunkt angegeben ist. Diese Query liefert also immer ein Array zurück, dessen Inhalt - sofern es einen gibt,  ein neuer Nutzer würde ein leeres Array zurückgeben - immer dem Type Vacation folgen würde.

In dieser Struktur liegt ein großer Vorteil von graphQL. Da der Server genau weiß, welche Daten zurückgegeben werden können, hat der Client Einfluss darauf, ob er alle Daten, die ihm der Server zur Verfügung stellen kann, haben möchte, oder nur eine Auswahl davon.

Beispiel: Beim Setzen der Marker benötigt der Client die eindeutige ID des Urlaubs, die Reiseart, sowie die Latituden und Longituden der zu setzenden Punkte. Daten wie das Rating, der persönliche Text oder der Name sind hier irrelevant. In der Query gibt man dann einfach hinter den curly braces an, welche Daten man zurückhaben möchte. Dadurch löst sich das Problem des Overfetchings, welches bei REST-APIs häufig vorkommt.

```
query VACATION($uid: String!){
        getVacation(uid: $uid) {
            id,
            traveledBy,
            fromLocation {
                latLong
            },
            toLocation {
                latLong,
                ifAirportFinalDestination
            }
        }
    }
```

= Auswahl der Felder des Schema „Vacation“

# Backend

`graphQL - eine REST-Alternative`

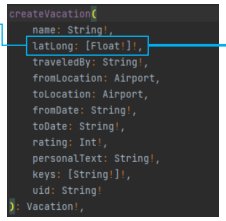
Bei der Verwendung von graphQL sind mir allerdings auch zwei negative Aspekte aufgefallen:

1. Errorhandling

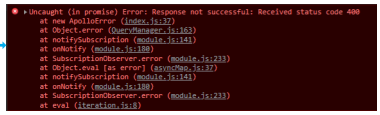
Entspricht ein in der request übergebenes Objekt nicht exakt dem vorgegebenen Type, so gibt der Server eine direkte Response mit Errorcode 404 zurück - ohne jede Erklärung weshalb. Wünschenswert wäre ein JSON-Objekt in der response mit Informationen wie beispielsweise „Type Mismatch. Field XY does not match expected data type“. Gerade zu Beginn und in Types mit vielen Feldern erschwert das das Debugging erheblich.

Request Objekt


Type



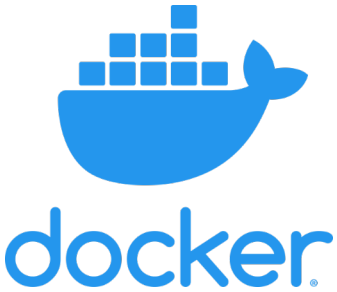

Response




2. Operationen, die über primitive Datentypen hinausgehen

Geht die request über primitve Datentypen hinaus, wird graphQL schnell unübersichtlich. Gerade der File Upload hat ein großes Problem dargestellt, da hierzu ein Type benötigt wurde, welcher Dateien im Format {createReadStream, filename, mimetype, encoding} erwartet. Da der Standard Apollo graphQL Client aber nativ keinen File Uploads unterstützt, muss ein Drittanbieter-Package installiert werden, welches auf Basis des normalen Clients eine Upload-URI bereitstellt.

Da dieser Type und der Client Upload jeweils aus unterschiedlichen Quellen stammt, war diese Lösung sehr unzuverlässig, weshalb der Upload über einen extra REST-Endpunkt nur für diesen Zweck geregelt wird.


docker

# Backend

## Docker

Docker ist ein Tool zum „containern“ von Anwendungen durch virtuelle Maschinen (Containervirtualisierung). Konkret bedeutet das, dass Docker anhand den Anleitungen einer sogenannten Dockerfile den Deploymentprozess von Entwicklungscode hin zu Produktionscode übernimmt und das finale Resultat in ein Image schreibt. Auf Basis dieses Images können beliebig viele Container erstellt werden. Diese stellen in gewisser Weise Instanzen des Images dar.

Da Docker sehr gut im cachen ist, empfiehlt es sich, die Dockerfile so feinschichtig wie möglich aufzubauen. Sieht Docker, dass ein gewisser Step nicht von Codeänderungen betroffen ist, so wird das Resultat dieses Steps aus einem vorherigen Imagebuild übernommen. Beim Installieren von Node-Packages kann dies beispielsweise sehr viel Zeit sparen.

Das Baseimage

```
#React
FROM node as reactBuild
```

Workingdirectory

```
WORKDIR  /frontend
```

Kopiert die package.json

```
COPY package*.json ./
```

Installiert node_modules

```
RUN npm install
```

Wurden während des letzten Builds keine neuen Packages installiert, verläuft der gesamte Prozess bis hier komplett aus dem Cache

Kopiert den restlichen Code

```
COPY . ./
```

Baut aus React-Code den Code fürs Deployment

```
RUN export NODE_OPTIONS=--openssl-legacy-provider && npm run build
#/React
```

Das Baseimage

```
#Nginx
FROM nginx
```

Kopiert aus React-Build den Code ins nginx-Verzeichnis

```
COPY --from=reactBuild /frontend/dist / usr/share/nginx/html/
```

Löscht Standard Config

```
RUN rm /etc/nginx/conf.d/default.conf
```

Kopiert neue Config

```
COPY nginx/nginx.conf /etc/nginx/conf.d
```

Öffnet Port 80 nach außen

```
EXPOSE 80
```

Startet nginx

```
CMD ["nginx", "-g", "daemon off;"]
#/Nginx
```

# Backend

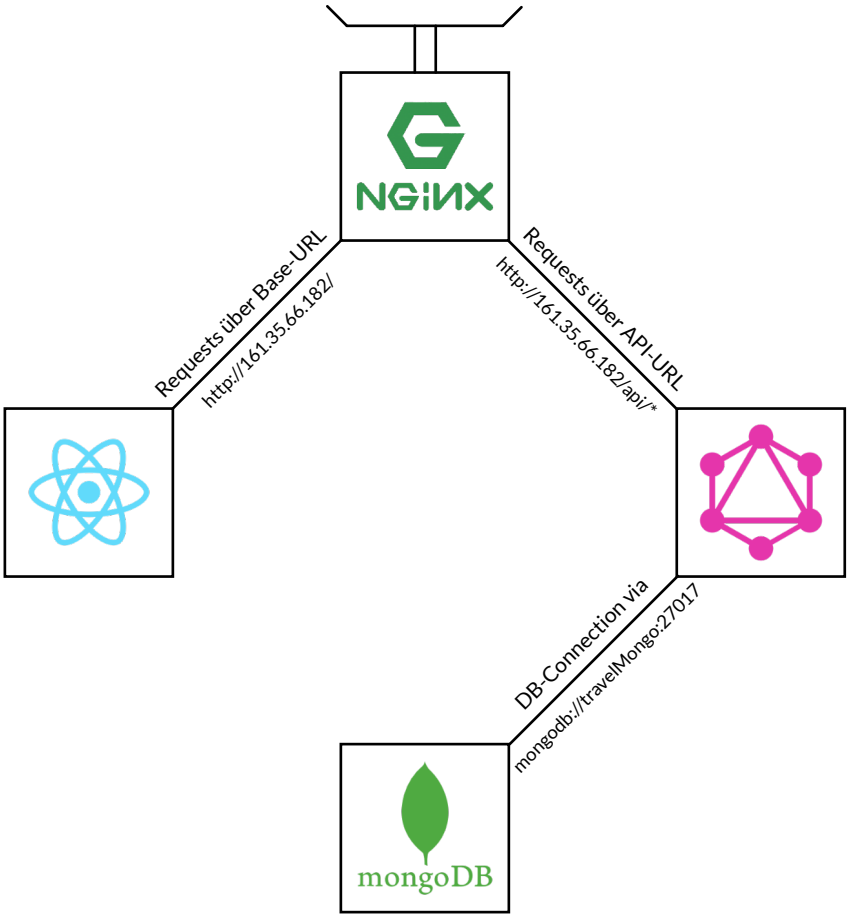
## Docker

Ergänzt wird Docker durch docker-compose. Dieses Tool erlaubt die Zusammenarbeit mehrerer Container aus jeweils unterschiedlichen Images.

Prinzipiell sind alle Container in sich geschlossen, allerdings bietet docker-compose die Möglichkeit, virtuelle Netzwerke zu erstellen und Containern Zugriff auf diese zu gewähren. Der große Vorteil hierbei liegt im internen DNS. Exposed ein Container (wie in diesem Fall), in dem ein Backend läuft, Port 4000, so können alle Container im selben Netzwerk über *servicename*:4000 Requests an dieses stellen.

So lassen sich auch Datenbanken komplett ohne direkte Verbindung zur Außenwelt nutzen. Da die Container alle nur untereinander kommunizieren reicht eine Verbindung zwischen Backend und Datenbank.

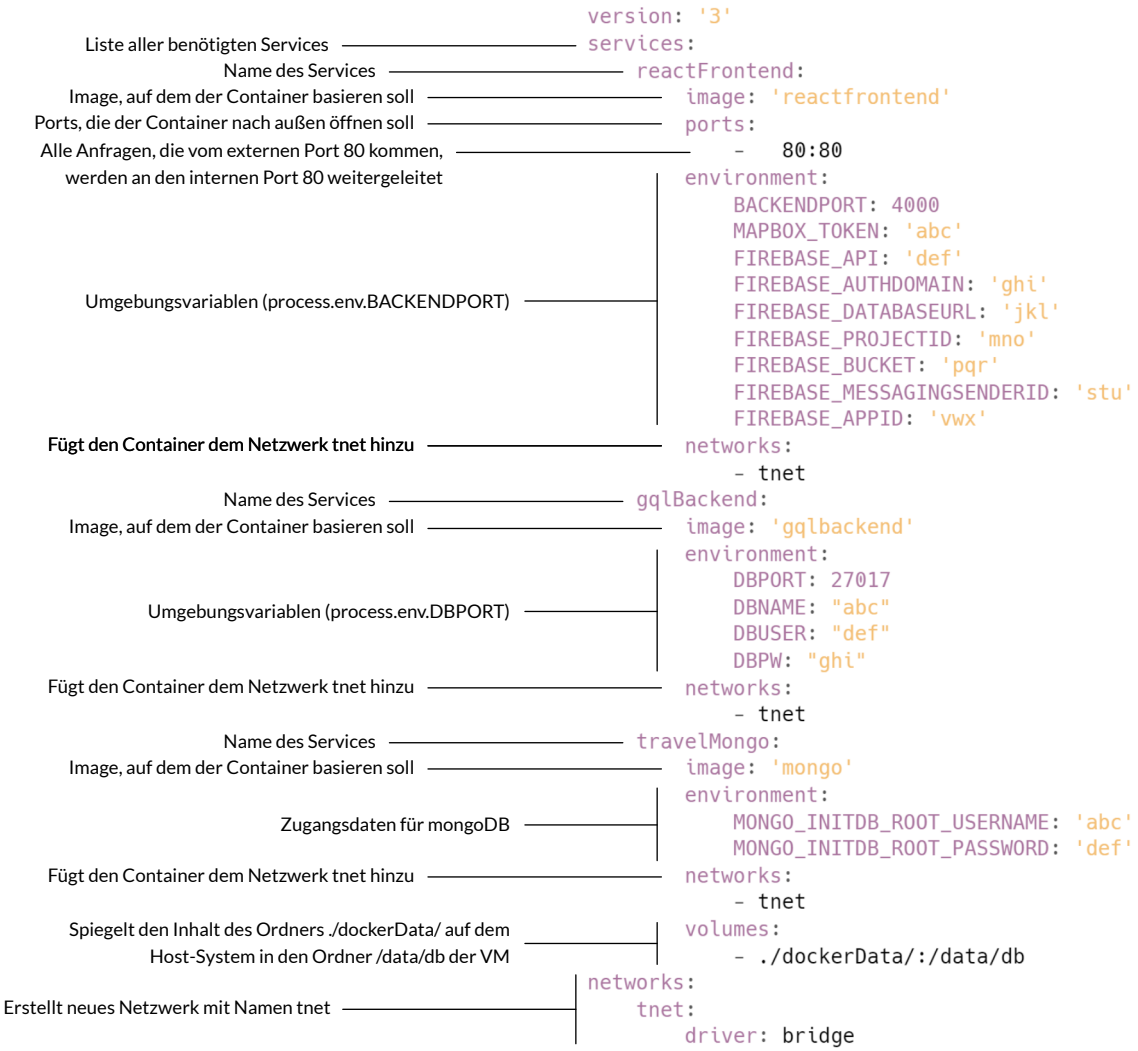
Aus diesem Umstand ergibt sich der Aufbau für dieses Projekt:



# Backend

## Docker

Diese ganze Struktur lässt sich durch den Befehl „docker-compose up“ komplett hochfahren und über „docker-compose down“ wieder restlos entfernen. Einzige Bedingung hierfür ist das Vorhandensein einer docker-compose.yml. Diese enthält alle Informationen zu den Containern, ihren Images, Environment-Variablen und Volumes. Der Aufbau hierfür ist der folgende:



# Backend

## Docker

Gerade beim Thema Backups bietet Docker einen weiteren Vorteil.

Container und Images ohne State (also in diesem Fall das nginx Image und das Backend) können einfach über den Dockerhub gesichert werden. Das empfiehlt sich sowieso, da dies zusätzlich ermöglicht, das Image lokal zu bauen, hochzuladen und dann auf dem Server direkt über die docker-compose.yml herunterzuladen.

Statefull Container - wie mongoDB - lassen sich am einfachsten über Volumes sichern. Diese sind wie Übergänge zwischen VM und Host. Der Ordner fpr mongoDB, in dem alle Datenbankeinträge gesichert werden, ist standardmäßig in ~/data/db. Diesen Ordner kann man in jedes beliebige Verzeichnis auf dem Host System spiegeln und dann beispielsweise über einen Cronjob in regelmäßigen Abständen auf externe Ressourcen sichern.


NGINX

# Backend

## Nginx

Nginx dient in dieser Anwendung als „Gatekeeper“. Der Container mit nginx-Instanz ist der einzige Container, der tatsächlich direkten Zugang zum freien Internet hat. An dieser Stelle fungiert es als reverse-proxy, der Anfragen über die Base-URL direkt mit dem React-Build bedient und alle API-Requests, die über http://url/api/* eingehen, ans Backend via upstream weitergibt.

```
upstream api {
    server gqlBackend:4000;
}

server {
    listen 80;
    location / {
        root /usr/share/nginx/html/;
    }
    location /api/ {
        proxy_pass http://api/;
        proxy_set_header Host $http_host;
    }
}
```