Technical University of Cologne

Report on the Practical Phase

Technology
Arts Sciences
TH Köln

Silvan Büdenbender

Supervisor: Prof. Dr. Hubert Randerath

A report submitted in partial fulfilment of the requirements
for the practical phase of the Bachelor in Computer Science & Engineering

in the

Faculty of Information, Media and Electrical Engineering

May 29, 2019

Contents

List of Figures

Chapter 1

Introduction

This project took place in the setting of a business analytics web application. Eventually it should become Software as a service. What that means will be briefly outlined below.

1.1 Business Analytics

Business analytics in this case refers to the fact that given application provides capabilities to browse and interpret advertisement data. A more in depth explanation of the provided functionality will be given in 2.3.

1.2 Web Application

A web-application is a tool that is provided online without the need to install any software but a web browser to access the internet. The distinction between website and web-application is not a clear one but it might be along the lines that web-applications deliver more dynamic content which allows for a richer interaction with the user while websites mostly provide static content. In addition to this web-applications, once loaded, may continue to work offline.

In this project the web-application can further be specified as a Single-page application or SPA for short. SPAs try to mimic behavior of desktop applications to enhance the user experience. They do so by dynamically changing the HTML-page the user is currently interacting with instead of loading a new document from the server.

1.2.1 State Handling

Whether we create a dynamic website/web-application/SPA we always need to think about handling state on the client-side e.g. in the browser. There will most likely be user data as well as application data that will be send to the client. Two technologies that provide a framework for state handling will be presented in 5.4.

1.2.2 Client-Server Communication

As with anything online, there has to be a server to offer the requested website/web-application. Apart from that there might as well be further resource-servers or authentication-servers. To make those interact properly we have to enable them to communicate with each other.

Another decision in question is how the workload of a web-application/SPA is split between the client and the server.

1.3 Software as a Service

Software as a Service, commonly abbreviated as SaaS, is a business model also called "software on demand" which allows the end user to use a subscription-based payment model accessing the software usually through a web-application with no other requirement than internet access, a browser and online sign-up. Often SaaS-products offer a free plan which allows end-users to try the product before they decide to invest any money.

1.4 Outlook

This project takes a look at the data being analysed, how it flows from its origin to the end-user and how to improve on that flow. How the described setting plays into this flow will become clear over the next few chapters.

Chapter 2

The Company

The company enabling me to do this project is Gipfelstürmer Kommunikation GmbH. It is a small company located in the beautiful and lively Ehrenfeld, Cologne. They are an advertising agency focused on the so called "Point of Sale" or POS.

2.1 Employees

About 10 permanent employees work at Gipfelstürmer Kommunikation with support from freelancers every now and then.

Most notably are Dr. Isabelle Engelhardt as CEO (chief executive officer), Silke Amelang as Head of Creation and Ralf Schmitt as Head of Technology, who has been my mentor during this project.

2.2 Philosophy

As I got to know Gipfelstürmer Kommunikation, who has been my employer for round about two years now, it always felt at home. Not home as "in my room", but as home in "another room of home where I sit down to work" which I deeply appreciate. There is a lot of respect and transparency in place, in combination with a flat hierarchy. I feel encouraged as well as required to give my best effort on what I do.

2.3 Products

Gipfelstürmer Kommunikation offers a variety of products related to advertisement. Those are described below.

2.3.1 Print

Gipfelstürmer Kommunikation provides all the print media related to the POS like flyers, posters, stickers, banners and any other the client requests.

2.3.2 Digital

Gipfelstürmer Kommunikation provides an online market place for ordering personalized print media to re-sellers of clients of the company.

Further they offer the BrandObserver as their latest product to give valuable insights in advertising data. It will be explained in chapter 3.

Chapter 3

The Project

This project is about transforming the data-flow in the given application. Data-flow refers to the whole process which starts by importing data into our database and ends when data is presented to the end user. Below a detailed outline of the application will be given, then further objectives and requirements to this project will be defined.

3.1 Status Quo

The web-application that is the subject of this project is called BrandObserver.

3.1.1 Functionality

The BrandObserver is a tool that provides insight on advertisement data. Initially a new customer can choose which brands are of interest to him. Those then will define the cost (more brands equals more data being bought) and be visible inside the application. The data being bought has the following structure (simplified for brevity):

- Item
  - categories (brand/medium/...)
  - visual/audio data
  - list of occurrences
  - metadata

So as it can be seen the underlying data will be a list of items whose "brand"-category matches one of the chosen brands of the customer. An item corresponds to one advertisement, for example a TV-spot or a newspaper article. The "medium"-category will tell what of those it is. The most important bits of data are for one part the visual/audio representation of the item as it allows the customer to actually experience the advertisement. The other essential part are the list of occurrences. Those list all events of that advertisement being recorded in the physical world. For a TV-spot it will list at which time on which channel it was broadcasted, for a newspaper article it will list appearances in different newspapers. Crucial to this is the event date and time, as well as the estimated costs of that occurrence, both are provided.

Then the Brand-Observer allows for navigating this data first by specifying a timespan of interest and then further grouping items belonging to that timespan by their categories. An item belongs in a timespan when at least one occurrence of it has happened during that timespan. While navigating so, the user will see averages, sums and time-series plots over the given timespan and grouping. All these computations happen in the browser. At several points he has the possibility to get a detailed view of one item which lists all the available data of that item.

3.1.2 Technology

The Brand-Observer is constructed as a SPA build on top of a MEAN stack. MEAN consists of a **M**ongoDB, a document-oriented database, **E**xpress.js, a web-server framework, **A**ngular2+ and **N**ode.js as a JavaScript execution environment. At the start of this project the Angular2+ front end uses Redux for state handling and REST for communication with the back-end. There are python services used for importing data into our MongoDB. A much more detailed description and explanation is supplied in chapter 5.

3.1.3 Data Flow

At the start of this project there were two primary endpoints in the express based back-end, one for authentication and one for loading a bulk of data. That bulk of data was then transformed on the client side according to some user configuration. The transformation was tightly coupled to the view/route the user was on.

Only very rudimentary caching was in place. As the data being loaded depends on the timespan selected by the user, if a newly selected timespan was enclosed by the previous one, it was read from the existing data, else a new request to the server was made.

3.2 Problems and Objectives

The project was facing issues with the client side aggregation logic, which had proven to be really fast once the necessary data was present in the browser but came with some caveats. Mostly it seemed really complicated to acquire the data in the correct form as it has been formerly specified to be a certain shape which was not in line with the actual requirements of the application so some unnecessary complexity had been introduced to account for that. A second issue, not really in scope as this project started, was the amount of data which was send to the client. Not an actual issue but an inconvenience arose from the use of Redux for state-handling. While it brings some really nice features, it also brought quite some boilerplate code and unclear structure with it as explained in chapter 5.4.2.

After a brainstorm to tackle the first issue the idea came up that every component in the application should be able to give a description of the data it wants and then it should just receive that data. As there were plans for an in-house solution to this, a colleague stumbled across a relatively new technology which had quite the uprising at that time called GraphQL. That technology seemed like a really good fit for the issue and will be elaborated on in chapter 5.3.

In addition to that the application needed refactoring in order to become customer-agnostic while at the start of this project it had grown too close to one specific customer.

3.2.1 Objective One

Reimplement data loading/transformation/state with the concept and tools of GraphQL.

3.2.2 Objective Two

Decouple application from any customer specific code.

3.3 Requirements

The resulting application should be scalable, still fast. It should be easy to grow the application on the foundation that will be laid with that project, e.g. adding new features or setting up new customers.

To be more precise, response time should be constantly low to provide a flawless user experience.

Just as important is an increase in code structure and simplicity of data-flow. Typescript (see chapter 5.1) features should be leveraged to enhance code quality.

Due to the loading of bulk data a more advanced caching algorithm should be created.

Chapter 4

Concept of the Solution

It is recommended to the reader to take a good look at chapter 5 before continuing.

The first step was to explore and read up on the concepts of GraphQL. The behavior and capabilities of that technology have to be explored in order to find an appropriate way to restructure the existing application accordingly.

4.1 Preparation

Starting at the official documentation of GraphQL (1) more information was collected from the Apollo documentation (2) and GraphQL Yoga (3). During further research GraphQL Code Generator (4) had been found, which combines nicely with TypeScript.

4.2 Findings

After a lot of reading it has been decided that every Angular component should have its own query. That would allow for a complete decoupling of the component hierarchy and the data flow, as components could order their desired data from wherever they are. A crucial factor to this is the existence of Apollo Client which enables the same interface for client-side state and server-side data.

4.3 Concept: Server

Adjust database schema and structure for multiple customers. Rewrite the server to offer a GraphQL API.

4.3.1 Database Layout

In order to be able to deal with new customers and grow towards a SaaS-application the database structure had to be adapted. So far only one database which holds all data for one customer exists. In order to cater to possible customers that are agencies like Gipfelstürmer itself it is required to allow users to login and then access multiple customers.

The current layout is shown in figure 4.1 and will be adopted to figure 4.2.

Figure 4.1: *Previous database layout (tables simplified for brevity)*

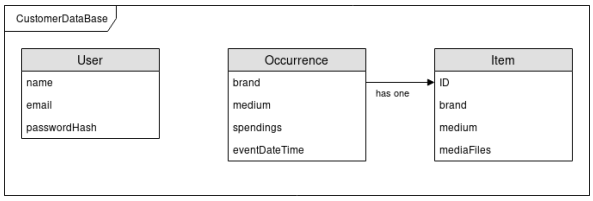
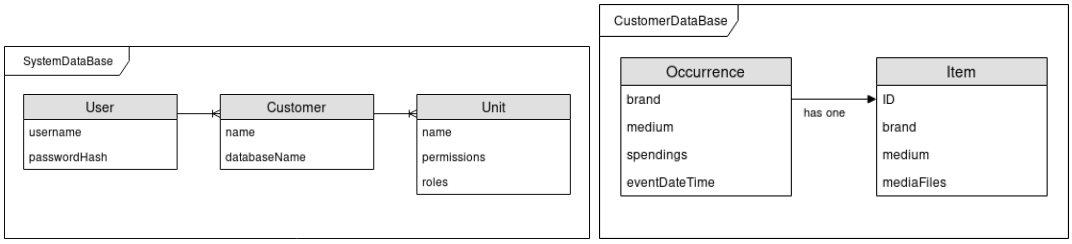


Figure 4.2: *New database layout (tables simplified for brevity)*



As displayed in figure 4.2, there are a few new tables. It is allowed for a user to reference multiple customers, where a user is an actual person and a customer is a company that pays for the service and data. A customer further references multiple units which are meant to be functional elements of our application which can be combined at will to tweak the application for customer needs. Each customer also has its own database which holds their bought set of data, composed of items and occurrences. Items behave as explained in chapter 3.1.1, with there occurrences stripped into a second collection, as a lot of the visualisation results from the time-series described by those and aggregations of those. The extra collection allows for quick and easy access.

Plan

Implement the server with GraphQL Yoga.

- write types
- write resolvers

Challenges

It will be the first time implementing a GraphQL server.

4.4 Concept: Client

4.4.1 Caching

As the data is aggregated on the client side a more effective caching algorithm is required. Time-series data is loaded, which means a list of occurrences arrive on the client side which are included in the timespan specified by the user and sorted in descending order.

The algorithm in place checks if the newly requested timespan is included in the current one. If that is the case, the data is read from the present one. Else a new request is fired.

That algorithm is very wasteful, as it does not care about overlaps in the data which can be used to reduce the amount of data being loaded. Also it only cares about the currently loaded data and just discards anything before that.

That leads to the following potential for improvement:

- keep every result in the cache
- try to read as many data as possible from the cache
- fill eventual missing data in by backend requests

Now a layout of the new algorithm will be given:

An Apollo link (GQL-middleware) will be created on the client-side that cares for the data fetching query. The link remembers which timespans have been queried. The actual caching of the data happens automatically by the apollo-cache-inmemory. First, when a new query is recognised by the link, it checks the previously fetched timespans for overlaps. Secondly it reads the overlaps from the cache and assembles new queries to fill missing parts of the data which are send to the server. Lastly after it read the data from the cache and received data from the requests it assembles the complete result for the initial query and returns that result. The implementation is provided in appendix .1.

4.4.2 Utility

Another required piece of logic is the refetching of interdependent queries. There exist local aggregation queries that build upon the previously mentioned data fetching query. Those local aggregations need to update once new data has been loaded. There is a feature offered by apollo-client for live reload called "watch queries", but these only react to changes to the cached data they returned.

An update is required once the data the query builds upon changes. A half-automatic way of doing this has been designed, called query-registry. It is a combination of a decorator "refetchOn(QUERYNAME)" and an Angular service exposing the method "refetchAllDependendOn(QUERYNAME)". All one needs to do is add a decorator to the variable that holds the queryRef (a type to describe an active query) specifying which query it depends on. The other part is calling the method mentioned above and all dependent queries will be refetched.

Also it is necessary to rewrite the client to use Apollo Client for state-management and data fetching in general.

4.4.3 Challenges

The exact capabilities of apollo-client and apollo-cache-inmemory have to be explored.

Chapter 5

Technology

In this chapter the employed technologies will be shortly introduced, focusing on the parts relevant to the application.

5.1 JavaScript and TypeScript

JavaScript, also known as ECMAScript or JS for short. JS has been designed for running in the browser but due to runtime environments like Node.js it has become viable across the whole stack and can be used as a general purpose programming language. JS is prototype-based, can imitate OOP (object-oriented programming) and can also be used with a very functional programming style. It allows for the programmer to do virtually anything which is great for prototyping but dangerous for production code.

That is where TypeScript or TS comes in. TS is a superset of JavaScript which adds static type checking to the JS language. That is a huge benefit for large-scale application as it vastly reduces the amount of possible bugs due to types and interfaces. TS will be transpiled to JS before running it in production.

TS is the programming language most parts of the application have been written in.

5.2 MongoDB

MongoDB is a document-oriented database. It organizes documents in collections. Collections can have a schema that describes the shape of their contained documents and what is allowed to be inserted, but they do not have to have on. Documents can be arbitrarily complex key-value data, where key and values themself can be documents. They can contain for example but not only common types like Floats, Strings, DateTime; Arrays of those as well as arbitrary binary data. In addition to its great flexibility it provides a really powerful aggregation framework which allows for complex computation and transformations of data right in the database itself.

5.3 GraphQL

GraphQL on its own is just a specification that describes a framework for client-server communication.

5.3.1 Types and Schema

The idea is to look at the application back-end as data packages which are connected. These data packages are described by types. Types have a name and fields. Each field has a type. By the types of the fields, the types are interconnected. They thereby form a graph.

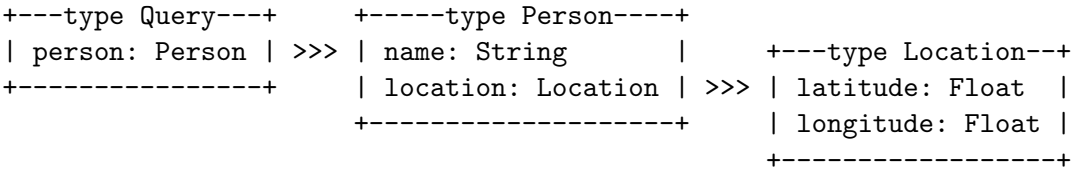


Figure 5.1: *Example of graph by connection of types*

As illustrated in figure 5.1 this connection of types then allows for a traversal of this graph. GraphQL uses special "query"-types which act as entry point into this graph. Formulating this traversal allows for arbitrary combinations of data as long as it is allowed by the structure of the graph. We could now formulate a query like figure 5.2.

```
query {
    person {
        name
        location {
            latitude
            longitude
        }
    }
}
```

Figure 5.2: *Example of query on graph in 5.1 with explanations*

There is no need to make an exhaustive request like this i.e. query all the fields, although queries have to end in so called scalar types. Scalars are everything that is not a custom type nor a query or a mutation, they hold the actual data. Mutations are also entry points, but they are meant to change data not necessarily to retrieve it. The distinction is purely conceptional and not enforced at any point. Fields can be parameterized to allow for more control and flexibility.

All the types together are called schema and fully describe your back end API. With GraphQL there is only one "/graphql" endpoint that your queries are run against.

```
query <-- "entry type"
    person <-- "person type"
        name <-- "string type (scalar/data)"
        location <-- "location type"
            latitude <-- "float type (scalar/data)"
            longitude <-- "float type (scalar/data)"
```

Figure 5.3: *5.2 with type explanations*

5.3.2 Resolvers

To convert your query into data there is some work to do. The response from a GraphQL-endpoint closely mirrors the query as JSON (JavaScript Object Notation) data. The actual data is obtained by so called resolver functions. Every field of the previously defined types is mapped to a resolver function with the same name as the field.

```
1  Query: { // our type
2      person: (parent, arguments, context) => { // our field
3          // return some person from database
4      }
5  }
6  Person: { // our type
7      name: (parent, arguments, context) => { // our field
8          // return parent.name;
9          // OR
10         // return <other service call to get name>;
11         // OR
12         // return "Nino";
13     }
14 }
```

Listing 5.1: Resolver

As shown in figure 5.3.2 there are plenty of options when programming the resolvers. The only necessity is that the returned value is of the correct type. Elsewise we are free to perform arbitrary computations. The option to work with the previously resolved field (via 'parent') exists, evaluation of parameters (via 'arguments') is possible or accessing anything that is setup as a context (via 'context') which could be a database connection or other services.

A response could look like 5.4.

```
{
    "data":{
        "person": {
            "name": "Nino"
        }
    }
}
```

Figure 5.4: *Example query response*

5.4 Angular

Angular2+ is a component based SPA-framework. It is really massive as in it provides solutions and patterns for a wide range of problems out of the box. That really steepens the learning curve a lot.

Components are functional units and the basic building block of Angular. Components span a tree-structure throughout the application. Starting with one root component one is able to nest components at will to arbitrary depth. Components can be purely logical in that sense that they do not have a visual representation or they could exist just to show something to the user. Mostly it is somewhat in between and totally a design choice.

It has no opinion on state management.

5.4.1 RxJS

RxJS is natively integrated with Angular. It provides operators and abstractions for functional and stream-based programming. The core idea is that any data source is not just a single value but a stream of such. For example a button click could be interpreted as a single event that just happened. It also could be seen as one item in a stream of events, as the button might be clicked again. So at the heart of RxJS lies the 'Observable' as an abstract representation of a data stream or source. From this source a transformative pipeline of operations can be defined which will be applied to any item emitted from the source as it makes its way to an 'Observer' which is an abstraction on data sinks or consumers. 'Observers' 'subscribe to' 'Observables' which then communicate in a Publish/Subscribe-fashioned way.

5.4.2 Redux

Redux is a pattern to archive predictable behavior of application state. It consists of three main parts: Actions, Reducers and State.

- State is supposed to be a complete representation of your application and given a particular state it should be able recreate the entire application behavior and UI from that. The state is an immutable object.
- Actions describe changes to the state. Actions can represent user interaction, responses from the server or any other arbitrary event. Actions have a type and a payload (data).
- Reducers are pure functions which evaluate incoming actions and create a new state object, according to the action and its payload.

Every new state is published via an RxJS-Observable, so the parts of the application can listen to state changes that interest them and react to those changes.

It also allows for 'timetravel'-debugging as the entire history of taken actions and the resulting state can be recorded and replayed as desired. More information can be obtained from (5).

5.4.3 Apollo Client

The Apollo Client for Angular provides first and foremost handy abstractions to interact with GraphQL-APIs. It also provides code-generation; as queries are self-descriptive all necessary information to represent those as code of our choosing is given.

Caching

Apollo Client also offers the capability to perform caching for our application, so that identical requests do not have to be requested multiple times. The cache stores data in a normalized form as a flat key-value store. Keys are generated by the GQL-types and provided IDs on the data.

Local State

Build on top of that cache Apollo Client enables us to use that cache as our global application state. We can interface with the cache with direct reads/writes and also define an entire client-side GraphQL schema that lets us query our local data the same way we query remote data. We are also able to query remote and local data in the same query.

Links

We are able to provide middleware, so called 'Links' to Apollo Client. These links allow us to intercept and modify queries on their way to the network and back.

Chapter 6

Implementation

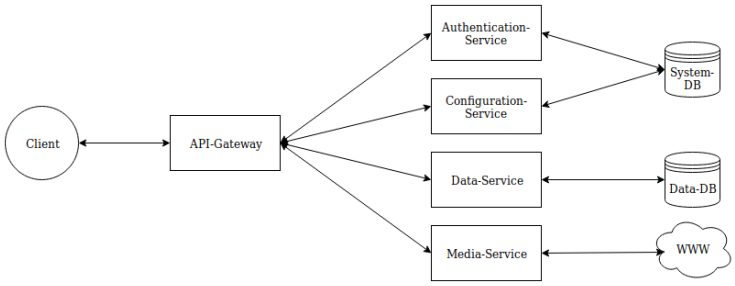
In this chapter some more details regarding the implementation of the proposed solution will be provided as well as unpredicted problems and how they have been solved.

6.1 Server

Regarding our server it took quite a while to develop a solid mental model of GraphQL and to notice some subtle differences between the server- and client-side behavior.

Following a corporate decision the server was to be implemented embedded in a micro-service-like architecture after the initial monolithic prototype. The architecture can be seen in figure 6.1.

Figure 6.1: *Server Architecture*



As an in-depth explanation of the entire architecture seen in figure 6.1 is beyond the scope of this work, only a brief one will be given.

The **Authentication-Service** is the identity provider for the BrandObserver. It allows user to log in and retrieve access token required for further authenticated interaction.

The **Configuration-Service** assembles a user's configuration of our software once he has logged in successfully. It is assembled from multiple sources that provide parts of the entire configuration.

The **Data-Service** provides our application with the actual data to be analysed and displayed.

The **Media-Service** acts as a proxy to format some resources accessed from the WWW that would elsewise be incompatible with the BrandObserver.

The **API-Gateway** redirects request to the appropriate service to handle those. All of the services above expose a GQL-API and their schemata are merged into one by the API-Gateway. This allows for individual development of existing and future functionality.

I did not take part in development of every single service listed.

6.2 Client

It had proven difficult to translate some transformation logic into client-side resolver functions due to some now known limitations in client-side GQL, but it did work in the end. The solution was not very scalable and verbose though. As all the logic necessary was in place we noticed another big problem almost right at the end of this project: The cache implementation of apollo-inmemory-cache was very insufficient in regards to reading and writing huge amounts of data, which the BrandObserver did as mentioned in chapter 3.2. To be more exact about this it sometimes would load more then 400.000 objects from the database. The cache took about 16 seconds to normalize the data and about the same de-normalizing it again when it was read from the cache. This issue was hidden during development due to the way smaller data set in use for the prototype.

To tackle this problem it has been concluded after several different approaches that it would be necessary to move the aggregation of the data to the server. Tests showed that is was very slow to read that many data from the database, which was another bottleneck (after the client-side cache) as it was transferring up to 100 MB in size. Being that wasteful with the clients bandwidth, memory and time could not be tolerated. It was possible to leverage the aggregation framework of MongoDB in combination with GraphQL to create a working solution in just two days.

New types had been defined which would represent the necessary aggregated data the Angular components would want to have. Now server-side resolvers for the previously defined types are backed by fast aggregations which brought the response-time on below one second consistently, while increasing our response time on average from around 30ms (client side transformation) to 200ms (network request and database aggregation). This is due to the change in strategy as we do not preload all raw data which after an initial delay served further transformations alomst instantly as they were computed on the client without the network involved. Now we have latency whenever the user visits a specific view of our data for the first time in his session as a network request is made. Same requests in the same session will be served from the cache. In general the application is more respectful in regards to the clients computing capacity, bandwidth and time, as pre-aggregated datasets are much smaller and thereby served more quickly.

Chapter 7

Conclusions

In retrospective I'd like say that I have learned a lot. We established a future proof foundation of our application that renders a clear framework on how to extend it. The decoupling of the data and the component hierarchy is a huge improvement. Further the use of GQL as a client and server side interface unifies the mental framework of how to query and modify data.

I also learned that I would have liked to do more extensive testing of the tools that are to be implemented. Only after we had already commited to the tools, I learned about some limitations of them. apollo-client for example is still under active development and while it can be used for production code it still lacked some useful functionality or features that could be considered essential. However there is a helpful and active community that provides workarounds and support.

Further I can say I would like to take a way more structured approach to upcoming projects. A more distinct articulation of the desired outcome as well as specified measurements for the set goals that are checked on a regular basis will help to keep the project on track. They will make it much more tangible if the current actions are in line with the bigger goal. This will also allow for a more detailed analysis of how successful the project was in retrospective.

Another problem was to incorporate all the requirements that came up once the prototype was supposed to be upgraded to the next version of the product. Some tasks came up that were not related to the initial set of goals of the practical phase which diverged my work from those set goals. The direction however stayed the same.

Finally I would like to thank Prof. Dr. Randerath for allowing me to work on this project with such freedom and Gipfelstürmer Kommunikation for the opportunity to try myself at this project.

Bibliography

[1] "Graphql introduction." [Online]. Available: https://graphql.org/learn/

[2] "Apollo client angular docs." [Online]. Available: https://www.apollographql.com/docs/angular/

[3] "Graphql yoga." [Online]. Available: https://github.com/prisma/graphql-yoga

[4] "Graphql code generator." [Online]. Available: https://graphql-code-generator.com/

[5] "Redux for angular." [Online]. Available: https://ngrx.io/

Appendices

.1 Caching algorithm

The code written for the initial client side cache logic.

```
import {
  ApolloLink,
  NextLink,
  Operation,
  Observable
} from 'apollo-link';
import {
  isWithinRange,
  areRangesOverlapping,
  subMilliseconds,
  addMilliseconds,
  differenceInDays,
  endOfDay,
  startOfDay,
  isAfter, isBefore,
} from 'date-fns';
import { InMemoryCache } from 'apollo-cache-inmemory';

/**
 * Class to keep track of the timespan covered by the cache.
 */
export class TimeSpan {
  public readonly start: Date;
  public readonly end: Date;
  public isCached: boolean;
  public includedIn: TimeSpan;

  public constructor(params: TimeSpanParams) {
    this.isCached = params.isCached || false;
    this.includedIn = params.includedIn || null;
    this.start = startOfDay(params.start);
    this.end = endOfDay(params.end);
  }

  public equals(other: TimeSpan): boolean {
    return this.start.getTime() === other.start.getTime() && this.end.getTime
        () === other.end.getTime();
  }
}

/**
 * Interface to provide an accumulator function for the specific
     implementation of the link.
 */
export interface ResultAccumulator<T> {
  reducer: (accumulator: T, current: any) => T;
  initialAccumulator: T;
}

/**
 * Interface to make sure results comply with the structure that apollo uses.
 */
```

```
51  export interface CompleteResult<T> {
52    data: T;
53  }
54
55  /**
56   * Interface to describe parameters for constructing a timespan.
57   */
58  export interface TimeSpanParams {
59    start: Date;
60    end: Date;
61    isCached?: boolean;
62    includedIn?: TimeSpan;
63  }
64
65  /**
66   * Interface to pass important context around in the link.
67   */
68  interface ResolvingContext {
69    cache: InMemoryCache;
70    operation: Operation;
71    forward: NextLink;
72    queriedTimeSpan: TimeSpan;
73  }
74
75
76  export abstract class TimeSpanPatcherLink<Q, R> extends ApolloLink {
77    // sorted by sum of days in timeSpan
78    private readonly cachedTimeSpans: Array<TimeSpan> = [];
79
80    protected constructor(
81      private readonly nameOfOperationToCache: string,
82      private timeSpanFactory: ((params: TimeSpanParams) => TimeSpan) = (params
          : TimeSpanParams) => new TimeSpan(params)
83    ) {
84      super();
85    }
86
87    public static isRangeIncludedInRange(toIncludeStart: Date, toIncludeEnd:
        Date, includedIntoStart: Date, includedIntoEnd: Date): boolean {
88      return isWithinRange(toIncludeStart, includedIntoStart, includedIntoEnd)
89        && isWithinRange(toIncludeEnd, includedIntoStart, includedIntoEnd);
90    }
91
92    /**
93     * This is a naive algorithm to reduce time series data T1 to time series
          data of T2.
94     *
95     * T1 is only represented through its data.
96     * T2 is only represented through its start/end date.
97     *
98     * T2 could be inside T1 like so:
99     * |<---T1---|<----T2-inside-->|-->|
100    *
101    * T2 could overlap with T1 like so:
102    * |<---T1--|<--overlap-->|----T2->|
103    *
```

```
104     * T2 could not overlap like so:
105     * |<---T1-->| no overlap |<--T2-->|
106     *
107     * The algorithm works by starting at the edges of T1 and 'walks' inwards
          until
108     * it steps over the border to T2 and stop there.
109     * The time series data is then cropped between the two found borders and
          returned.
110     *
111     * @param cachedInstantOfTimeData: array of time series data of a known
          timespan
112     * @param toTimeSpan: the timespan the param above shall be reduced to
113     * @param keyToInstantOfTime: the key of where to find the dateTime of time
          series data (assumes flat object)
114     */
115    public static naiveCrop(cachedInstantOfTimeData: Array<any>, toTimeSpan:
          TimeSpan, keyToInstantOfTime: string): Array<any> {
116      if (cachedInstantOfTimeData.length === 0) return [];
117
118      let low = 0;
119      while (isBefore(cachedInstantOfTimeData[low][keyToInstantOfTime],
          toTimeSpan.start)) {
120        if (low === cachedInstantOfTimeData.length - 1) return [];
121        low += 1;
122      }
123
124      let high = cachedInstantOfTimeData.length - 1;
125      while (isAfter(cachedInstantOfTimeData[high][keyToInstantOfTime],
          toTimeSpan.end)) {
126        if (high === 0) return [];
127        high -= 1;
128      }
129
130      return cachedInstantOfTimeData.slice(low, high + 1);
131    }
132
133    /**
134     * This algorithm improves on the previous one by incorporating a technique
          called 'galloping'.
135     * It takes incrementally larger steps towards the border until it steps
          too far.
136     * Then it starts with small steps again right before it stepped too far.
137     *
138     * @param cachedInstantOfTimeData: array of time series data of a known
          timespan
139     * @param toTimeSpan: the timespan the param above shall be reduced to
140     * @param keyToInstantOfTime: the key of where to find the dateTime of time
          series data (assumes flat object)
141     */
142    public static gallopingCrop(cachedInstantOfTimeData: Array<any>, toTimeSpan
          : TimeSpan, keyToInstantOfTime: string): Array<any> {
143      if (cachedInstantOfTimeData.length === 0) return [];
144
145      let low = 0;
146      let step = 1;
147
```

```
148     while (isBefore(cachedInstantOfTimeData[low][keyToInstantOfTime],
            toTimeSpan.start)) {
149
150       if (low === cachedInstantOfTimeData.length - 1) return [];
151       low += step;
152
153       if ((low > cachedInstantOfTimeData.length - 1) || !isBefore(
              cachedInstantOfTimeData[low][keyToInstantOfTime], toTimeSpan.start))
               {
154         if (step === 1) break;
155         low -= step;
156         step = 1;
157       } else {
158         step = step * 2;
159       }
160     }
161
162     let high = cachedInstantOfTimeData.length - 1;
163     step = 1;
164
165     while (isAfter(cachedInstantOfTimeData[high][keyToInstantOfTime],
            toTimeSpan.end)) {
166
167       if (high === 0) return [];
168       high -= step;
169
170       if ((high < 0) || !isAfter(cachedInstantOfTimeData[high][
              keyToInstantOfTime], toTimeSpan.end)) {
171         if (step === 1) break;
172         high += step;
173         step = 1;
174       } else {
175         step = step * 2;
176       }
177     }
178
179     return cachedInstantOfTimeData.slice(low, high + 1);
180   }
181
182   /**
183    * This method is called by the apollo link API when it advances queries
           through the link stack.
184    *
185    * @param operation: query that is in flight
186    * @param forward: next link
187    */
188   public request(operation: Operation, forward: NextLink): Observable<any> {
189     if (operation.operationName !== this.nameOfOperationToCache) {
190       return forward(operation);
191     }
192
193     console.log('inside TSP link');
194     console.log(operation);
195     console.log(operation.getContext());
196
197     const { cache, reset } = operation.getContext();
```

```
198
199     if (reset) {
200       this.cachedTimeSpans.length = 0;
201
202       const data = {
203         shouldReset: {
204           __typename: 'Reset',
205           id: 'RESET',
206           reset: false
207         }
208       };
209       cache.writeData({ data });
210     }
211
212     // TODO: implement 'reset' variable in query
213     const { from, to } = operation.variables;
214     const resolvingContext: ResolvingContext = {
215       cache,
216       operation,
217       forward,
218       queriedTimeSpan: this.timeSpanFactory({ start: from, end: to })
219     };
220
221     return this.constructResponseFromCacheAndNetwork(resolvingContext);
222   }
223
224   private constructResponseFromCacheAndNetwork(context: ResolvingContext):
        Observable<CompleteResult<Q>> {
225     console.log('build response from cache called');
226
227     const matched: Array<TimeSpan> = this.findOverlappingCachedTimeSpans(
          context.queriedTimeSpan);
228
229     if (matched.length === 1 && matched[0].includedIn) {
230       console.log('found fully cached');
231       return this.resolveTimeSpansToData(matched, context);
232     } else {
233       const stitched: Array<TimeSpan> = this.stitchPartialTimeSpans(context.
            queriedTimeSpan, matched);
234       return this.resolveTimeSpansToData(stitched, context);
235     }
236   }
237
238   private findOverlappingCachedTimeSpans(queryInput: TimeSpan): Array<
        TimeSpan> {
239     console.log('find overlaps called');
240
241     const inputFrom: Date = queryInput.start;
242     const inputTo: Date = queryInput.end;
243
244     const overlaps: Array<TimeSpan> = [];
245
246     // find (partially) cached data
247     this.cachedTimeSpans.some((cachedTimeSpan: TimeSpan) => {
248       const cachedFrom: Date = cachedTimeSpan.start;
249       const cachedTo: Date = cachedTimeSpan.end;
```

```
250
251       if (TimeSpanPatcherLink.isRangeIncludedInRange(inputFrom, inputTo,
              cachedFrom, cachedTo)) {
252         queryInput.includedIn = cachedTimeSpan;
253         queryInput.isCached = true;
254         overlaps.push(queryInput);
255         return true;
256       } else {
257         if (areRangesOverlapping(inputFrom, inputTo, cachedFrom, cachedTo)) {
258           overlaps.push(cachedTimeSpan);
259         }
260       }
261     });
262
263     // returns timeSpans overlapping with query ordered by start of timeSpan
264     return overlaps.sort((a: TimeSpan, b: TimeSpan) => a.start < b.start ? -1
            : 1);
265   }
266
267   private stitchPartialTimeSpans(queryInput: TimeSpan, partialCached: Array<
        TimeSpan>): Array<TimeSpan> {
268     console.log('stitched timeSpans called');
269     // partial must be ordered by from date (they are)
270     let dateIndex: Date = queryInput.start;
271     const timeSpansToResolve: Array<TimeSpan> = [];
272
273     partialCached.forEach((cachedTimeSpan: TimeSpan) => {
274       // if gap between cached timeSpans
275       if (dateIndex < cachedTimeSpan.start) {
276         // missing timeSpan
277         timeSpansToResolve.push(
278           this.timeSpanFactory(
279             {
280               start: dateIndex,
281               end: subMilliseconds(cachedTimeSpan.start, 1),
282               isCached: false
283             }
284           )
285         );
286         // cached timeSpan not too big
287         if (cachedTimeSpan.end <= queryInput.end) {
288           cachedTimeSpan.isCached = true;
289           timeSpansToResolve.push(cachedTimeSpan);
290           dateIndex = addMilliseconds(cachedTimeSpan.end, 1);
291         } else {
292           timeSpansToResolve.push(
293             this.timeSpanFactory(
294               {
295                 start: cachedTimeSpan.start,
296                 end: queryInput.end,
297                 isCached: true,
298                 includedIn: cachedTimeSpan
299               }
300             )
301           );
302           dateIndex = addMilliseconds(queryInput.end, 1);
```

```
303         }
304       } else if (isWithinRange(dateIndex, cachedTimeSpan.start,
              cachedTimeSpan.end)) {
305         // fully or partially useful cached timeSpan
306         timeSpansToResolve.push(
307           this.timeSpanFactory(
308             {
309               start: dateIndex,
310               end: cachedTimeSpan.end,
311               isCached: true,
312               includedIn: cachedTimeSpan
313             }
314           )
315         );
316         dateIndex = addMilliseconds(cachedTimeSpan.end, 1);
317       }
318     });
319
320     // if still some data missing
321     if (dateIndex < queryInput.end) {
322       // missing timeSpan
323       timeSpansToResolve.push(
324         this.timeSpanFactory(
325           {
326             start: dateIndex,
327             end: queryInput.end,
328             isCached: false
329           }
330         )
331       );
332     }
333
334     return timeSpansToResolve;
335   }
336
337   private resolveTimeSpansToData(matchedTimeSpans: Array<TimeSpan>, context:
          ResolvingContext): Observable<CompleteResult<Q>> {
338     console.log('resolve to data called');
339
340     let chainDataObservable: Observable<R> = new Observable<R>(observer => {
341       observer.complete();
342       return () => {};
343     });
344
345     matchedTimeSpans.forEach((matchedTimeSpan: TimeSpan) => {
346       if (matchedTimeSpan.isCached) {
347         console.log('is cached');
348         if (matchedTimeSpan.includedIn) {
349           console.log('is included in');
350           // read cached data
351           const cachedQuery: Q = context.cache.readQuery({
352             query: context.operation.query,
353             variables: {
354               from: matchedTimeSpan.includedIn.start,
355               to: matchedTimeSpan.includedIn.end
356             }
```

```
357           });
358
359           // find required data from cached query and emit through zen-
                observable
360           // @ts-ignore: TS somehow does not recognise Observable-Type
361           chainDataObservable = chainDataObservable.concat(
362             new Observable<R>(observer => {
363               observer.next(
364                 this.cropTimeSpanData(
365                   this.mapQueryResultToData(cachedQuery),
366                   matchedTimeSpan
367                 )
368               );
369               observer.complete();
370               return () => {};
371             })
372           );
373         } else {
374           // read cached data
375           const cachedQuery: Q = context.cache.readQuery({
376             query: context.operation.query,
377             variables: {
378               from: matchedTimeSpan.start,
379               to: matchedTimeSpan.end
380             }
381           });
382
383           // emit all data from cached query through zen-observable
384           // @ts-ignore: TS somehow does not recognise Observable-Type
385           chainDataObservable = chainDataObservable.concat(
386             new Observable<R>(observer => {
387               observer.next(this.mapQueryResultToData(cachedQuery));
388               observer.complete();
389               return () => {};
390             })
391           );
392         }
393       } else { // data not cached, needs network request
394         // modify original request
395         context.operation.variables = { from: matchedTimeSpan.start, to:
              matchedTimeSpan.end };
396
397         console.log('operation in resolve: ', context.operation);
398
399         // pass altered query on to next link in stack and concat result as
              zen-observable
400         // @ts-ignore: TS somehow does not recognise Observable-Type
401         chainDataObservable = chainDataObservable.concat(
402           context.forward(context.operation).map(result => {
403             // TODO: could return error?
404             return this.mapQueryResultToData(result.data);
405           })
406         );
407       }
408     });
409
```

```
410     const { reducer, initialAccumulator } = this.accumulateQueryResults();
411
412     return chainDataObservable
413       .reduce(reducer, initialAccumulator)
414       .map(this.postProcessResult)
415       .map(result => {
416         this.addTimeSpanToCache(context.queriedTimeSpan);
417         return result;
418       });
419   }
420
421   private addTimeSpanToCache(toAdd: TimeSpan): Array<TimeSpan> {
422     this.cachedTimeSpans.push(toAdd);
423     // sort by biggest timespan first
424     this.cachedTimeSpans.sort((a: TimeSpan, b: TimeSpan) => differenceInDays(
          a.start, a.end) - differenceInDays(b.start, b.end));
425     return this.cachedTimeSpans;
426   }
427
428   public abstract accumulateQueryResults(): ResultAccumulator<R>;
429
430   public abstract postProcessResult(value: R): CompleteResult<Q>;
431
432   public abstract mapQueryResultToData(queryResult: Q): R;
433
434   public abstract cropTimeSpanData(data: R, timespan: TimeSpan): R;
435 }
```