Hochschule Kempten
University of Applied Sciences

Student Research Project

Computer Science - Game Engineering (Bachelor)

Development of a tool for the graphical representation of software projects

Daniel Wildegger

Supervisor	R. Alden
Advisor	Prof. Dr. M. Lenke
Submission Date	1. April 2017
Realized in the	Faculty of Computer Science

Abstract

In this paper we will look at the various tools that were used to develop a tool as mentioned above, in the context of a web application. To begin with, we will give a quick introduction to the topic and also talk about the purpose of this project. In the second chapter we will go from the backend to the frontend and take a closer look at each of the tools. The third chapter will focus on the technologies used to make the application publicly available through a public domain. The fourth chapter will talk about the actual development, as in why the before discussed technologies were chosen, lessons learned during the process and problems faced together with their solutions. After this, there will be a short chapter about if, looking back, the chosen technologies where a good decision and why they might not have been. The paper will end with a documentation of the codebase, together with a list of suggestions on what could be added to/improved in the application in further development.

Contents

1 Introduction

There are many tools on the internet for building graphs to visualize data. Famous examples are "ConceptDraw Pro", "Lucidchart" [Par20] or "draw.io". However when using these the user spends a lot of time on creating a nice looking diagram, centering important components etc., to get a pleasant looking result in the end or they are bound to displaying hardware components.

The idea for this project is to offer an online editor with CRUD-functionality for a network, consisting of components of software projects, as well as a first implementation of an algorithm that will, in most cases, create a nice looking layout on its own.

1.1 Purpose

The purpose is to try out and test a combination of technologies. This software, or variations of it, might later be incorporated into a bigger project or adapted to fit another use case. The experiences and impressions during development can be of help when thinking about what technology stack to use, which is why a big part of this paper is dedicated to documenting this process.

2 Backend to Frontend

In this chapter we will introduce the individual components, tools and frameworks this application is built with. For some of them we will give some more insights on how they work internally, the others are big enough on their own. The order will be the same as they appeared in the development process.

2.1 The GRAND-stack

GRAND stands in this case for **G**raphQL, **R**eact, **A**pollo and **N**eo4j **D**atabase. [LGK20] React is a frontend framework, Apollo is used for statemanagement on the client side and communicating to the database on the server side. GraphQL will be used for fetching and mutating data. Neo4j is a graph database and the server will communicate with it through a JavaScript driver provided by the Neo4j community. More details can be found in the Development and Documentation section.

2.2 Query Language - GraphQL

GraphQL is a data query language as well as specification. Its development was started by Facebook in 2012 and it was open sourced in 2015. [Fou20]

After their application suffered from poor performance on mobile devices, they took a new implementation using natively implemented models and views. This required a new API for their news feed as it was previously delivered as pure HTML.
After evaluating different common options like RESTful-APIs and FQL they often saw the same problems: The ratio of data actually used compared to the one fetched was very small, the number of requests [Dev19] and the amount of code on both server and client side to prepare the data was big. [Byr15]
For example, for loading the start page of a single user, there would have been a lot of different requests necessary:

- *https://facebook.com/user/id* - Get all user specific data
- *https://facebook.com/user/id/events* - Get all possibly relevant events
- *https://facebook.com/user/id/friends-suggestions* - Get all friend suggestions
- ...

[Dev19]

GraphQL aims to resolve all these issues: Reduce the amount of unnecessary data transferred, reduce the number of requests and increase the developer productivity by making it easier to use fetched data. [Byr15]

It allows developers to get a lot of different data from a single endpoint. This means that instead the above shown 3+ endpoints, when using GraphQL all requests would go to *graph.facebook.com*, with a query similar to:
(todo: define graphql styles)

Listing 2.1: A GraphQL Query

```
query {
   user(id: 1) {
      name
      events {
         count
      }
      friends_suggestions {
         name
         mutual_friends {
            count
         }
      }
   }
}
```

[Dev19, with adaptions] Where the answer would be a JSON-string: (todo: define JSON style)

Listing 2.2: Example Response Data

```
{
   "data": {
      "user": {
         "name": "Brandon Minnick",
         "events": {
            "count": 4
         },
         "friends_suggestions": {
            "name": "Seth Juarez",
            "mutual_friends": {
               "count": 18
            }
         }
      }
   }
}
```

[Dev19, with adaptions]

The query can be as extensive as the developer needs it, it will return only the data requested and the answer string can be directly accessed like a JSON-object. By that GraphQL fulfills all its design goals.

The previously shown query then needs to be resolved by a server thats able to interpret GraphQL and resolve the query. All non primitive data types have to be defined following the GraphQL specification. An example for a user schema might be:

```
type User {
   id: ID!
   name: String!
   events: [Event]
   friends: [Friend]
```

```
   friends_suggestions: [Friend_Suggestion]
}
```

where "Event", "Friend" and "Friend_Suggestion" themselves are other types described in a similar manner.

By putting a "!" behind a property the programmer marks it as required, meaning it can never be null or empty. The square brackets define that the property is a list of the type they surround.
To be able to run queries in the first place, we must first define a root type for all queries:

```
schema {
   query: Query
}
```

In this root type all possible queries must be described:

```
type Query {
   user(id: ID!): User
}
```

Here we describe a query that can be executed as shown in **2.1** by providing an ID to the query and the correct query name, together with a collection set telling the server which fields to fetch. In the parentheses any query arguments are listed, in this example id must be provided. After the double dot the return type is named.
The server will then make requests to the DB, fetch the requested data and return it to the user once all fields were filled with values.
For further information about the extensive type system please see the official GraphQL specification. [Fac18]

2.3 Database - Neo4j

General

Neo4j is a so called graph database. The idea of graph databases is, compared to traditional relational databases, a young concept and differs in a few concepts. At the moment Neo4j is the 22nd most popular database overall [DbE20a] and the most popular graph database. [DbE20b]

- Unlike most relational databases, who store data through tables and joins, Neo4j stores data in the form of actual nodes and relationships between such [Neod]. In other DBMS relations between items generally are achieved through join-/lookup-tables which have to be generated. [Neob]
- When running a query on a relation DB the server will run through a table and when it finds the searched item it might look for the ID of a related item and start indexing again. With a graph DB the server will index [Lin12a, minute 32] once to find the initial node and can then directly access all connected items as they are stored through their relation with the current one. [Neo20] These are stored as memory pointers which makes following them extremely efficient.

- Neo4j uses Cypher as query language. The Cypher syntax was supposed to visually represent the shape of the data a user wants to retrieve instead of describing how to get data, as well as offer the power and functionality other languages offer. [Neoc]

Cypher

For matching all nodes connected to node A through a "Neighbor" relationship, we simply state

Example 2.3.1 MATCH (n:Node {label: "A"})-[:Neighbor]-(n2:Node) RETURN n, n2

Parentheses represent a node, square brackets a relationship. The naming works after the following pattern: <name>:<type>. In this example n is the name for the first node and n2 the name of the list of connected nodes. We didn't specify a name for the relationship as we did not want to retrieve data from it. By using curly braces we can specify certain properties a node or relationship should have. The other way of doing so would be

Example 2.3.2 MATCH (n:Node)-[:Neighbor]-(n2:Node) WHERE n.label = "A" RETURN n, n2

which might look a bit cleaner. We could also return only specific values of n and n2 and give them names by stating
...RETURN n.label AS Label1, n2.label AS Label2
The following information about Neo4j internals is all from [Lin12a] and [Lin12b]. Sadly, these sources are all old and probably outdated, yet there does not seem to be more updated information on the internet.

The Graph on Disk

Internally, there are 3 types of records saved on the disk: node-, relationship- and property-records. All of these have fixed sizes to allow for quicker allocation during the start up process. Every record has an "inUse" field, as well as a unique ID with which Neo4j is able to exactly locate a searched record on the disk. [Lin12a, minute 08]
Properties on nodes are saved through a linked list like object. The exact implementation however does not alter the idea behind it. A property knows about its type and has a next pointer. Each node saves the pointer to its first property whose next pointer will lead to the next property etc.. Should a next pointer be empty the algorithm knows that it has reached all properties of a node.
In addition to the first property, each node knows about its first relationship. If a it is the *first* one, is simply being determined by the order of creation. A relationship has pointers to its start- and end-node, to its type and four others to other relationships, which are best explained in an example traversal in pseudo code:

```
1 if node n has relationship pointer r:
2   if n is start node of r:
3     if r has StartNext pointer sn:
4       set r = sn
5       repeat from line 2
6     endif
7     else
8       visited all relationships $ \rightarrow $ terminate
```

```
      endelse
   endif
   else
      if r has EndNext pointer en:
         set r = en
         repeat from line 2
      endif
      else
         visited all relationships $ \rightarrow $ terminate
      endelse
   endelse
endif
```

We see that every relationship has two next pointers. Which one will be used for further traversal, depends on if the source node is start- or end-node in the current relationship.
In addition to this, the same pointers exist into the other direction, meaning that there are also two pointers called StartPrevious and EndPrevious. The question for selecting which one will be chosen for further iteration stays the same.

The Graph in Memory

Upon start up these records are being loaded into the "FS Cache" (File System Cache). Neo4j will then partition these into equally sized regions and create a hit counter for each of them, to encounter high traffic regions that will be loaded into the "Node/Relationship Object Cache" which is more similar to an actual graph.
Here each node holds a list of relationships that are grouped by the relationship type to allow for quick traversals, and relationships only hold their properties as well as start- and end-node and their type. Any references to other records are being done by its ID.

Traversing

For finding a node to start traversing the graph, Neo4j uses traditional indexing. [Lin12a, minute 32] Once the start node is found, 2 concepts take over:

1. **RelationshipExpanders** which will for a node return all relevant relationships to continue traversing from that node

2. **Evaluators** which return if traversing should continue on this branch ($\rightarrow$ expand) or not and if this node should be included in the result set or not.

When accessing a node the first thing the system will try to do is fetch it from the cache. If it shouldn't be there, the next place that will be checked is the FS Cache. Should the region that contains the node be apparent here, the access is quick but blocking, meaning that the entire region is getting locked. In the case that the region is out of the FS cache the operation is blocking and slower.
The locking is necessary to make sure that no other transaction will evict that area from the memory while the current one reads the data.

Adding Cypher

As Cypher describes the shape of the searched data, a searched query will be converted into a representative pattern graph that matches the searched structure.

When a query is run, the first thing that happens is that matching start-nodes are searched in the database (through indexing). When a node is found, traversing the database starts as described above. For Expanders and Evaluators to know what to return, they simply compare the pattern graph described through Cypher with the graph that was found so far and see if there is more data that matches.

2.4 Server - ApolloServer

Apollo Server is a spec-compliant GraphQL server. It can be embedded into Node.js middleware like Express or Fastify [Graa] and will listen for connections on a defined port.
When it receives one it will read the query and call the respective route, or resolvers as they are called.
In addition to that the server will deliver, together with some more, a *context* object to each route that contains a driver which connects to a database, which is Neo4j in our case. Using this object together with a specified Cypher query we can manipulate the DB.

Example Resolver

A resolver to create a node might look like the following:

Listing 2.3: A Basic Resolver

```
1 async CreateNode( _, args, ctx ) {
2   const session = ctx.driver.session();
3   const query = `
4         CREATE (n:Node:${ args.nodeType } {id:$id, label:$label, nodeType:
              $nodeType})
5         SET n += $props
6         RETURN n`;
7   const results = await session.run(query, args);
8   return results.records.map(record => record.get('n').properties)[0];
9 }
```

Lets break down whats happening in this piece of code:

- Line 1 contains the function definition. "args" is an object that contains all data sent with the query from the frontend. "ctx" is the context object that contains the neo4j driver to communicate with the DB. The first argument "_", which is a placeholder here as we do not need it, is the so called "parent" which is equal to the previous resolver in the resolver chain. (More about this later REMOVE THIS COMMENT)
- In line 2 we acquire a session to communicate with the database. [Neoa] Over this object we can send parameters that get executed right away.
- Lines 3 to 6 define a Cypher query which is similar to the ones shown in **{2.3.1}** and **{2.3.2}**.
- In line 4 we make use of the args object and embed the nodeType directly into the query string by using template strings. This is necessary because at this position of a cypher query we can't make use of query variables the same way we do in the rest of the query. Later in that line we can see that by using $<variableName> we can access query variables we pass along.

- Line 5 demonstrates the usage of an object we can pass as query variable. This object can't only contain simple datatypes, but its really useful to set various values at once.
- Finally, in line 7 we send the specified query string together with the args object (that must contain all referenced variables) to the database. By using the ES6 await keyword we make sure that code execution doesn't continue until the results are returned.
- In the last line we iterate over the record set and retrieve any properties by the in the query specified name. Using only the first element of the array is specific to this case, as CreateNode is defined to return a single node, not an array of such.

Resolver Chain

To explain the resolver chain we will take a look at the following example GraphQL query:

```
query GetBooksByLibrary {
   libraries {
      branch
      books {
         title
         author {
            name
         }
      }
   }
}
```

which will be executed on this schema

Listing 2.4: Schema Definition

```
# A library has a branch and books
type Library {
   branch: String!
   books: [Book!]
}

# A book has a title and author
type Book {
   title: String!
   author: Author!
   branch: String!
}

# An author has a name
type Author {
   name: String!
}

type Query {
   libraries: [Library]
}
```

To resolve the query we need 4 resolvers:

- A root resolver which defines the entry point for the query
- One resolver each for "Library", "Book" and "Author"

Assuming we have static arrays called "libraries", "books" and "authors" that are filled with data, the resolvers might look like the following:

Listing 2.5: Resolver Definition

```
const resolvers = {
   Query: {
      libraries() {
         return libraries;
      }
   },
   Library: {
      branch(parent) {
         return parent.branch;
      },
      books(parent) {
         return books.filter(book => book.branch === parent.branch);
      }
   },
   Book: {
      title(parent) {
         return parent.title;
      },
      author(parent) {
         return authors.find(author => author.name === parent.author.name);
      }
   },
   Author: {
      name(parent) {
         return parent.name;
      }
   }
};
```

First, the Query resolver is hit and it will search for a defined key that is similar to the name mentioned in the highest level of the query object in **2.3**, in this case "libraries". In the GraphQL schema under **2.4** we defined that this query will return an array of Library objects.
Knowing this, the server will now go through each object of this array and look for resolvers of the in the query specified fields. This object is passed as *parent* into the next resolver in the resolver chain.
For each library we want the branch and an array of books. As branch is a primitive type it does not need to be further resolved. Books however, returns an array of non-primitive types. To find out which books we need to return we can access the value *parent.branch* and compare it to the branch of each book in the books array and return those who match.
books is again an array of a non primitive type and has to be further resolved by iterating

through the array and accessing the requested values title and author. Title is just a string, whereas author will get resolved further etc.

Cypher in GraphQL

Using GraphQL directives we can "annotate" our schema and specify precisely certain actions or checks the server should perform when accessing a field.
We could create the following schema:

Listing 2.6: Example Directive Declaration

```
directive @deprecated(
   reason: String = "No longer supported"
) on FIELD_DEFINITION | ENUM_VALUE

type ExampleType {
   newField: String
   oldField: String @deprecated(reason: "Use ‘newField‘.")
}
```

Directives can be distinguished by the @-symbol and are placed after a field definition to annotate one. When querying *oldField* on *ExampleType* the server might only respond with "Use 'newField'" and not send any data. The exact behavior depends on how directive behavior is defined in the server.
The use cases range from formatting strings, enforcing access permissions to value checking when the client sends data and many more. For information about how directives can be implemented please refer to
In the GRAND-stack we can use a pre-defined directive called "@cypher" and through that use cypher statements directly in the schema definition file. A great and short example is getting all connected nodes for a specific node:

Listing 2.7: Cypher in GraphQL

```
type Node {
   ...
   connectedTo: [Node] @cypher(statement: "MATCH (this)--(:Link)--(n:Node)
      return n")
   ...
}
```

The node that is currently being iterated over in the resolver chain is passed as *this* to Neo4j. Then it'll look for other nodes that are connected through any relationship of type *Link* and return these. In addition to this, ApolloServer can generate default resolvers for queries and mutations meaning we do not have to write a resolver for *Node* on our own. This combination makes writing query resolvers a rare occasion when using the GRAND-stack.

Please note that this feature is only available when *APOC* is enabled on this Neo4j instance (more information about this in the end of chapter 3 and chapter 4).

2.5 Frontend - React

React is a JavaScript-Framework to create component-based user interfaces. Each component has to define a *render* method which describes what appears on the screen. In this method the programmer writes basic HTML or can embed other react components.
The standard *index.html* is pretty short when using react. The only code a programmer writes there is normally in the header area to include CDNs or other resources. The body contains only one element:
* id="root"> *

Components

In *index.js* this root div will be referenced by the react-internal render method and recursively build the basic html out of the defined react components:

Listing 2.8: Hello World in React

```
// index.html:
<!DOCTYPE html>
<html>
<head>
   <title>Intro-App</title>
</head>
<body>
   <div id="root"></div>
</body>
</html>

// index.js:
ReactDOM.render(
   <App/>,
   document.getElementById( 'root' ),
);
// App.js:
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            Hello World
         </div>
      );
   }
}

export default App;
```

We define a class component by extending from *React.Component*. Each class component must at least have a *render()* method. React will use the return values of these methods to build the DOM.

When we start the react app, open it in the browser and select inspect we will see the following in the "Elements" tab (ignoring a script for live updates in the head):

```
<html>
<head>
   <title>Intro-App</title>
</head>
<body>
   <div id="root">
      <div>
         Hello World!
      </div>
   </div>
</body>
</html>
```

React starts traversing at whatever component is put into the *ReactDOM.render* method and repeats the process for each component until primitive html elements that can be rendered directly are reached.

Every react class components can receive values from its parent element, by passing them like normal html properties (e.g. line 3 below). In the child they can be accessed through a variable called *props* and using JSX we can embed the value of variables directly in the component (e.g. line 14 below):

Listing 2.9: Using Props

```
// index.js:
ReactDOM.render(
   <App text={ 'Hello World' },
   document.getElementById( 'root' ),
);

// App.js:
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            { this.props.text }
         </div>
      );
   }
}
export default App;
```

Both **2.8** and **2.9** will produce the exact same output.

State

React class components have a state which can be used to manage user actions on a component, as well as general application data. *state* is a simple JavaScript object but should be

treated as immutable and only be updated through the *setState()* method. Modifying the state directly can lead to bugs and/or unexpected behavior of the application.

To demonstrate this, we'll add a *Counter* component to the application:

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(
   <App text={ 'Hello World' }/>,
   document.getElementById( 'root' ),
);

// App.js
class App extends React.Component {
   render() {
      return (
         <div>
            { this.props.text }
            <Counter/>
         </div>
      );
   }
}
export default App;

// Counter.js
import React from 'react';

class Counter extends React.Component {
   constructor( props ) {
      super( props );
      this.state = { val: 0 };
      this.increase = this.increase.bind( this );
      this.decrease = this.decrease.bind( this );
   }

   increase( e ) {
      e.stopPropagation();
      let { val } = this.state;
      val++;
      this.setState( { val } );
   }

   decrease( e ) {
      e.stopPropagation();
      let { val } = this.state;
      val--;
      this.setState( { val } );
   }

```

```
49    render() {
50       return (
51          <div>
52             <p>Current score: { this.state.val }</p>
53             <button onClick={ this.increase }>+</button>
54             <button onClick={ this.decrease }>-</button>
55          </div>
56       );
57    }
58 }
59 export default Counter;
```

When defining a custom constructor for a class component it is necessary to call *super(props)* first. To define an initial state of the component we can set *this.state = { val: 0 }*. This is the only place where state should modified directly. In comparison to that in lines 43 and 50 we first create a copy of the value we want to modify by using Object Destructuring and then use *this.setState({ val })* to update it. By doing so we do not modify the state object directly.

In lines 31 and 32 we bind the this keyword on the increase and decrease functions. By leaving this step, accessing *this.setState()* in any of the methods would crash the application as *this* would be the global window object which doesn't have a *setState* method defined.

The render method defines the output of the component. We'll render a paragraph telling the current score by accessing the state together with two buttons and their respective click handlers.

Updating Components

React is known for its good performance [Spu20] even in large applications. To understand better how it achieves this, we will look a bit at the internal process of rendering and updating the components.

On the initial render React will create a component tree from which it'll then build the DOM that the browser converts into displayable objects and paints them on the screen. To show how react determines which part of the DOM it has to update, we'll quickly walk through the previous example:

When clicking the increase or decrease button in the previous example we update the components state which will automatically trigger a re-render. In such a small component it wouldn't really matter if React simply rendered the whole component. In a component containing hundreds of lines and probably many other sub-components the decision to render all of just because, would take a long time and be fatal for performance especially if really all we'd have to do is re-render line 52.

Finding The Differences

(maybe when coming back to this check https://reactjs.org/docs/faq-internals.html https://reactjs.org/d and https://github.com/acdlite/react-fiber-architecture and THIS:: https://www.youtube.com/watch?v) To know which lines to update, React performs a few steps:

- The *setState* function will mark the component and all its children as dirty. [Kur17]
- It'll recursively walk through all components marked as dirty, building them in the virtual DOM

- For each built element it compares it to the value in the actual DOM. If they differ it'll deduce if the item should be replaced, removed or updated and does so accordingly.

By doing so, instead of re-rendering the whole *Counter* component it only re-renders

```
<p>Current score: { this.state.val } </p>
```

Stateless Functional Components

It is also possible to create stateless functional components. In their pure form they do not contain state and normally only show either static data or data they get passed through props. If we look at the *App* component in the above examples we'll find that it does exactly that. Knowing this, we could re-write it:

```
// App.js
import React from 'react';
import Counter from './Counter';

function App( props ) {
   return (
      <div>
         { props.text }
         <Counter/>
      </div>
   );
}
export default App;
```

Which is a lot shorter and has another big advantage: It protects from laziness. [Hou16] As this component doesn't support local state it is not too tempting to quickly hack something new into it. Rather the programmer gets encouraged to think about the structure and create a proper component for a new feature together with its own state object that only that component needs.

And of course visually the result is equal to the one in **2.8** and **2.9**.

2.6 Client - ApolloClient

ApolloClient is a state management library for JavaScript that manages data with GraphQL. It offers an all in one solution for fetching, caching and modifying application data and together with automatic UI updates upon events from the server. [Grab]

Hooks

ApolloClient 3 offers this by providing custom **hooks**. Hooks are a new addition to React since React 16.8 and were introduced to improve stateless functional components [Facb] and can only be used in such. [Faca] There were a few reasons that led to the introduction of hooks like the appearance of complex class components that couldn't be split into smaller ones, not re-usable stateful logic and classes being not ideal for future optimization. [Facb] Hooks allow for example the usage of state in functional components.

Apollo Hooks

In addition to some built-in Hooks provided by React it is also possible to create them. Apollo implemented many own hooks, we will focus on *useQuery*, *useLazyQuery* and *useMutation*. The first argument to all of these is a GraphQL string. A query for the first two, a mutation for the last one.

The second argument is the options object. By adding properties to this the execution behavior can be influenced. Probably the most important argument is *variables*. This is an object containing key-pair values that equal to the ones used in a GraphQL query as shown in **2.1**. In addition to that there are *onError* and *onCompleted*. These are two callback functions that allow for executing actions upon completion or handling of possible errors. Imagine *GET_DATA* being a valid GraphQL query string:

```
function Test() {
   useQuery( GET_DATA, {
      variables: { id: 1 },
      onCompleted: data => console.log( data );
      onError: error => console.log( error.message );
   } );
   ...
}
```

For the rest of the arguments please refer to API reference form Apollo.

To be able to use returned data and inform the user about the current status, all of these three hooks return a few other objects, the most important being *data*, *loading* and *error*. These are boolean values and allow for conditional rendering inside a component, depending on their values:

```
function Test() {
   const { data, loading, error } = useQuery( GET_DATA, {
      ...
   } );

   if ( loading ) return 'Fetching data.';
   if ( error ) return 'Error when fetching data.';
   return 'Success!';
}
```

In comparison to *useQuery* the other two also return a function object that can be called when we wish to execute the query or mutation (see example below).

The *useMutation* hook is the only one of the three allowing for an *update* argument in the options object. In this we can access the local cache and have access to the results returned from the mutation. This can be useful if we need to update internal data depending on the result of a server operation:

Listing 2.10: Creating a Mutation

```
function Mutate() {
   const [ runMutation, { data, loading, error } ] = useMutation( DO_THING, {
      update: ( cache, { data } ) => {
         // update cache with return data from mutation
      }
```

```
6     } );
7  }
```

This approach works well and is more or less the only way of handling return results from external mutations.

Local Resolvers

We could do the same for local state changes, but as these functions can become long it might clutter the component with a lot of code that is not actually for it. To get rid of this problem we can define local resolvers in the client instance:

Listing 2.11: Local Resolvers

```
1  const client = new ApolloClient({
2     ...,
3     resolvers: {
4        Mutation: {
5           setData: ( _, variables, { cache } ) => {
6              // manage state update
7           },
8           ...
9        },
10    },
11    ...,
12 });
```

Just like on the server side we set up resolvers and can access query variables to it. To call it, we also define a GraphQL query like so:

```
1  const SET_DATA = gql`
2     mutation SetData($data: [String]!) {
3        setData(data: $data) @client
4     }
5  `;
```

The name in line 3 has to match with the one defined in the resolver in **2.11** in line 5. But whats even more important is the *@client* directive in the end of line 3. This tells Apollo to not contact the server, but rather resolve the mutation locally. We can then call the above defined mutation like this:

```
1  const Test() {
2     const [ runSetData ] = useMutation( SET_DATA );
3
4     const handleClick = e => {
5        e.stopPropagation();
6        runSetData( { variables: { x: 'test' } } );
7     }
8
9     return (
10       <button onClick={ handleClick }>Click me</button>
11    );
12 }
```

The Apollo Cache

A unique feature of Apollo is its smart cache. Once a query to the server has been executed, the results are saved in the cache and should the exact same query be executed again, the cache will first check if it has results for this query saved locally. If so, it'll return them from there, reducing network traffic and improving performance of the whole application.

Of course in some cases it might be necessary to always have the newest data from the cache. The exact behavior can be specified through the *options* object passed to a query or mutation.

The setup time for working with Apollo Client is very short as it offers great default settings that will get any application going quickly. We just looked at the absolute basics of it, it offers features like pagination, subscriptions, optimistic UI and much more. Further it allows the programmer to define all its behavior precisely when he wants to and by that its a great tool to work with.

3 Deployment

While developing in a local environment it is also important to test the application in a production environment as early as possible. Many frameworks make optimizations to improve performance. This might lead to errors that can't be seen in a local setup. Often they also change error behavior. While during development an error might only lead to a warning message, in production the same error might cause the application to crash.

Another reason to go for early deployment is to allow other people to see, use and test the application. Someone who sees the app for the first time or at least doesn't know how things were thought to work will make completely different use of it than a programmer. New issues with the workflow can be discovered and problems that a developer would never have thought of be reported. Further the layout will be tested on different screens, browsers and devices, unveiling unknown layout troubles. Issues discovered early are much easier to fix than those who are integrated deeply into the codebase.

In this chapter we will talk about the tools that were used to deploy the application to a publicly available domain.

3.1 AWS

AWS is a cloud platform from Amazon that offers over 175 different services in 24 different geographic regions around the world to allow for high availability all around the world. [AWS20d]
AWS was launched in 2002 only offering SOAP (a messaging protocol used in distributed systems to exchange information) and XML interfaces. [Roj20] In 2003 the idea of a big system offering services and tools was born as the leadership realized that one of the company's strengths was building and running effective, reliable and scalable data centers. After developing a more precise idea of how this strengths could be used to fulfill a lot of companies' needs the first public launch was in 2006 with *Simple Storage Service* (S3), the *Elastic Compute Cloud* (EC2) and a *Simple Queue Service* (SQS) following shortly after. [Mil16]

Since then the services offered by AWS keep growing and other companies like Google, Microsoft or IBM started to compete in the business. In a 2019 research from Gartner AWS was found as one of the leading cloud Infrastructure as a Service providers world wide (IaaS) [BGSW19] regarding *Completeness of Vision* and *Ability to execute*.

3.1.1 AWS-EC2

EC2 is a service that allows the creation of virtual machines with over 300 different variations of computational capacities like the number of CPUs, Memory, Storage and Network Performance. [AWS20a] Available configurations that can run on an instance are called Amazon Machine Images (AMIs) and include various Linux based distributions like Ubuntu, Debian or fedora as well as Windows. In addition to empty operating systems Amazon and different communities also provides images with certain pre-installed software making sure that instances can be set up quickly.
Basically, an EC2 instance is a virtual machine that runs on an Amazon server. [Sri19]

3.1.2 AWS-ECS

ECS is a service to run automatically managed containers on AWS servers. Many big companies make use of this service because of its scalability, reliability and security. [AWS20a] These are the most important terms when talking about AWS ECS:

A **definition** is the blueprint of a task, specifying which Docker image(s) to use, ports to expose, can set environment variables and memory needed etc. [Fra18]

A **task** is an instance that is created following the specifications in the task definition. A task can run various Docker containers at once. A task is where an application actually gets executed.

A **service** is used to manage tasks. It is possible to specify a minimum and maximum number of running tasks and when to start new or stop running tasks to make sure the necessary calculation power is always necessary. A service can make use of a load balancer which will distribute network traffic equally over all running tasks in the service.

A **Cluster** is a logical grouping of services.

These components can be visually put together as shown in figure **3.1**. These components can be visually put together as shown in the figure below. (https://docs.aws.amazon.com/AmazonECS/l

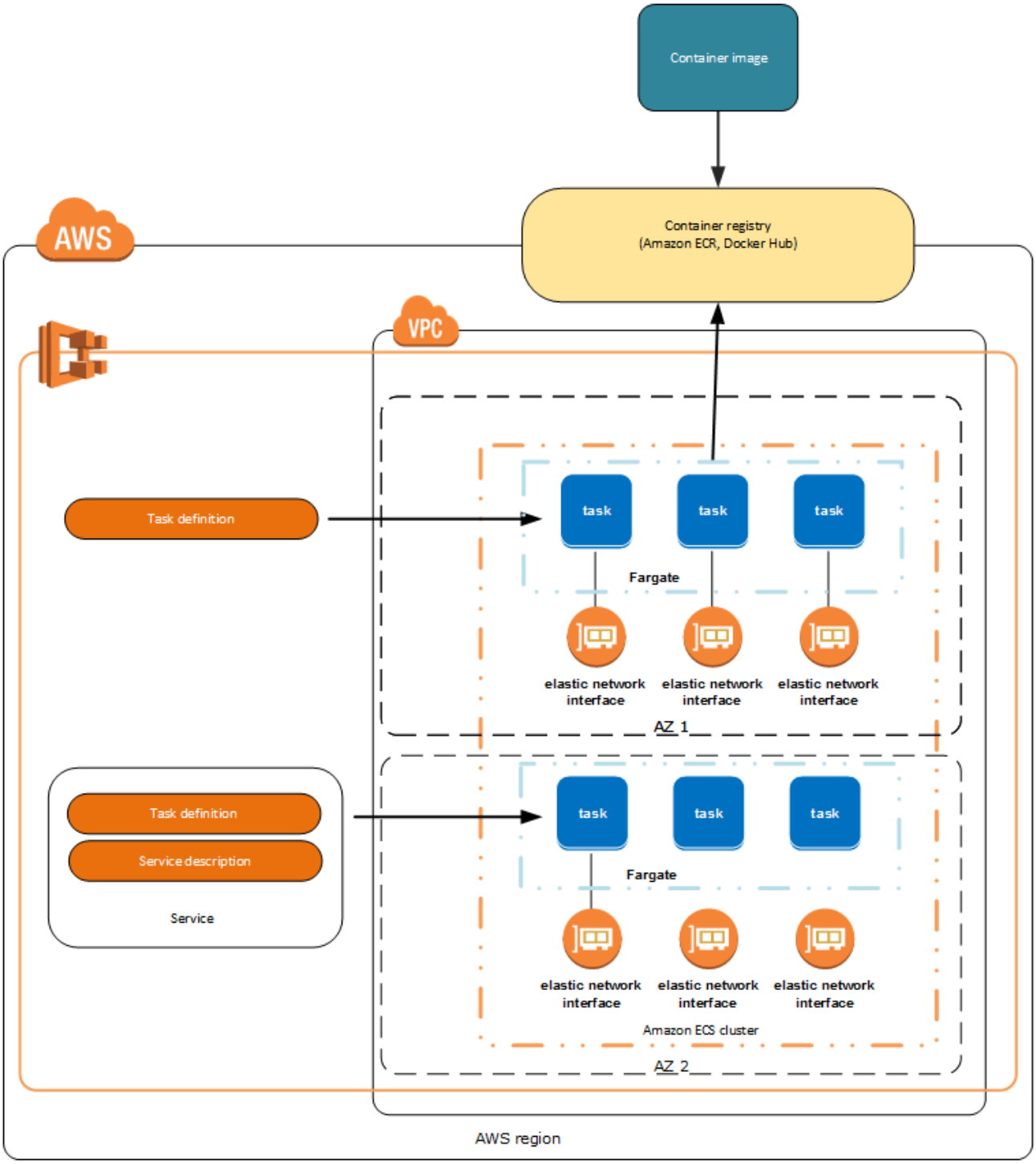

Figure 3.1: Strucute of AWS ECS

The orange square symbolizes ECS inside the big AWS infrastructure. We will ignore the square denoted with "VPC" as it is not relevant for further understanding.
A task can be created from a task definition and be placed in a service. The services are here shown as light blue squares labeled *Fargate*. Fargate is a computation engine from Amazon which calculates which virtual machine to use according to the task definition and creates and starts it and then runs the docker image from a specified container registry automatically. [AWS20c] The orange square with dots and dashes is a cluster. This cluster contains 2 services, reach running 3 tasks.

The services of a cluster can be run in different *Availability Zones* (AZ), meaning that they run on servers from AWS located in different parts of a geological region.

3.1.3 AWS-Amplify

Amplify is a services that aims to make the deployment as well as the development of applications as easy as possible. It comes with many advanced features that help setting up an app quickly. Some examples are: Authentication, API creation, Analytics, Push Notifications and many more [AWS20b]. It creates a certificate for any deployed application, allowing for HTTPS connections. Further it is able to scan a connected GitHub repository and provide a template configuration based on the framework used.

Another really nice feature is the easy deployment flow: For each application it is possible to connect various branches. Each branch will have its unique URL and once a developer pushes to a connected branch, Amplify will build the newest status of the application. This is a great feature for testing new features in a production environment without having to deploy to a master server directly.

3.2 Docker

Docker is a platform that provides the ability to a developer to create any environment in which he wants to execute code in. Before comparing it to a virtual machine we'll take a look at the basic terminology:

A *Docker File* is a text file containing instructions that tell the Docker Engine how to build a *Docker Image*. [Jan17]

This *Docker Image* is a file containing necessary data to execute a given program. The in the docker file specified libraries are saved, folders from the local machine are copied, environment variables are set etc..

When running the image, we will get a *Docker Container*. It will execute specified commands and by that for example download node modules. This container is an instance executed by the *Docker Engine* which runs on top of the Host OS of an actual machine.

While it is similar to a virtual machine, there are certain differences between them:
Operating System Every VM comes with its own operating system, making it heavy in terms of memory and processing power they require. Docker containers in turn all share the hosts operating system and only require the docker engine to be installed on the machine: [Avi19]

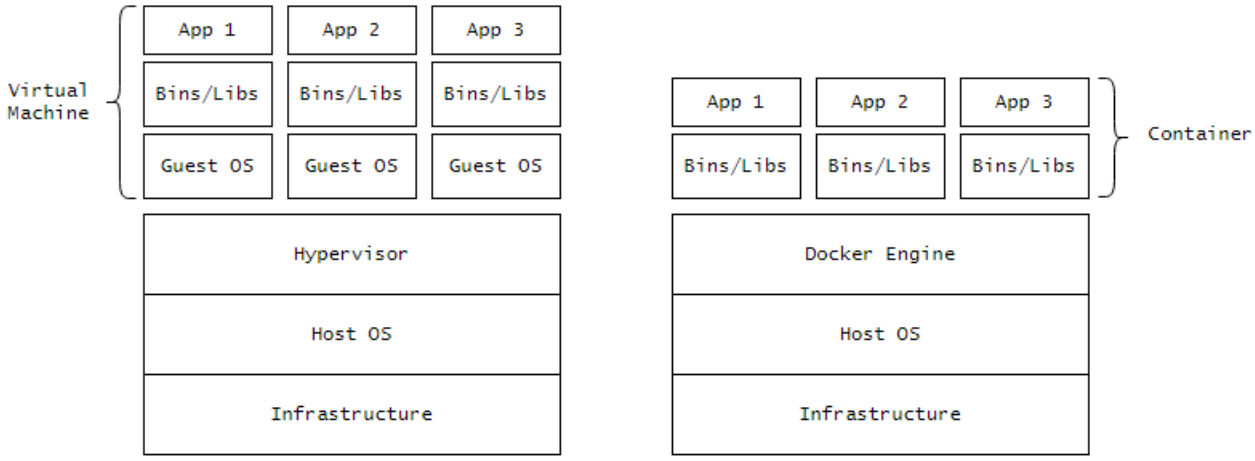

Figure 3.2: Docker vs VM

Security Following the previous point, every VM has its own operating system and is strongly isolated in the host kernel. Docker containers all run on a single kernel. Furthermore docker resources are shared. If an attacker gets access to one container he'll be able to exploit all containers in a cluster. [Avi19])
Portability Containers can easily be ported to any machine that has the docker engine installed. There is no further configuration necessary, they'll the same on any machine. VMs are more difficult to port just due to their sheer size. In addition, the process of setting up a virtual machine differs from operating system to operating system.

4 Development

In this chapter I will talk about the steps in development and explain some thought processes on the way. There might and probably will be parts where you see flaws in my work. This is intended to keep you from making the same mistakes when starting to develop an application with the GRANDstack.

This chapter will also contain a few difficulties I encountered and how I managed to get them working. Again, this part is not supposed to be a "How to" kind of guide - I was completely new to all of the technologies used when starting this project - but rather to document some of my decisions and the reason why I made them.

4.1 Why the GRANDstack

To find out which tech stack to use, the first step was to analyze what the application should be able to do:

- The app will be interactive, not purely informational
- The user will create, modify and delete data
- Local changes made to data can be saved permanently
- Data must be displayed as graph
- At some point we might want to be able to implement real time updates for changes made by another user

This will give a list of requirements our tech stack should fulfill:

- Updating the DOM should be done efficiently as the layout will change depending on user actions
- A local state management will be needed to keep track of local changes
- A database will be necessary to permanently save changes
- An intuitive way for saving graph data
- Communication with the server should be lightweight and easy to implement

Having looked at the individual parts of the GRANDstack, you can see quickly that it contains most of the things I imagined to be useful: Neo4j is a graph database, actually saves a graph on disk and is extremely performant. GraphQL will allows to only transmit the data needed in case of a lot of communication with the server. Together with Apollo, which offers a lot of help when implementing subscriptions, live updates should not be too hard to realize. Having a state management framework and a backend server that are designed to work together also promises for a trouble free development. The Apollo hooks make it very intuitive to update the UI according to state changes.

Furthermore this stack contains many relatively young technologies and as mentioned in chapter 1 this project also serves to explore these.

4.2 Getting started with Neo4j

When starting with Neo4j I wanted to first create a small basic data set that wouldn't be too complicated so I could experiment with it. Having this said I came up with this network:

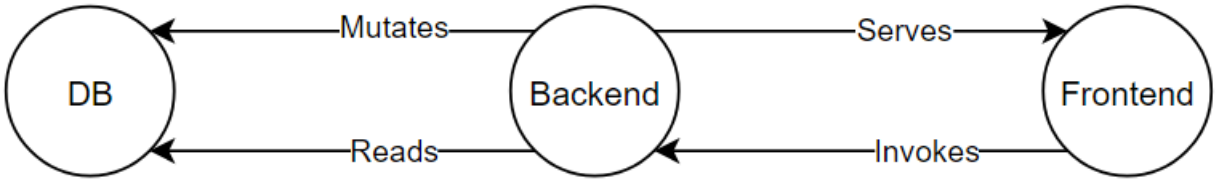

Figure 4.1: A small network of software components

This is the network I would like to display in the application. It helped me a lot to first draw it on paper and then think of the individual components needed. As there will be one graph in the database and one in the application it is important to carefully separate them when talking about a node: This could refer to one either in the database or in the application. While the node in the app is always a node in the database, this is not necessarily the case the other way around as a node in the database could also represent data about a connection in the application.

I created the three nodes for DB, Backend and Frontend using the following code in the Neo4j Browser:

```
1 CREATE (:Node:UserInterface {id: randomUUID(), label: "Frontend", story:
    "Interaction point for the user", nodeType: "UserInterface"})
2 CREATE (:Node:API {id: randomUUID(), label: "Backend", story: "Endpoint for
    requests, fetches from and mutates data on the DB", nodeType: "API"})
3 CREATE (:Node:Persistence {id: randomUUID(), label: "DB", story: "Saves data for
    the methodical designer", nodeType: "Persistence"})
```

Everything between the parentheses of a create statement defines a new node in Neo4j. By using colons I defined labels for a node. These can be used to filter for a certain type of node and can improve performance when traversing.

All of them are of type node and each of them has its own specific label, marking them as different node types. By using curly braces I defined properties on each of them: A unique ID for each node by using Neo4j's *randomUUID()* function, a label which will be displayed in the application and a story which shortly describes its functionality for each node. Furthermore I gave each node a nodeType. This might seem a bit counter intuitive as this was already defined in the labels for the node. However later when retrieving the nodes from the database it'd only be possible to retrieve *all* labels of a node by using Neo4j's *RETURN labels(n)* function. This would return a list, but for displaying them I'd only need one which is why I chose to include this information in the properties as well.

This is how the created graph looks like in the Neo4j Browser:

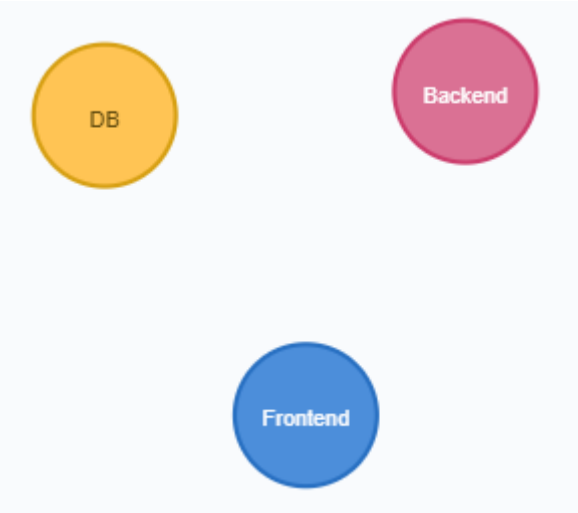

Figure 4.2: The nodes in Neo4j Browser

Before directly creating the respective connections between the nodes to complete the data for the image I first thought about what properties such a link might contain later:

- Data about the x-end of the connection (arrow-type, color, an annotation about how the x-node component sees the connection)
- The same for the y-end
- Information about if the link is part of a sequence of steps and if so a short description of the step
- A label, id, story and linkType
- The IDs of the nodes it connects

I thought of saving all this information on the relationship between the nodes. But if the user now updates only the label of a node and saves this change, the frontend will send all fields and the DB will have to write all other properties as well without actually modifying them.

To somewhat minimize the amount of data sent I decided to split the information among various nodes:

- One link-node that containing the label, id, story and linkType
- Two link-end-nodes containing visual information about the arrow as well as a "note" field to clarify how one node might perceive the connection
- One sequence-node to specify whether or not the node is part of a sequence

The link-node would have direct connections to the components it connects. The sequence- and link-end-nodes would be connected to the link-node and can be accessed by finding the link and from there looking for respective connections.

Having this in mind I went on to create the link-nodes and attach them to their components. At first I didn't create any sequence or link-end nodes to keep things simple:

```
MATCH (api:API) WHERE api.label = "Backend"
MATCH (db:Persistence) WHERE db.label = "DB"
CREATE (l:Link)
SET l.label = "Mutates", l.linkType = "Mutate", l.story = "See JIRA for details:
    https://.."
CREATE (api)<-[:IS_X]-(l)-[:IS_Y]->(db)
```

This code will find a node with an *API* label and one with a *Persistence* label. We can be sure that this will find the correct nodes in this example because only one of each type exists. Normally the node ID would be necessary to identify it. Putting a string in front of the first colon in the match case will declare a variable name for the found node which can be used later to reference it. Then the code will create a link from api to DB, giving it a *Link* label and assigning the link the variable name *l*. Using this variable I set the link's properties label, linkType and story. This time the story serves as reference to an external document describing the relationship in more detail. Then using another create statement Neo4j will create a connection between the link-node (a Neo4j node representing a link in the application) and the node-node (a Neo4j node representing a node in the application). After executing the Neo4j Browser will show the following image:

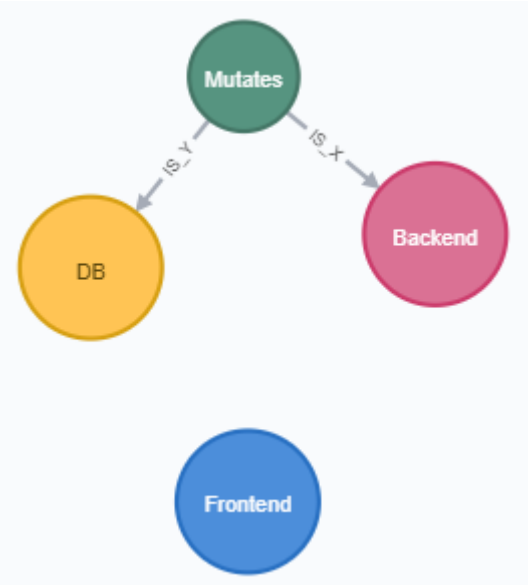

Figure 4.3: After creating the first link-node

After adding the rest of the connection using the same approach the whole representation of **figure 4.1** on the data layer looks like:

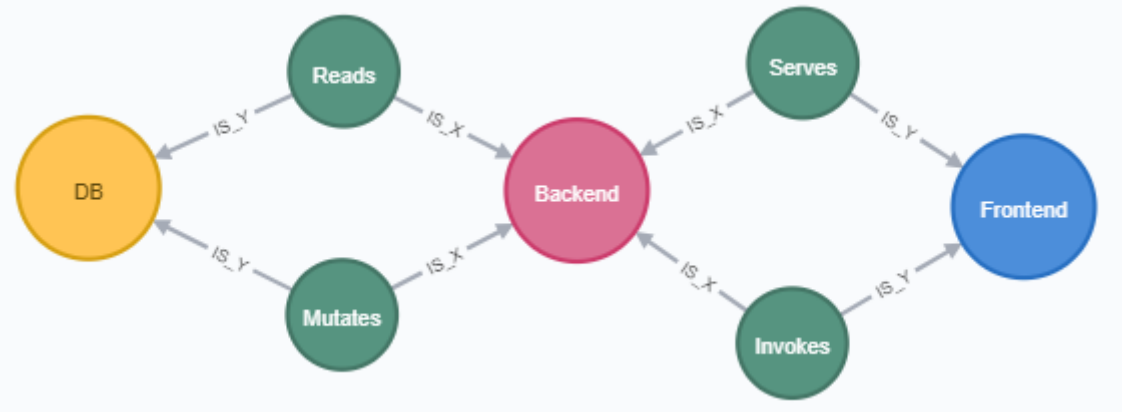

Figure 4.4: Whole graph in the DB

Its important that directions of the arrows here do not represent the direction of the link in the image. They are only necessary because Neo4j doesn't allow for undirected links in create statements. To make it uniform we made all relationships to be outgoing from the link nodes.

4.3 The GraphQL Schema

Development of applications that leverage GraphQL usually start by creating a schema to define how data looks like. The good thing about this is that it can be adapted if during the process we discover that we need to make some changes to it.

Here I want to take a brief look at the GraphQL schema this app uses and mention a few things that didn't work out as I thought they would.

Listing 4.1: GraphQL Enums

```
enum NodeType{
  API
  Command
  Query
  Event
  Persistence
  AbstractUserInterface
  Object
  Computation
  Container
  Domain
  Invariant
  ArchitecturalDecisionRecord
  Definition
}

enum LinkType{
  PartOf
  Trigger
  Read
  Mutate
  Generic
```

```
}
```

The value of *NodeType* or *LinkType* when creating or updating a node or link sent to the server must be one of these. Any other value will result in an error.

```
enum ArrowType{
  Default
  none
  SharpArrow
  Curve
  Diamond
  Arrow
  Box
  Triangle
  Bar
  InvTriangle
}
```

These are the values an arrow on a link displayed in the application can have. The actual values in this enum depend highly on the options from the canvas library used (in this example visjs).

```
interface IDisplayable{
  """
  Minimal data necessary for the object to appear on screen
  """
  id: ID!
  label: String
  story: URI
}

type Node implements INode & IDisplayable{
  id: ID!
  label: String!
  nodeType: NodeType!
  story: URI
  Links: [Link] @cypher(statement: "MATCH (this)--(l:Link) RETURN l")
  synchronous: Boolean
  unreliable: Boolean
  connectedTo: [Node] @cypher(statement: "MATCH (this)--(:Link)--(n:Node) return
      n")
}

type Link implements ILink & IDisplayable{
  id: ID!
  label: String!
  linkType: LinkType!
  x: Node! @cypher(statement: "MATCH (this)-[:X_NODE]->(n:Node) RETURN n")
  y: Node! @cypher(statement: "MATCH (this)-[:Y_NODE]->(n:Node) RETURN n")
  x_end: LinkEnd @cypher(statement: "MATCH (this)-[:X_END]->(le:LinkEnd) RETURN
      le")
  y_end: LinkEnd @cypher(statement: "MATCH (this)-[:Y_END]->(le:LinkEnd) RETURN
```

```
    le")
  sequence: SequenceProperty @cypher(statement: "MATCH
    (this)-[:IS]->(s:Sequence) RETURN s")
  story: URI
  optional: Boolean
}
```

Coming from an Object Oriented background I wanted to use interfaces. Having an ID, label and story object to be apparent on all entities that can be shown on screen makes sense. What I didn't know when creating the schema is that visjs requires each link to be connected to an x and y node, in other words: a link can't "float". This means that there are two more required fields in the ILink interface, making the IDisplayable interface more or less loosing its purpose of defining properties that are required to be displayed.

To retrieve the links attached to the node we make use of the *@cypher* directive we talked about earlier. This is an array which can be empty as nodes do not necessarily need be connected in any way. Same counts for the array of connected nodes. The two booleans *synchronous* and *unreliable* were subject to a lot of discussion as they could also be placed on links rather that nodes. In the end I decided to place them on nodes as it made more sense to me that for example an event would be synchronous rather than the dispatch call for it.

optional on link is a value that can be used to clarify that a link of a sequence will only be done in certain cases.

```
type SequenceProperty{
  group: String
  seq: Int
}

input NodeInput{
  label: String
  story: URI
  nodeType: NodeType
  synchronous: Boolean
  unreliable: Boolean
}

input NodeCreateInput{
  synchronous: Boolean
  unreliable: Boolean
  story: URI
}

type LinkEnd{
  note: String
  arrow: ArrowType
}
```

Every sequence on a link can contain a string identifying the sequence group it belongs to as well as a sequence number to display the order in which steps will be executed. Each link end can contain a note about how the component on the respective end perceives the incoming connection.

The input types are used to be able to pass an object into GraphQL queries or mutations instead of just primitive datatypes. This idea sounds promising as we can just pass all input objects from a form to the query. But as you can see in the above code snippet there is one input type for creating and one for updating nodes. The reason for this decision was that I'd pass all optional parameters as an input object and the required ones manually. In the end it turned out that this led to a lot of object modifying just to get the right structure from the input types.

This schema could be improved by using GraphQL fragments and make it shorted by that, but I'd still avoid the usage of these input types and instead pass all arguments directly.

Listing 4.2: Mutation Type Definition

```
1  \label{exMutations}
2  type Mutation{
3    SeedDB: seedReturn
4    CreateNode(id: ID!, label: String!, nodeType: NodeType!, props:
         NodeCreateInput): NodeOperationReturn
5    CreateLink(id: ID!, label: String!, x_id: ID!, y_id: ID!, linkType: LinkType!,
         props: LinkCreateInput): LinkOperationReturn
6    CreateSequence(link_id: ID!, props: SequencePropertyInput):
         SequenceOperationReturn
7    CreateLinkEnd(link_id: ID!, props: LinkEndInput): LinkEndOperationReturn
8
9    MergeSequence(link_id: ID!, props: SequencePropertyInput):
         SequenceOperationReturn
10   MergeLinkEnd(link_id: ID!, props: LinkEndInput): LinkEndOperationReturn
11
12   UpdateNode(id: ID!, props: NodeInput): NodeOperationReturn
13   UpdateLink(id: ID!, props: LinkInput): LinkOperationReturn
14   UpdateSequence(link_id: ID!, props: SequencePropertyInput):
         SequenceOperationReturn
15   UpdateLinkEnd(link_id: ID!, props: LinkEndInput): LinkEndOperationReturn
16
17   DeleteNode(id: ID!): deleteReturn
18   DeleteLink(id: ID!): deleteReturn
19   DeleteSequence(link_id: ID!): deleteReturn
20   DeleteLinkEnd(link_id: ID!, xy: String!): deleteReturn
21
22   RequestEditRights: EditRightOperationReturn
23   FreeEditRights: EditRightOperationReturn
24 }
```

This is the definition of the root mutation type. It clarifies even more the previously made point of having a lot of trouble formatting the inputs for a mutation: When dispatching a *CreateNode* mutation 2 (label and nodeType) of the form inputs have to be passed directly, while all others need to be put into an object called props that contains the fields defined in the respective input type. This makes the code for form handling a lot harder to re-use, especially as the input types vary for updating and creating.

Lets talk about the return types I defined for each operation:

```
1  interface IReturnInfo{
2    success: Boolean!
```

```
  message: String
}
type NodeOperationReturn implements IReturnInfo {
  success: Boolean!
  message: String
  node: Node
}
type LinkOperationReturn implements IReturnInfo{
  success: Boolean!
  message: String
  link: Link
}
```

Having built a small project using a REST-API before starting this app, this is how I imagined the process:

- Frontend dispatches a call to the server
- Server starts resolving
- Resolving is successful
  - Server sets *success* to true and adds the created object to the payload
- Resolving fails
  - Server sets *success* to false and adds an error message
- Frontend checks *success* and depending on its value will either display data about the modified object(s) or the error message defined by the server

Maybe it was due to my little experience but it didn't seem intuitive to me at all how to define custom responses from the ApolloServer to the client. Just by accident I found a way to return custom error messages, but it was already late in development process and there was not enough time left to build it into all places where it would've been necessary. More about this in chapter 5.

```
type Query{
  Nodes: [Node]
  Links: [Link]
  IsProjectBeingEdited: EditRightQueryReturn
}

schema {
  mutation: Mutation
  query: Query
}
```

The end of the schema contains the definition for the query type and the root schema.

4.4 Communicating with the DB through ApolloServer and GraphQL-Playground

The next step was to communicate with the database through the server. After setting up a small ApolloServer (more on setup in chapter 6) I created a resolver to seed the database with some default data similar to the one shown previously in Figure 4.4. This was really helpful as especially in the beginning I made many mistakes which left the dataset in a bad shape and fixing it manually would've taken a lot of time. In Listing 4.2 in line 2 is the GraphQL definition. The resolver looks like the following:

```
1  const seedQuery = require( './seed' );
2  async SeedDB( _, __, ctx ) {
3     const session = ctx.driver.session();
4     const deleteQuery = `
5        MATCH (n) DETACH DELETE n
6     `;
7     await session.run( deleteQuery );
8     await session.run( seedQuery );
9     await session.close();
10    return {
11       success: true,
12    };
13 },
```

The *seedQuery* is a long cypher query defined in an external file. After deleting all nodes and connections in the database we pass using the *deleteQuery* we run it to create new data to use.

To run this resolver and seed our database we will open the GraphQL Playground in the browser and execute the following GraphQL query:

Listing 4.3: Seeding the DB through GraphQL Playground

```
1  mutation seedDB {
2    SeedDB {
3      success
4    }
5  }
```

The name in line 1 is completely optional. What matters is line 2 as this string will be used to identify the correct resolver. Using the curly braces we define a result set. After hitting the play button we will get the expected result:

Listing 4.4: Seeding Result

```
1  {
2    "data": {
3      "SeedDB": {
4        "success": true
5      }
6    }
7  }
```

The next step was retrieving data. To save some time when writing queries I used the neo4j-graphql-js npm package which combines the resolvers and GraphQL schema to an executable schema. This package can generate many queries and mutations based on the schema we provide but still allows the usage of resolvers I define on my own. By doing so, I could fetch nodes without any more coding:

Listing 4.5: Fetching Nodes

```
query nodes {
  Nodes {
    id
    nodeType
    label
  }
}
```

With the result being:

Listing 4.6: Result Set

```
{
  "data": {
    "Nodes": [
      {
        "id": "738e414d-bc1f-4e90-ad76-ec44d34f1a71",
        "nodeType": "AbstractUserInterface",
        "label": "UI"
      },
      {
        "id": "6c4b4dba-5726-47bc-8fb5-10affcf03ef7",
        "nodeType": "API",
        "label": "Server"
      },
      {
        "id": "2554b296-ffed-4028-ad80-1181dfe97ecd",
        "nodeType": "Persistence",
        "label": "NeoDB"
      },
      {
        "id": "ded515d5-8016-4324-a756-201b9e1f2db0",
        "nodeType": "Event",
        "label": "Create Node"
      }
    ]
  }
}
```

Finally I wanted to create a resolver to create a node. The reason why I did this by hand was that I wanted to have control over the Cypher query, to be sure that the properties would be assigned the way I wanted it:

```
    async CreateNode( _, args, ctx ) {
      try {
```

```
3            const session = ctx.driver.session();
4            const query = `
5               CREATE (n:Node:${ args.nodeType } {id: $id, label: $label,
                    nodeType: $nodeType})
6               SET n += $props
7               RETURN n`;
8            const results = await session.run( query, args );
9            await session.close();
10           return {
11              ...defaultRes,
12              node: PrepareReturn( results, 'n', defaultNode ),
13           };
14        }
15        catch ( e ) {
16           return errorRes( e );
17        }
18     }
```

In line 5 I make use of ES6 template strings and use the *args* parameter to create the correct label in the query string, as it is not possible to use query variables in labels in Cypher. This example also shows the usage of query variables in Cypher really well. The *args* object will be destructured and its keys are available to the query by using the *$* sign. This example also shows how to use input types in line 6 to set various properties at once using Cypher. In the GraphQL Playground it can be used like the following:

Listing 4.7: Using the Create Node resolver

```
1  mutation createNode($props: NodeCreateInput) {
2    CreateNode(id: "1", label: "new test", nodeType: Object, props: $props){
3      success
4      node {
5        id
6        label
7      }
8    }
9  }
```

And the *Query Variables* section contains the contents for *props*:

```
1  {
2    "props": {"synchronous": false, "unreliable": false, "story": "test"}
3  }
```

4.5 Making ApolloServer and ApolloClient communicate

4.6 Building the UI

4.6.1 Components

4.7 Problems

4.7.1 Keeping the data consistent when saving changes

4.7.2 AWS-Healthcheck

4.7.3 Apollo Error-Codes

4.7.4 Apollo Chrome Dev-Tools

4.7.5 CORS-problems

4.8 Graph-Layout

Tree-Layout

Flower-Layout

4.9 Behavior Decisions

4.10 Avoiding data corruption through multiple editors at once

4.11 Detect multiple connections between two nodes

5 Looking back

6 Documentation

7 Ideas for the Future

List of Figures

List of Tables

Bibliography

[Avi19] Avi: *Docker vs Virtual Machine - Understanding the Differences*. https://geekflare.com/docker-vs-virtual-machine/. Version: 15.09.2019. – Last visited 08.08.2020

[AWS20a] Amazon Web Services, Inc.: *Amazon EC2*. https://aws.amazon.com/ec2/?nc1=h_ls. Version: 2020. – Last visited 07.08.2020

[AWS20b] Amazon Web Services, Inc.: *AWS Amplify*. https://aws.amazon.com/amplify/. Version: 2020. – Last visited 07.08.2020

[AWS20c] Amazon Web Services, Inc.: *AWS Fargate*. https://aws.amazon.com/fargate/. Version: 2020. – Last visited 07.08.2020

[AWS20d] Amazon Web Services, Inc.: *Cloud computing with AWS*. https://aws.amazon.com/what-is-aws/. Version: 2020. – Last visited 05.08.2020

[BGSW19] Bala, Raj ; Gill, Bob ; Smith, Dennis ; Wright, David: *Magic Quadrant for Cloud Infrastructure as a Service, Worldwide*. https://www.gartner.com/doc/reprints?id=1-1CMAPXNO&ct=190709&st=sb. Version: 16.07.2019. – Last visited 05.08.2020

[Byr15] Byron, Lee: GraphQL: A data query language. (14.09.2015). https://engineering.fb.com/core-data/graphql-a-data-query-language/. – Last visited 30.07.2020

[DbE20a] *DB-Engines Ranking*. https://db-engines.com/en/ranking. Version: 08.2020. – Last visited 23.08.2020

[DbE20b] *DB-Engines Ranking of Graph DBMS*. https://db-engines.com/en/ranking/graph+dbms. Version: 08.2020. – Last visited 23.08.2020

[Dev19] Developer, Microsoft: *Intro to GraphQL, Part 1: What is GraphQL*. Version: 13.09.2019. https://www.youtube.com/watch?v=zvZP0PVAdR0 Last visited 30.07.2020

[Faca] Facebook, Inc.: *Hooks at a Glance*. https://reactjs.org/docs/hooks-overview.html. – Last visited 04.08.2020

[Facb] Facebook, Inc.: *Introducing Hooks*. https://reactjs.org/docs/hooks-intro.html. – Last visited 04.08.2020

[Fac18] Facebook, Inc., 06.2018. http://spec.graphql.org/June2018/

[Fou20] Foundation, GraphQL: *What is GraphQL?* https://foundation.graphql.org/. Version: 2020. – Last visited 30.07.2020

[Fra18] Fraser, Dominic: *A beginner's guide to Amazon's Elastic Container Service*. https://www.freecodecamp.org/news/amazon-ecs-terms-and-architecture-807d8c4960fd/. Version: 20.05.2018. – Last visited 07.08.2020

[Graa] GraphQL, Apollo, https://www.apollographql.com/docs/apollo-server/. – Last visited 30.07.2020

[Grab] GraphQL, Apollo: *Introduction to Apollo Client*. https://www.apollographql.com/docs/react. – Last visited 04.08.2020

[Hou16] House, Cory: *React Stateless Functional Components: Nine Wins You Might Have Overlooked*. https://hackernoon.com/react-stateless-functional-components-nine-wins-you-might-have-overlooked-9 Version: 01.03.2016. – Last visited 04.08.2020

[Jan17] Janetakis, Nick: *Differences between a Dockerfile, Docker Image and Docker Container*. https://nickjanetakis.com/blog/differences-between-a-dockerfile-docker-image-and-docker-container. Version: 08.08.2017. – Last visited 08.08.2020

[Kur17] Kurian, Gethyl G.: *How Virtual-DOM and diffing works in React*. https://medium.com/@gethylgeorge/how-virtual-dom-and-diffing-works-in-react-6fc805f9f84e. Version: 24.01.2017. – Last visited 04.08.2020

[LGK20] Lyon, William ; Graham, Michael ; Klovedal, Viktor S.: *Getting Started With GRANDstack*, 2020. https://grandstack.io/docs/getting-started-neo4j-graphql/. – Last visited 30.07.2020

[Lin12a] Lindaaker, Tobias: *Neo4j Internals*. Version: 25.01.2012. https://skillsmatter.com/skillscasts/2968-neo4j-internals Last visited 30.07.2020

[Lin12b] Lindaaker, Tobias: *Neo4j Internals*. Version: 25.01.2012. https://www.slideshare.net/thobe/an-overview-of-neo4j-internals Last visited 30.07.2020

[Mil16] Miller, Ron: *How AWS came to be*. https://techcrunch.com/2016/07/02/andy-jassys-brief-history-of-the-genesis-of-aws/?guccounter=1. Version: 02.07.2016. – Last visited 05.08.2020

[Neoa] Neo4j, https://neo4j.com/docs/api/javascript-driver/4.1/class/src/driver.js~Driver.html. – Last visited 30.07.2020

[Neob] Neo4j: *Concepts: Relational to Graph*, https://neo4j.com/developer/graph-db-vs-rdbms/. – Last visited 30.07.2020

[Neoc] Neo4j: *Cypher Query Language*, https://neo4j.com/developer/cypher/. – Last visited 30.07.2020

[Neod] Neo4j: *Neo4j Graph Database*, https://neo4j.com/developer/neo4j-database/. – Last visited 30.07.2020

[Neo20] Neo4j: *What is a graph database? (in 10 minutes).* Version: 09.06.2020. https://www.youtube.com/watch?v=REVkXVxvMQE Last visited 30.07.2020

[Par20] Parker, Jeff: *Top 10 Network Diagram, Topology and Mapping Software.* https://www.pcwdld.com/top-10-network-diagram-topology-and-mapping-software. Version: 15.01.2020. – Last visited 30.07.2020

[Roj20] Rojas, Alec: *A Brief History of AWS.* https://mediatemple.net/blog/cloud-hosting/brief-history-aws/. Version: 31.08.2020. – Last visited 05.08.2020

[Spu20] Spukas, Linas: *React Workk Phases.* https://dev.to/spukas/react-work-phases-4eaj. Version: 15.03.2020. – Last visited 04.08.2020

[Sri19] Srinivasan, Krishna: *Is EC2 a virtual machine?* https://www.quora.com/Is-EC2-a-virtual-machine. Version: 29.11.2019. – Last visited 05.08.2020

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben, sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Ort, den 01. Januar 2100 .....

Unterschrift des Verfassers

Ermächtigung

Hiermit ermächtige ich die Hochschule Kempten zur Veröffentlichung der Kurzzusammenfassung (Abstract) meiner Arbeit, z. Bsp. auf gedruckten Medien oder auf einer Internetseite.

Ort, den 01. Januar 2100 .....

Unterschrift des Verfassers