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 also take a brief look into how each of the tools works internally. The third chapter will focus on the deployment process and 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

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

There are many tools on the internet for building graphs to visualize data. Famous examples are "ConceptDraw Pro", "Lucidchart" (https://www.pcwdld.com/top-10-network-diagram-topology-and-mapping-software) 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.
Furthermore the purpose of this project is to try out technologies. This software, or variations of it, might later be incorporated into a bigger project. The experiences and impressions can be of help if someone thinks about building similar software.

2 Backend to Frontend

In this chapter we will look at the individual software components this application consists of. 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. (https://grandstack.io/d started-neo4j-graphql) Neo4j is not necessarily designed to work well with the other 4, but combining the "neo4j-driver" and "apollo-server-express" packages makes it almost feel like it was.

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. (https://foundation.graphql.org/)
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 HTML.
After evaluating different common options like RESTful-APIs and FQL there were often the same problems: The ratio of data actually used to the one fetched was very small, the amount of requests (https://www.youtube.com/watch?v=zvZP0PVAdR0) and the amount of code on both server and client side to prepare the data was big. (https://engineering.fb.com/core-data/graphql-a-data-query-language/)
For example, for loading the start page of a single user, there would have been a lot of different requests necessary: (https://www.youtube.com/watch?v=zvZP0PVAdR0)

- *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
- ...

GraphQL aims to do the opposite: Reduce the amount of data transferred, reduce the amount of requests and increase the developer productivity. (https://engineering.fb.com/core-data/graphql-a-data-query-language/)

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:
(https://www.youtube.com/watch?v=zvZP0PVAdR0)

Example 2.2.1
query {

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

Where the answer would be a JSON-string:

Example 2.2.2

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

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 one 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. (http://spec.graphql.org/June2018/)
To be able to run queries in the first place, one 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 **{2.2.1}** 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. (http://spec.graphql.org/June2018)

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 (https://db-engines.com/en/ranking) and the most popular graph database. (https://db-engines.com/en/ranking/graph+dbms)

- Unlike most relational databases, who store data through tables and joins, Neo4j stores data in the form of actual nodes and relationships between such (https://neo4j.com/developer/neo4
database/). In other DBMS relations between items generally are achieved through join-/lookup-tables which have to be generated. (https://neo4j.com/developer/graph-db-vs-rdbms/)

- 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 (https://skillsmatter.com/skillscasts/2968-neo4j-internals around 32:45) 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. (https://www.youtube.com/watch?v=REVkXVxvMQE) 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. (https://neo4j.com/developer/cypher-query-language/)

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 (https://skillsmatter.com/skillscasts/2968-neo4j-internals) and (https://www.slideshare.net/thobe/an-overview-of-neo4j-internals). 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. (Video be ca. 08:27)
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:

Example 2.3.3

```
if node n has relationship pointer r:
  if n is start node of r:
    if r has StartNext pointer sn:
      set r = sn
      repeat from line 2
    endif
    else
      visited all relationships → terminate
    endelse
  endif
```

else
if r has EndNext pointer en:
set r = en
repeat from line 2
endif
else
visited all relationships → 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. (https://skillsmatter.co
neo4j-internals around 32:45) 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 (→ 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 (https://www.apollographql.com/docs/apollo-server/). It can be embedded into Node.js middleware like Express or Fastify 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:

Example 2.4.1

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

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. (https://neo4j.com/docs/api/java driver/4.1/class/src/driver.js Driver.html) 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:

Example 2.4.2

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

which will be executed on this schema

Example 2.4.3

```
# 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:

Example 2.4.4

```
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 **{2.4.2}**, in this case "libraries". In the GraphQL schema under **{2.4.4}** 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:

Example 2.4.5

```
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:

Example 2.4.6

```
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.

2.5 Frontend - React

2.6 Client - ApolloClient

3 Deployment

3.1 AWS

3.1.1 AWS-EC2

3.1.2 AWS-ECS

3.1.3 AWS-Amplify

3.2 Docker

4 Development

4.1 Getting started with Neo4j

4.2 Manipulating the DB through ApolloServer and GraphQL-Playground

4.3 Making ApolloServer and ApolloClient communicate through GraphQL

4.4 Building the UI

4.4.1 Components

4.5 Problems

4.5.1 Keeping the data consistent when saving changes

4.5.2 AWS-Healthcheck

4.5.3 Apollo Error-Codes

4.5.4 Apollo Chrome Dev-Tools

4.5.5 Graph-Layout

Tree-Layout

Flower-Layout

4.6 Behavior Decisions

4.7 Avoiding data corruption through multiple editors at once

4.8 Detect multiple connections between two nodes

4.9 CORS-problems

5 Looking back

6 Documentation

7 Ideas for the Future

List of Figures

List of Tables

Bibliography

[BCW12] Brambilla, Marco ; Cabot, Jordi ; Wimmer, Manuel: *Model-Driven Software Engineering in Practice*. San Rafael : Morgan & Claypool Publishers, 2012 (Synthesis Lectures on Software Engineering). – ISBN 9781608458837

[FMT10] Fischer, Frank ; Mandl, Heinz ; Todorova, Albena: Lehren und Lernen mit neuen Medien. In: Tippelt, Rudolf (Hrsg.) ; Schmidt, Bernhard (Hrsg.): *Handbuch Bildungsforschung*. Wiesbaden : VS, Verl. für Sozialwiss, 2010. – ISBN 978–3–531–15481–7, S. 753–771

[KT08] Kelly, Steven ; Tolvanen, Juha-Pekka: *Domain-specific modeling: Enabling full code generation*. Hoboken, N.J. : John Wiley & Sons, 2008. – ISBN 0470036664

[LC12] Lämmel, Uwe ; Cleve, Jürgen: *Künstliche Intelligenz*. 4. München : Hanser, 2012. – ISBN 978–3–446–42758–7

[LR93] Löhr-Richter, Perdita: Zur Diskussion: Methodologie - Methodik - Methode: Was steckt dahinter? In: *Emisa Forum* 13 (1993), Nr. 1, 39–41. `http://subs.emis.de/LNI/EMISA-Forum/Volume13_1/Emisa_1_93_S39-41.pdf`

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